The Antelope Relational Database System

Datascope: A tutorial

BRTT

BOULDER REAL TIME TECHNOLOGIES, INC.

Printed in the United States of America.

Boulder Real Time Technologies, Inc.
2045 Broadway, Suite 400
Boulder, CO 80302

CHAPTER 1

Overview

Antelope is a collection of software which implements the acquisition, distribution and archive of environmental monitoring data and processing. It provides both automated real time data processing, and offline batch mode and interactive data processing. Major parts of both the real time tools and the offline tools are built on top of the Datascope relational database system. This tutorial explains some basic concepts behind relational database systems and how these concepts appear in Datascope.

Datascope: What is it?

Datascope is a relational database system in which tables are represented by fixed-format files. These files are plain ASCII files; the fields are separated by spaces and each line is a record. The format of the files making up a database is specified in a separate schema file. The system includes simple ways of doing the standard operations on relational database tables: subsets, joins, and sorts. The keys in tables may be simple or compound. Views are easily generated. Indexes are generated automatically to perform joins. General expressions may be evaluated, and can be used as the basis of sorts, joins, and subsets.

The system provides a variety of ways to use the data. There are c, FORTRAN, tcl/tk and perl interfaces to the database routines. There are command line utilities which provide most of the facilities available through the programming libraries.

There are a few GUI tools for editing and exploring a database. And, since the data is typically plain ASCII, it's also possible to just use standard UNIX tools like sed, awk, and vi.

Datascope: Features

- Datascope is small, conceptually simple, and fast.
- Datascope has interfaces to several languages (c, FORTRAN, tcl/tk, perl and MATLAB), a command line interface, and GUI interfaces. These provide a wide range of access methods into databases.
- Datascope does not provide access through a specialized *query language*, such as SQL.
- Datascope provides most of the features of other commercial database systems, including:
  - data independence
  - schema independence
  - view generation through joins, subsets, sorts, and groups
  - automatic table locking to prevent database corruption when multiple users are adding records to a table
- The organization of tables and fields within a Datascope database is specified with a plain text schema file. This schema file, in addition to specifying the fields which make up tables, and the format of individual records in every table, provides a great deal of additional information, including:
  - short and long descriptions of every attribute and relation
  - null values for each attribute
  - a legal range for each attribute
  - units for an attribute
  - primary and alternate keys for relations.
  - foreign keys in a relation

  This additional information is useful for documenting a database, and makes it easier for a newcomer to learn a new database.

- The detailed schema often makes it possible to form the natural joins between tables without explicitly specifying the join conditions.
- Datascope schema files and database tables are stored in normal ASCII files on the UNIX file system. These files can be viewed and edited using normal text editors (although it is inadvisable to hand edit database tables). File access permissions are controlled through the normal UNIX file permissions.
- The keys in Datascope tables may include ranges, like a beginning and an ending time. This is useful, and sometimes essential, for time dependent parameters, like instrument settings. Indexes may be formed on these ranges, and these indexes can considerably speed join operations. (When two tables are joined by time range keys, the join condition is that the time ranges overlap.)
- Datascope has an embedded expression calculator which can be used to form joins, sorts and subsets. This calculator contains many functions which are peculiar to environmental science applications, such as spherical geometry, exhaustive time conversion functions and seismic travel time functions.

Datascope: What is it good for?

Relational database systems are a proven method for representing certain types of information, much more powerful than the traditional grab-bag approach of data files, log files, handwritten notes, and *ad hoc* data formats. Datascope is a general-purpose relational database management system which is ideal for managing the large and complex data volumes that are produced by a modern environmental monitoring network. It is relatively easy and intuitive when compared to other commercial database products. It provides a way of moving from the traditional plethora of formats to a better approach which organizes the data, documents it, and provides powerful tools for manipulating it.

Datascope should be useful to anyone who needs to organize data and is interested in applying relational database technology, but can't afford the time, learning, development, and people resources which most other commercial database systems require.

CHAPTER 2

Test Drive

Learning a database system such as Datascope takes some time and involves at least the following steps:

- learning about relational databases in general
- learning the tools and operations a particular DBMS provides
- learning a particular database schema
- learning a particular database

This chapter gives a whirlwind tour of a small example database, using the general purpose Datascope tool *dbe*. This will get your feet wet, show you quickly how to do a variety of useful things, and get you started learning about relational databases in general, and Datascope in particular.

Datascope was originally developed for seismic applications and the demo database has seismic data. It contains data recorded at seismic stations around the world and parameter data describing those instruments (location, gains, orientation). This is the "raw data" part of the database. In addition, the database contains information which is derived from the raw data, typically information about earthquakes: location, size, and first arrivals of seismic energy from various earthquakes at the various stations.

What is a relational database?

A database can be any collection of information, hopefully organized in some fashion that makes it easy to find a particular piece of information. Relational databases organize the data into multiple *tables*. Each table is made up of *records*, and each record has a fixed set of *fields* (sometime referred to as "*attributes*"). The structure of a database, i.e. the tables and the fields which make up a record, is called the *schema*. The schema for our demo is a variation of a schema developed at the Center for Seismic Studies.

A standard reference text for databases is "An Introduction to Database Systems", by C.J. Date. Start with it if you would like to learn more about relational databases in particular.

dbe: a window on a database

dbe is a general purpose tool for exploring, examining, and editing a relational database. It provides in a single interactive, graphical tool most of the functionality provided by Datascope. Because it is window and menu driven, it is fairly easy to learn. This discussion will lead you through a session with dbe, but probably the best way to learn it is to explore on your own. Follow along with this discussion by running dbe on the demo database that comes with the Antelope distribution and is normally installed in */opt/antelope/data/db/demo.*

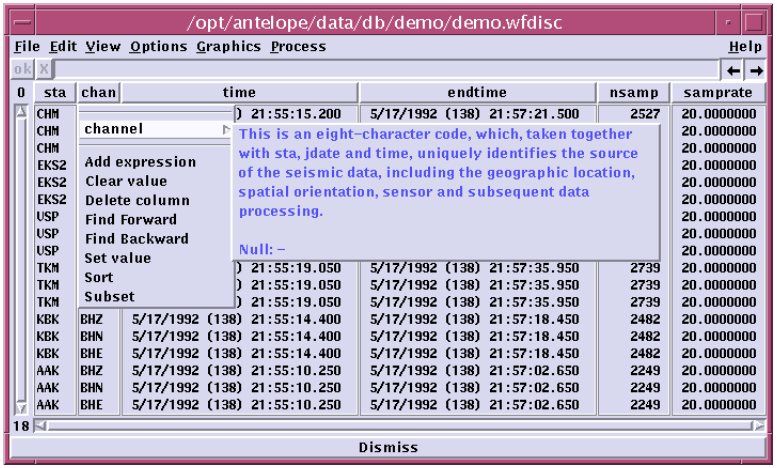
Begin in an empty directory where you can write files, and start dbe:

```
% dbe /opt/antelope/data/db/demo/demo
```

This brings up a database window with multiple buttons, one for each table of the demo database.

Viewing a table

Press the button labeled *wfdisc*. This brings up a new spreadsheet-like window on the *wfdisc* table.



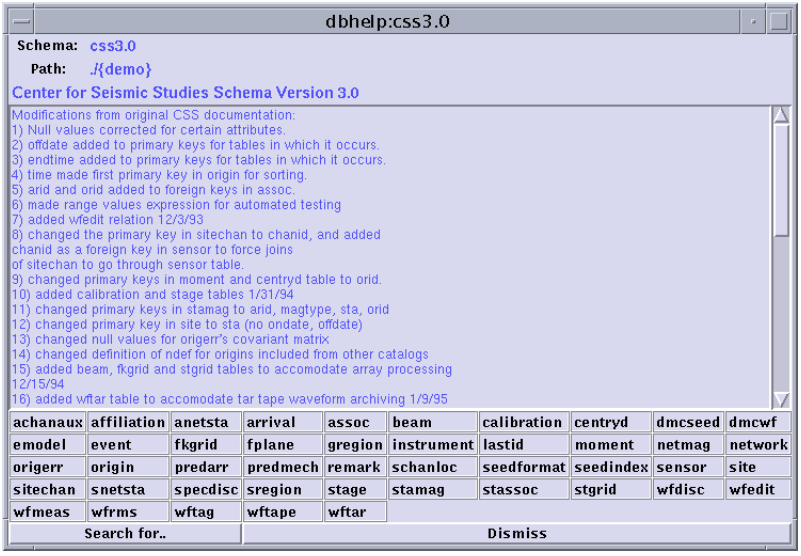
The window title is the name of the table. Beneath it is a menu bar, and directly beneath that is a text entry area. This entry area is used both for directly editing fields, and for various operations which require text input, like entering an expression or searching for a particular value.

The main portion of the window has a column for each field, up to the limit of what will fit on the screen. The scrollbar on the left controls the range of records displayed, while the scrollbar on the bottom may be used to scroll by column, and show the columns which didn't fit on the screen.

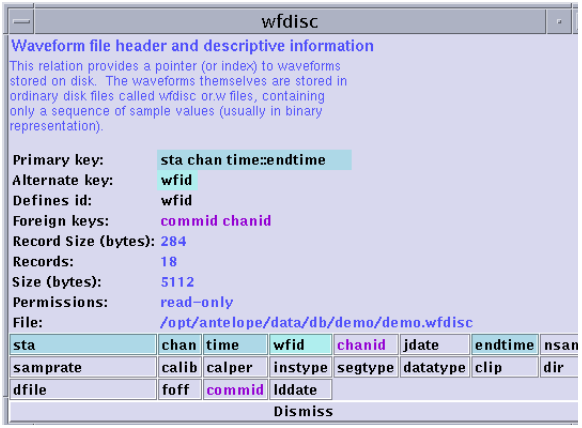
At the top of each column is a column header button showing the field name. These buttons bring up menus which allow several column specific operations like sorting, searching, or editing.

Viewing schema information

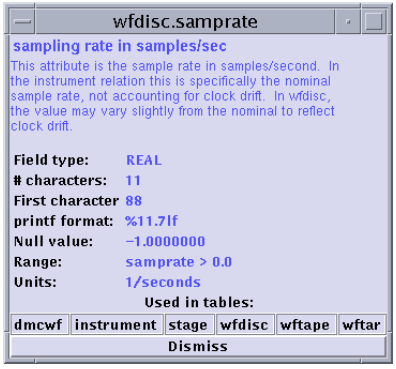
One entry of the header button shows detail information about *field*. There is similar information about the *table* under the *Help->On wfdisc* far right menu of the menubar. For even more information about the schema, try the *Help->On Schema* option; this brings up a window with buttons for each table:



Each table button brings up a window describing that table, showing the keys and other information from the schema. And they contain buttons for each field of the table. Press the *wfdisc* button, bringing up the window for the wfdisc table.



Press a field button to bring up a window showing information about a field. The row of table buttons at the bottom shows each table which uses this field.



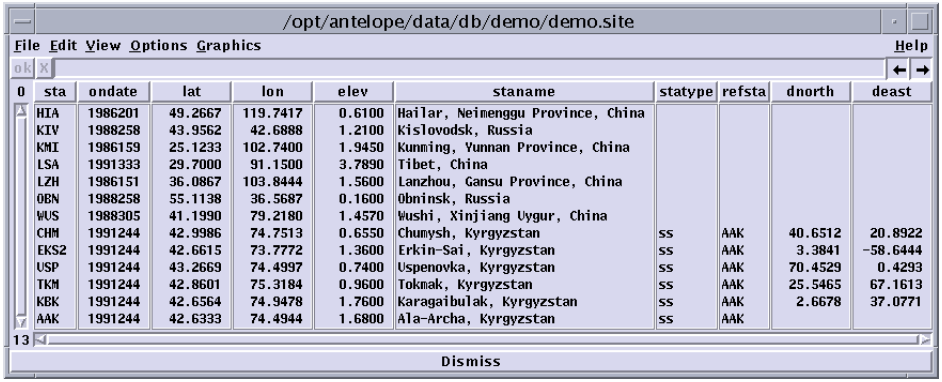
This adjunct to dbe is also available as a separate program, *dbhelp*.

Performing a join

Refer back to the help window for the *wfdisc* table; this table describes external files which contain recorded data from an instrument. The *sta* and *chan* fields specify a particular location and instrument. These fields, plus the time and endtime fields, all taken together, comprise the primary key for the *wfdisc* table. This means that for a particular station, channel and time range, there should be just one row in the *wfdisc* table.

This relates to a very fundamental idea behind relational databases: a particular piece of information resides in only one place. If it needs to be corrected, it need only change in one place. Contrast this with a typical situation where a correction may require updates in many locations; finding all the locations can be a major problem.

The *wfdisc* table provides a reference to the data for a particular instrument at a specific time and location. Notice that a considerable amount of information is missing: where on the globe was this data recorded? That information is not contained in the *wfdisc* table; instead it is kept in another table, the *site* table. Find the original dbe database window, and press the button labeled *site* (or use the menu *File->Open Table->site*).



0	sta	ondate	lat	lon	elev	staname	statype	refsta	dnorth	deast
	HIA	1986201	49.2667	119.7417	0.6100	Hailar, Neimenggu Province, China				
	KIV	1988258	43.9562	42.6888	1.2100	Kislovodsk, Russia				
	KMI	1986159	25.1233	102.7400	1.9450	Kunming, Yunnan Province, China				
	LSA	1991333	29.7000	91.1500	3.7890	Tibet, China				
	LZH	1986151	36.0867	103.8444	1.5600	Lanzhou, Gansu Province, China				
	OBN	1988258	55.1138	36.5687	0.1600	Obninsk, Russia				
	WUS	1988305	41.1990	79.2180	1.4570	Wushi, Xinjiang Uygur, China				
	CHM	1991244	42.9986	74.7513	0.6550	Chumysh, Kyrgyzstan	ss	AAK	40.6512	20.8922
	EKS2	1991244	42.6615	73.7772	1.3600	Erkin-Sai, Kyrgyzstan	ss	AAK	3.3841	-58.6444
	USP	1991244	43.2669	74.4997	0.7400	Uspenovka, Kyrgyzstan	ss	AAK	70.4529	0.4293
	TKM	1991244	42.8601	75.3184	0.9600	Tokmak, Kyrgyzstan	ss	AAK	25.5465	67.1613
	KBK	1991244	42.6564	74.9478	1.7600	Karagaibulak, Kyrgyzstan	ss	AAK	2.6678	37.0771
	AAK	1991244	42.6333	74.4944	1.6800	Ala-Archa, Kyrgyzstan	ss	AAK		

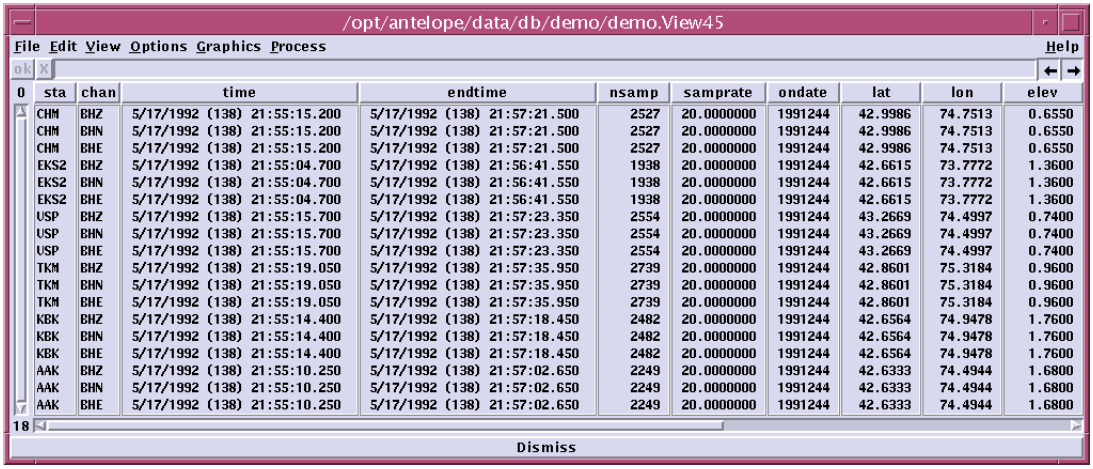
13

In this table, you can find the location at which a particular piece of data was recorded: latitude, longitude, and elevation. If the original elevation was measured incorrectly, it can be corrected here, in just one place. This is an important strength of relational databases, but it is also a problem: the data about location is not kept with the recorded data where it is most convenient during processing. Instead, when you need the location, you must look it up in the site table.

Looking up information in the site table is simplified by a relational operation called a ***join***. This means creating a new composite table composed of columns from other tables. In this particular case, we want to join wfdisc with site. Go back to the wfdisc window, and under the view menu, select “join->site”. The wfdisc window disappears, and a new window appears. This window contains a *view* into a table which is the join of wfdisc with site.

What about the join conditions?

Conceptually, the join operation may be viewed as combining every row of the first table with every row of the second table, but only keeping combinations which satisfy some condition. For this particular join, the condition to be satisfied is: station ids match, and the time range of the wfdisc row matches (overlaps) the time range of the site row. In most RDBMS (Relational DataBase Management Systems), you would need to specify this condition explicitly, but Datascope is able to infer and provide the join condition in many cases. The chapter on *Basic Datascope Operations* describes how this is accomplished.



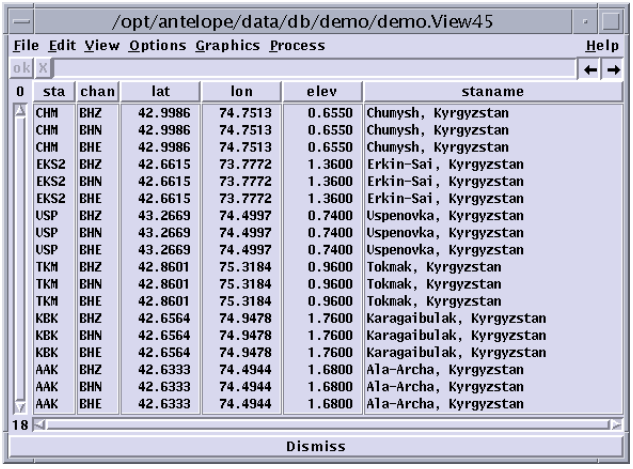
0	sta	chan	time	endtime	nsamp	samprate	ondate	lat	lon	elev
	CHM	BHZ	5/17/1992 (138) 21:55:15.200	5/17/1992 (138) 21:57:21.500	2527	20.0000000	1991244	42.9986	74.7513	0.6550
	CHM	BHN	5/17/1992 (138) 21:55:15.200	5/17/1992 (138) 21:57:21.500	2527	20.0000000	1991244	42.9986	74.7513	0.6550
	CHM	BHE	5/17/1992 (138) 21:55:15.200	5/17/1992 (138) 21:57:21.500	2527	20.0000000	1991244	42.9986	74.7513	0.6550
	EKS2	BHZ	5/17/1992 (138) 21:55:04.700	5/17/1992 (138) 21:56:41.550	1938	20.0000000	1991244	42.6615	73.7772	1.3600
	EKS2	BHN	5/17/1992 (138) 21:55:04.700	5/17/1992 (138) 21:56:41.550	1938	20.0000000	1991244	42.6615	73.7772	1.3600
	EKS2	BHE	5/17/1992 (138) 21:55:04.700	5/17/1992 (138) 21:56:41.550	1938	20.0000000	1991244	42.6615	73.7772	1.3600
	USP	BHZ	5/17/1992 (138) 21:55:15.700	5/17/1992 (138) 21:57:23.350	2554	20.0000000	1991244	43.2669	74.4997	0.7400
	USP	BHN	5/17/1992 (138) 21:55:15.700	5/17/1992 (138) 21:57:23.350	2554	20.0000000	1991244	43.2669	74.4997	0.7400
	USP	BHE	5/17/1992 (138) 21:55:15.700	5/17/1992 (138) 21:57:23.350	2554	20.0000000	1991244	43.2669	74.4997	0.7400
	TKM	BHZ	5/17/1992 (138) 21:55:19.050	5/17/1992 (138) 21:57:35.950	2739	20.0000000	1991244	42.8601	75.3184	0.9600
	TKM	BHN	5/17/1992 (138) 21:55:19.050	5/17/1992 (138) 21:57:35.950	2739	20.0000000	1991244	42.8601	75.3184	0.9600
	TKM	BHE	5/17/1992 (138) 21:55:19.050	5/17/1992 (138) 21:57:35.950	2739	20.0000000	1991244	42.8601	75.3184	0.9600
	KBK	BHZ	5/17/1992 (138) 21:55:14.400	5/17/1992 (138) 21:57:18.450	2482	20.0000000	1991244	42.6564	74.9478	1.7600
	KBK	BHN	5/17/1992 (138) 21:55:14.400	5/17/1992 (138) 21:57:18.450	2482	20.0000000	1991244	42.6564	74.9478	1.7600
	KBK	BHE	5/17/1992 (138) 21:55:14.400	5/17/1992 (138) 21:57:18.450	2482	20.0000000	1991244	42.6564	74.9478	1.7600
	AAK	BHZ	5/17/1992 (138) 21:55:10.250	5/17/1992 (138) 21:57:02.650	2249	20.0000000	1991244	42.6333	74.4944	1.6800
	AAK	BHN	5/17/1992 (138) 21:55:10.250	5/17/1992 (138) 21:57:02.650	2249	20.0000000	1991244	42.6333	74.4944	1.6800
	AAK	BHE	5/17/1992 (138) 21:55:10.250	5/17/1992 (138) 21:57:02.650	2249	20.0000000	1991244	42.6333	74.4944	1.6800

18

Arranging fields in a window

dbe chooses some order in which to display the fields of a view. This order may be inconvenient. To obtain a more useful layout, select the View->Arrange menu.

The Arrange option brings up a dialog window in which you may select the columns you wish to display, and the order in which they'll appear. Press the *none* button, then select the fields you want, and finally press *ok*.

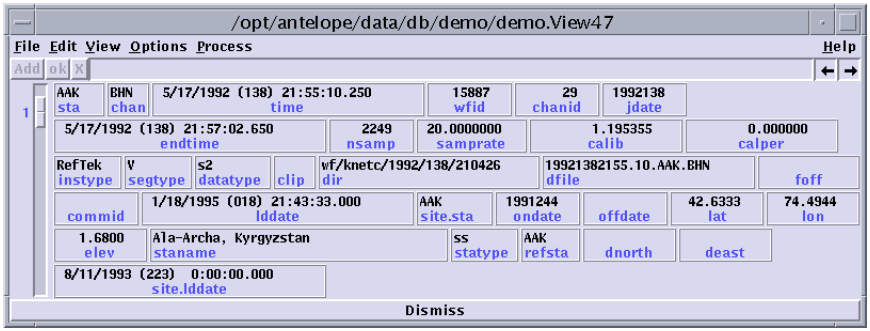


0	sta	chan	lat	lon	elev	staname
	CHM	BHZ	42.9986	74.7513	0.6550	Chumysh, Kyrgyzstan
	CHM	BHN	42.9986	74.7513	0.6550	Chumysh, Kyrgyzstan
	CHM	BHE	42.9986	74.7513	0.6550	Chumysh, Kyrgyzstan
	EKS2	BHZ	42.6615	73.7772	1.3600	Erkin-Sai, Kyrgyzstan
	EKS2	BHN	42.6615	73.7772	1.3600	Erkin-Sai, Kyrgyzstan
	EKS2	BHE	42.6615	73.7772	1.3600	Erkin-Sai, Kyrgyzstan
	USP	BHZ	43.2669	74.4997	0.7400	Uspenovka, Kyrgyzstan
	USP	BHN	43.2669	74.4997	0.7400	Uspenovka, Kyrgyzstan
	USP	BHE	43.2669	74.4997	0.7400	Uspenovka, Kyrgyzstan
	TKM	BHZ	42.8601	75.3184	0.9600	Tokmak, Kyrgyzstan
	TKM	BHN	42.8601	75.3184	0.9600	Tokmak, Kyrgyzstan
	TKM	BHE	42.8601	75.3184	0.9600	Tokmak, Kyrgyzstan
	KBK	BHZ	42.6564	74.9478	1.7600	Karagaibulak, Kyrgyzstan
	KBK	BHN	42.6564	74.9478	1.7600	Karagaibulak, Kyrgyzstan
	KBK	BHE	42.6564	74.9478	1.7600	Karagaibulak, Kyrgyzstan
	AAK	BHZ	42.6333	74.4944	1.6800	Ala-Archa, Kyrgyzstan
	AAK	BHN	42.6333	74.4944	1.6800	Ala-Archa, Kyrgyzstan
	AAK	BHE	42.6333	74.4944	1.6800	Ala-Archa, Kyrgyzstan

18

Viewing data in a record view

dbe normally presents data in a spreadsheet form, but sometimes it's difficult to see all the information on a single line. An alternative is to view the data one record at a time. The record view shows all the fields in the order in which they appear in the tables which make up the view. Click the right mouse button over the row which you want to see in a record view to bring up a new window. You can adjust the record either by clicking again on a different row, or by using the scrollbar on the left. Bring up multiple windows with shift-right-mouse.



Other database operations

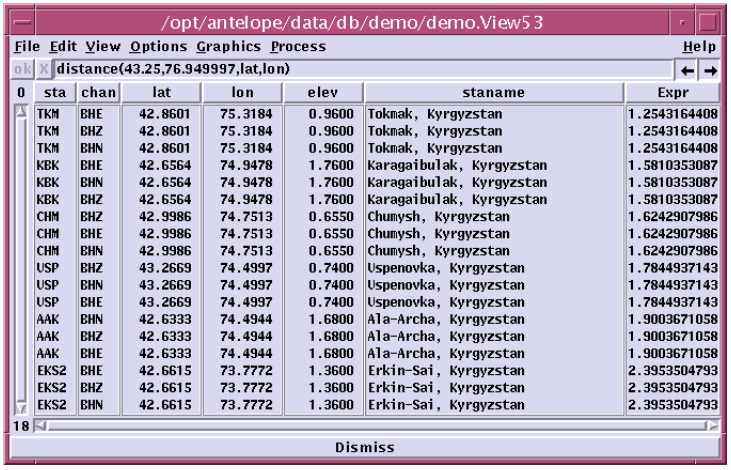
The join operation is probably the most difficult operation on a relational database. Other operations are simple in comparison. You can sort a table, using a list of fields or expressions. You can extract the subset of the records in a table which satisfy some conditions. You can combine these operations, performing a subset, then a join, then a sort, for example. We'll try some of these operations now.

Select *View->Sort* in the menubar of the joined table. This brings up a dialog window like the arrange dialog. Select some keys (maybe, *sta*, *chan*, *time*) for sorting, press *done*, and the table will be sorted, bringing up a new window. Notice the unique option, similar to the unix sort -u option. When you want to sort by only a single column, you can use the sort menu entry under the column as a short cut.

You can sort according to an expression as follows:

1. enter **`distance(43.25,76.949997,lat,lon)`**into the entry window.
2. select *add expression* under the staname column header.
3. a new column *Expr* should appear; select *Expr->sort* under this column.

These are the stations sorted by distance from Alma Ata:



0	sta	chan	lat	lon	elev	staname	Expr
	TKM	BHE	42.8601	75.3184	0.9600	Tokmak, Kyrgyzstan	1.2543164408
	TKM	BHZ	42.8601	75.3184	0.9600	Tokmak, Kyrgyzstan	1.2543164408
	TKM	BHN	42.8601	75.3184	0.9600	Tokmak, Kyrgyzstan	1.2543164408
	KBK	BHE	42.6564	74.9478	1.7600	Karagaibulak, Kyrgyzstan	1.5810353087
	KBK	BHN	42.6564	74.9478	1.7600	Karagaibulak, Kyrgyzstan	1.5810353087
	KBK	BHZ	42.6564	74.9478	1.7600	Karagaibulak, Kyrgyzstan	1.5810353087
	CHM	BHZ	42.9986	74.7513	0.6550	Chumysh, Kyrgyzstan	1.6242907986
	CHM	BHE	42.9986	74.7513	0.6550	Chumysh, Kyrgyzstan	1.6242907986
	CHM	BHN	42.9986	74.7513	0.6550	Chumysh, Kyrgyzstan	1.6242907986
	USP	BHZ	43.2669	74.4997	0.7400	Uspenovka, Kyrgyzstan	1.7844937143
	USP	BHN	43.2669	74.4997	0.7400	Uspenovka, Kyrgyzstan	1.7844937143
	USP	BHE	43.2669	74.4997	0.7400	Uspenovka, Kyrgyzstan	1.7844937143
	AAK	BHN	42.6333	74.4944	1.6800	Ala-Archa, Kyrgyzstan	1.9003671058
	AAK	BHZ	42.6333	74.4944	1.6800	Ala-Archa, Kyrgyzstan	1.9003671058
	AAK	BHE	42.6333	74.4944	1.6800	Ala-Archa, Kyrgyzstan	1.9003671058
	EKS2	BHE	42.6615	73.7772	1.3600	Erkin-Sai, Kyrgyzstan	2.3953504793
	EKS2	BHZ	42.6615	73.7772	1.3600	Erkin-Sai, Kyrgyzstan	2.3953504793
	EKS2	BHN	42.6615	73.7772	1.3600	Erkin-Sai, Kyrgyzstan	2.3953504793

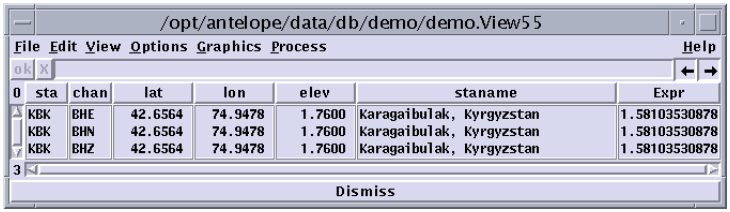
18

You can use the left scrollbar to scroll to a particular record. However, this may be inconvenient in a large table. As an alternative, try typing the station name (*USP*, for example) into the entry window, then click on one of the arrows to the right of th e entry window. This should move a matching record up to the top row of the display. You can alternatively type control-return or control-backspace, or use the find forward and find backwards menu options.

The simplest search just looks for a matching string in the entry record. However, you can enter a Datascope expression like *chan =~ /.*Z/*, or just a regular expression. A search with an empty expression advances one page.

Creating a subset view

Subset views are created by specifying a Datascope expression; only records which satisfy the expression are kept in the view. As a simple example, enter **`sta=="KBK"`** into the entry window, and then select *View->subset.*



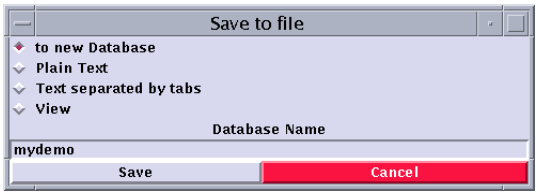
The original window disappears, and a new window with just the selected station appears. By default, dbe eliminates the old window after operations like *join*, *sort* and *subset*. This avoids cluttering the screen. However, you can keep the old window by selecting the *Options->keep window* menu.

For both searching and subsetting, you can look for records that satisfy more complex criteria -- like `time > “1992138 21:50” && chan == “BHZ”`. The syntax of Datascope expressions is similar to c and FORTRAN, and is covered in detail in a later chapter.

Using dbunjoin to create a subset database

There are a number of editing operations you can perform, but not on this demo database, which has been made read-only. Permissions are controlled strictly with standard UNIX permissions, so you can probably override this. Instead, let’s create a small local database that you can edit.

You already have a view of a subsetted join of wfdisc and site, and you have subsetted this table to contain only station KBK. Now join this table successively with sensor, sitechan, and instrument. These tables make up the core tables of the data side of the CSS database. The join you create references only rows which relate to the station KBK. Select *File->Save.*. on the menu. Select *to new database*, and enter **`mydemo`** as the name. Press the *Save* button.

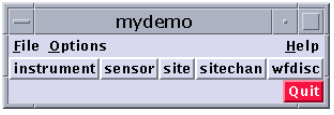


A new database is created in your current directory named `mydemo`. It has copies of each relevant row of the original database.

```
% ls
mydemo                mydemo.sensor       mydemo.sitechan
mydemo.instrument     mydemo.site         mydemo.wfdisc
```

Editing a database

You now have a copy of the database which you can edit. Open this database, either by running **dbe** against it, or by using the "File->Open Database.." menu.



Bring up a window on the site table by pressing the *site* button. This window should have just one record: there was just a single station in the view from which you created this database.

Before you can edit this table, you must select *Options->Allow edits* under the Options menu. After that, you can select a field by clicking in it, then edit that field in the entry area. When you are satisfied, click on the *ok*, or click on another field to edit. Scrolling will also save the edited value. For example, change the elevation from *1.760* to *1.670*.

You can change a whole column of values by entering an expression in the entry area, and using the *Set value* menu option under the column header. For instance, you could change all the dir fields in the wfdisc table from

wf/knetc/1992/138/210426 to plain *wf* by first bringing up the wfdisc window, then typing `wf` in the entry area, and choosing the *dir->Set value* menu option. Alternatively, you could get rid of the `138` directory in the path by putting *patsub(dir, "138/", "")* in the entry area, and choosing *dir->Set value*.

Note that these changes only change the table. The waveform files are actually still back in the original directory, and the wfdisc table is wrong. This operation (actually an unjoin, described later) does not adjust references to external files. You could correct this with a symbolic link, or by editing dir to make it */opt/antelope/data/db/demo*/wf/knetc/1992/138/210426

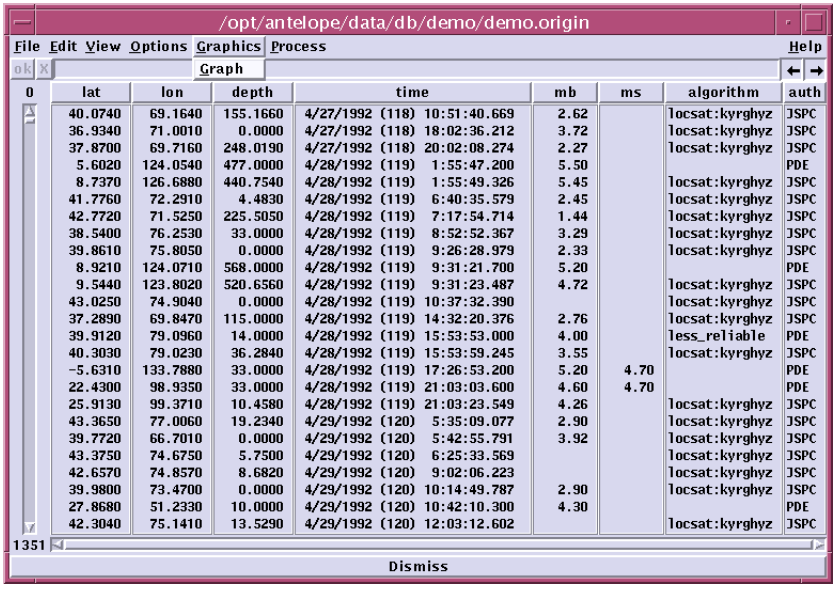
Try creating a new affiliation table, using the *File->Create New Table->affiliation* menu in the main dbe menubar. This brings up a dialog window into which you may type values, and then use add to add new records.

You can also delete rows by selecting a few rows with the mouse, and then using the *Edit->Delete* menu (this option will be disabled if you have not previously selected *Options->Allow edits*). For reasons which will become clear later, it's usually undesirable to physically remove the deleted records immediately. Instead, each field of these *deleted* records is set to the corresponding null value; a later *crunch* operation removes the null records.

Incidentally, multiple rows may be selected by dragging the mouse. Multiple selections are made by holding the shift key while clicking or dragging. However, moving or just clicking on the scrollbar clears all selections.

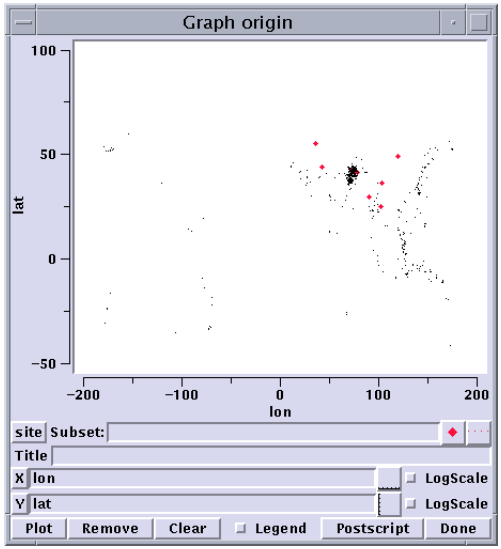
Simple graphing

dbe allows some simple graphing. Go back to the demo database, and bring up a window on the origin table. Select *Graphics->graph*:



lat	lon	depth	time	mb	ms	algorithm	auth
40.0740	69.1640	155.1660	4/27/1992 (118) 10:51:40.669	2.62		locsat:kyrghyz	JSPC
36.9340	71.0010	0.0000	4/27/1992 (118) 18:02:36.212	3.72		locsat:kyrghyz	JSPC
37.8700	69.7160	248.0190	4/27/1992 (118) 20:02:08.274	2.27		locsat:kyrghyz	JSPC
5.6020	124.0540	477.0000	4/28/1992 (119) 1:55:47.200	5.50			PDE
8.7370	126.6880	440.7540	4/28/1992 (119) 1:55:49.326	5.45		locsat:kyrghyz	JSPC
41.7760	72.2910	4.4830	4/28/1992 (119) 6:40:35.579	2.45		locsat:kyrghyz	JSPC
42.7720	71.5250	225.5050	4/28/1992 (119) 7:17:54.714	1.44		locsat:kyrghyz	JSPC
38.5400	76.2530	33.0000	4/28/1992 (119) 8:52:52.367	3.29		locsat:kyrghyz	JSPC
39.8610	75.8050	0.0000	4/28/1992 (119) 9:26:28.979	2.33		locsat:kyrghyz	JSPC
8.9210	124.0710	568.0000	4/28/1992 (119) 9:31:21.700	5.20			PDE
9.5440	123.8020	520.6560	4/28/1992 (119) 9:31:23.487	4.72		locsat:kyrghyz	JSPC
43.0250	74.9040	0.0000	4/28/1992 (119) 10:37:32.390			locsat:kyrghyz	JSPC
37.2890	69.8470	115.0000	4/28/1992 (119) 14:32:20.376	2.76		locsat:kyrghyz	JSPC
39.9120	79.0960	14.0000	4/28/1992 (119) 15:53:53.000	4.00		less_reliable	PDE
40.3030	79.0230	36.2840	4/28/1992 (119) 15:53:59.245	3.55		locsat:kyrghyz	JSPC
-5.6310	133.7880	33.0000	4/28/1992 (119) 17:26:53.200	5.20	4.70		PDE
22.4300	98.9350	33.0000	4/28/1992 (119) 21:03:03.600	4.60	4.70		PDE
25.9130	99.3710	10.4580	4/28/1992 (119) 21:03:23.549	4.26		locsat:kyrghyz	JSPC
43.3650	77.0060	19.2340	4/29/1992 (120) 5:35:09.077	2.90		locsat:kyrghyz	JSPC
39.7720	66.7010	0.0000	4/29/1992 (120) 5:42:55.791	3.92		locsat:kyrghyz	JSPC
43.3750	74.6750	5.7500	4/29/1992 (120) 6:25:33.569			locsat:kyrghyz	JSPC
42.6570	74.8570	8.6820	4/29/1992 (120) 9:02:06.223			locsat:kyrghyz	JSPC
39.9800	73.4700	0.0000	4/29/1992 (120) 10:14:49.787	2.90		locsat:kyrghyz	JSPC
27.8680	51.2330	10.0000	4/29/1992 (120) 10:42:10.300	4.30			PDE
42.3040	75.1410	13.5290	4/29/1992 (120) 12:03:12.602			locsat:kyrghyz	JSPC

This brings up an empty graph. Enter `lon` and `lat` in the x and y entry areas, either by typing or selecting from the menubutton label on the left of the entry area. Then press the “plot” button. Press the menubutton labeled “origin”, and select “site”. Use the button to the right of the Subset entry area which has a plot symbol in it to select a different plot symbol, color, and/or size. Press the “plot” button again. The result should look something like:



This graph shows all the origins (event locations or hypocenters) from the *origin* table as small black diamonds, and all the station locations as slightly larger red diamonds.

There are a variety of other ways to manipulate a graph; the best way to learn is to play with this. You can select a region of the graph by clicking the left mouse button twice to delineate the interesting region, which will then be magnified. You can do this multiple times; then clicking the right mouse will back out to the full view.

You can select subsets of the table by typing an expression in the Subset entry area, and you can change the scales to log scales. The plot can be saved as postscript, yielding a higher resolution than the screendump above.

Summary

This short tour of the demo database has introduced the dbe interface, and shown how to do simple *joins*, *subsets*, and *sorts*, as well as how to extract a small database from a large database. By simply playing with the various menus and buttons, you should now be able to form rather complex queries into the demo database. However, you will probably find it helpful to read the later chapters to learn more

about expressions and various database operations. dbe is probably the most useful single tool in the Datascope stable, but there are a variety of other tools for specialized use, and the primary value of Datascope comes in its use in programs.

CHAPTER 3

Schema and Data Representation

Datascope keeps tables as plain ASCII files. Each line is a separate record, and the fields occupy fixed positions within each line. (There is no variably sized text field.) The name of a file which represents a table is composed of two parts -- the database name and the table name, i.e. database.table. Typically, all the tables which make up a database are kept in a single directory. However, there is also provision to keep certain tables in a central location, but have multiple versions of other tables in other locations.

Database Descriptor Files

Datascope understands a descriptor file which specifies a few important parameters:

- the database schema name
- a path along which various tables of the database may be found.
- the table locking mechanism
- central id server

The schema name is used to look up a schema file. This file is typically kept in *$(ANTELOPE)/data/schemas*, but may instead be kept in the directory with the database descriptor -- this provides a means of testing alternative or modified schemas prior to installing them centrally.

The database path specifies a path along which to look for the files while hold the database tables. For any particular table, the first file matching the table name found along the path must contain the table.

The last two parameters are optional; they relate to table locking performed during the addition (*not deletion nor modification*) of records. The default is no locking. The other options are local filesystem locking or nfs filesystem locking. If you wish to share a database across multiple machines, you must use nfs locking.

In you use nfs locking, you must also set up and run an *idserver*, which ensures that each client gets unique integers for id fields in the database(s). This may be useful even when you are not using nfs locking, if you want to avoid duplicate ids among several databases.

Here's an example of a descriptor file

```
#
schema css3.0
dblocks nfs
dbidserverxx.host.com
dbpath /opt/antelope/data/db/demo/{demo}
```

The example above is the current preferred format, but Datascope still supports an earlier version which did not contain either dblocks or dbidserver. The order is important for this descriptor file: schema on the first line, dbpath on the second:

```
css3.0
/opt/antelope/data/db/demo/{demo}
```

One can specify the idserver and the locking mechanism with environment variables (DBLOCKS and DBIDSERVER), but this requires all databases to use the same locking and id server.

Representation of Fields

While the field values are represented in ASCII format in the disk files, Datascope converts them to three different binary formats for use in programs: double precision floating point, integer, and string. The calculator recognizes a few other types -- boolean, time and yearday -- and converts between them as necessary. (Time is

represented as a double precision floating point, and yearday is represented as an integer.)

There is actually one additional field type which Datascope uses internally -- a database pointer type. This type contains a reference to a single row or ranges of rows in another table. This type is the basis for views and grouping of tables.

Schema Description File

The structure of the individual files and the database overall is dictated by a *schema* file. This file describes the fields of the database, and specifies how these fields are used in each table. Datascope's schema file is unique in several respects:

- The schema file is a text file which is read and interpreted whenever a database is opened. Changes in the schema file are reflected in the next execution of a program which uses Datascope.
- A field with the same name has the same attributes (size, type, format) in every table in which it appears. In other DBMSs (DataBase Management Systems), the same name might apply to entirely different kinds of fields.
- There is considerably more information associated with every field and table than in most DBMS'. This additional information serves to document a database, and also allows Datascope to provide some more sophisticated operations like joins and some automated verification tests.

A schema file contains three types of statements: *Schema, Attribute,* and *Relation.*

Schema Statement

The Schema statement appears only once, at the beginning -- it provides a short and a long description of the schema overall. It is not required. The format of the statement is:

Schema *name*
Description ("*short description*")
Detail {
long description
}
Timedate *time-stamp-field*

;

You may also specify a field containing a time which is modified automatically whenever a record is changed. For the CSS schema, this field is *lddate*.

Attribute Statement

The Attribute statement describes a single field of the database. It specifies the size and type of each field, a a (C printf style) format code, a range of legal values, a null value, the units (if applicable), and a short and a long description of the field.

Attribute *name*

type (*length*)

Format ("*format*")

Range ("*expression*")

Null ("*null value*")

Units ("*physical units*")

Description ("*brief description*")

Detail {

Detailed description

}

;

Names should be alphanumeric, beginning with a letter. The legal types are *Real*, *Integer*, *String*, *Time*, *Yearday*, and *Dbptr*. The *length* specification is the number of characters to allow for the printed representation of the field. The **Format** code is a (C) printf style format code that specifies how to translate from the internal, binary representation (integer, double or string) to the printed format.

The *Null* value varies from field to field, but represents a field for which information is not available. It is not the same as the SQL NULL. It is usually a value outside the *Range*.

The *Range* should be a boolean expression which is true for valid values of the field.

The *Units* specification is not currently used anywhere, but should specify the physical units of the field in cases where this has some meaning.

The brief and detailed descriptions provide a convenient way of documenting the schema, and are available for help screens.

Only the name, type, length, and format are required; however, filling in all the clauses which make sense provides fairly extensive documentation.

Relation Statement

The Relation statement describes a table of the database. It has the following format:

Relation *name*
Fields (*field field ...*)
Primary (*key key ...*)
Alternate (*key*)
Foreign (*field field ...*)
Defines *field*
Separator ("*c*" "*c*")
Transient
Description ("*brief description*")
Detail {
Detailed description
}
;

The **Fields** clause lists the fields which make up a record of the table.

Datascope allows specifying two keys for a database table, a primary and an alternate key. The alternate key is often a single id field. A key *should* identify a unique record in the table; it is a mistake if one key matches more than one record in the table. **Datascope** does not prevent this situation, but **dbverify** will flag the problem.

A table may have fields which are basically indexes into other tables. Such fields may be identified with the **Foreign** clause. **Foreign** keys must be single fields, not a compound key.

Usually, a foreign key is an *id* field -- a small integer which identifies a row in a table, but has no intrinsic meaning. Some examples from the CSS database are *wfid*, *arid* and *inid*. These integers may be assigned in any arbitrary fashion, provided they are all unique. **Datascope** has provision for automatically generating these ids (see **dbnextid(3)**), but they must be identified in the schema by the **Defines** clause.

By default, **Datascope** separates the fields in a database record with spaces, and separate records with a linefeed. This is convenient for editing with a text editor (although tab would be a more convenient field separator for processing by **awk**). These defaults may be overridden by specifying field and record separators; specifying null strings will eliminate the separators altogether.

The **Description** and **Detail** clauses serve the same function as in the **Schema** and **Attribute** statements, providing brief and more detailed explanations of the field.

The **Transient** clause is described below; it is not typically used in a schema file.

Datascope Views

The schema file is usually kept in a central location, read and compiled whenever a database is opened. It should specify all the central tables of the database. However, it is possible to create additional tables on the fly. Such tables are *Transient*, have no direct identification in the schema file, and usually are not represented on disk directly.

The most common and useful variety of such tables are *simple views*. Simple views should be regarded (and are implemented) as arrays of database pointers. Each database pointer in a simple view identifies a single record of a base table. One dimensional arrays (vectors) are useful as sorted lists and subsets of the records of a single table. Two dimensional arrays represent joins of several tables; such joins may also be sorted or represent subsets of the complete join. For instance, a view of the *site* table (sorted and/or subsetted) could be described in the schema as

```
Relation site_view
        Fields ( site )
        Primary ( sta ondate::offdate )
        Transient
        Description ( “Example of simple vector view” )
        Detail {
```

```
        You create a table like this when you
        sort or subset the site table in dbe.
        }
    ;
```

A simple joined view might represent a join of the *wfdisc*, *sensor*, *site*, *sitechan*, and *instrument* tables:

```
Relation data_view
    Fields ( wfdisc sensor site sitechan instrument )
    Transient
    Description ( "Example of a simple joined view" )
    Detail {
        You can create a table like this in dbe
        by joining wfdisc to sensor, the result to
        site, that result to sitechan, and finally
        joining that result to instrument.
        }
```

While a simple view consists only of database pointers, more complex views which mix database pointers and other types of fields are also possible.

An example of a complex view is a grouped view. This view will have a set of fields which are represented directly in the table, and a special database pointer which refers to a range of rows in another table. This other table may be a base table, but is more often itself a simple, sorted view. The database pointer which refers to a range is always named *bundle*; there is currently no provision for keeping more than one such pointer in any table.

Reserved Names for Fields and Tables

The names of certain fields and tables bear a special meaning to **Datascope**. This is probably a mistake, but the cost to specify these fields into the schema rather than building them directly into Datascope was judged to be higher than the added value.

The particular choice of names usually relates to the origin of Datascope in supporting the CSS schema. A few names support misfeatures of the CSS schema, and

should just be avoided in new schemas; *commid* and *lineno* from the *remark* table, and *ondate* and *offdate* in the *site* and *sitechan* tables are examples.

Other names are arbitrary choices which serve to implement necessary features. A redesign might choose different names, but the functionality would be essentially the same; *dir*, *dfile*, *lastid*, *keyname* and *keyvalue* are examples.

The CSS database provides a separate *remark* table for adding comments; many tables then refer to a set of records in the *remark* table with the *commid* field. Each record in the *remark* table allows 80 bytes for a comment; however, longer comments can be entered by using multiple records with the same *commid* and different *lineno*. In a few places, Datascope accomodates this scheme explicitly. Routines are provided to add or extract comments. See dbremark(3) and dbremark(1).

The *site* and *sitechan* tables specify *ondate* and *offdate*, so-called "julian" days, for the time range, rather than just *time* and *endtime* (epoch times). This makes it impossible to specify changes in instrument orientation during a single day, and complicates the join between other tables like *wfdisc* and *sensor*, which specify a time range in epoch time. The different names must be recognized, and conversions must be done from yearday format to epoch time format. To deal with this, Datascope explicitly recognizes *ondate* and *offdate*, and explicitly handles cases where tables with *time* are joined to tables with *ondate* and *offdate*. The further special case of a null *offdate* indicating the indefinite future is handled explicitly. However, you would be wise to avoid using *ondate* and *offdate* in any new tables or schemas.

Dir and *dfile* specify a pathname to a file outside the tables. Such files could be regarded as another field of a table, but a field which the database is not capable of manipulating directly. In the CSS schema, *dir* and *dfile* are used to refer to recorded data, and to instrument response descriptions.

Waveform data is kept out of the database because of its volume. The parameter data in the database is a very small fraction of the size of the collected data. It's quite useful to have this parameter data online continuously and quite impossible (for most users at least) to keep the collected data online all the time.

The instrument response information is an example of information which is not best represented in a relational database form. While it would be possible to keep this information directly in the database, doing so would confer no additional advantages, and would have some direct costs in speed and convenience.

In order to assign unique values to id fields when new records are added, Datascope uses the *lastid* table. *Keyname* is the name of id, while *keyvalue* is the last assigned integer for that id. Datascope increments the latter when a new record is added to the table which defines the id. Other schemes might be devised to handle this problem, but this is adequate.

A word of caution regarding id fields

Because id fields present a fast and simple key to a table, there is a tendency to make lots of them, provide an id key for every table, and do the joins on these ids. This is usually a mistake. If possible, avoid ids and make your keys the combination of meaningful fields which uniquely specify a record in a table.

While ids are simple and seductively attractive, they introduce some of the knottiest problems in database management, whether you are using Datascope or any other relational database management system. Because they have no meaning outside the database context, if they are ever modified inappropriately, it may be difficult or impossible to recover. Id fields also complicate operations like merging or comparing two databases, sometimes to the point of making the operation impossible. Ids are especially bad in tables where the real key is a time range of some sort; *wfdisc* and *sitechan* are good examples in the CSS database. In either of these tables, any particular record could be split into two records covering adjacent time ranges. (This might be done to reflect actual changes in station parameters, or in the case of *wfdisc*, just to segment the recorded data differently.) Doing so would not affect the database integrity if all joins were made on the true keys of these tables. However, joins which use the ids in these tables (*wfid* and *chanid*) would no longer be correct, and fixing up the problem could be difficult.

Finally, *bundle* and *bundletype* are newly introduced fields which support grouping. *Bundle* is the name given to a database pointer in a complex view which refers to a range of rows in some other table. *Bundletype* is an integer which may be used to specify the level of the grouping -- that is, a table grouped by certain fields might be further grouped by a subset of those fields.

CHAPTER 4

Basic Datascope Operations

Datascope provides all the standard operations which any RDBMS must, albeit in a somewhat different fashion than the standard SQL approach. In addition to the simplest operations of reading, writing, adding and deleting records, it's possible to subset, sort, group, and join tables. You probably have an intuitive understanding of the subset, sort, and group operations, and the underlying code is conceptually simple. Joins are a bit more complex, and this chapter concentrates on explaining how Datascope handles joins.

Reading and Writing Fields and Records

Datascope, of course, provides ways of doing this, translating from the ASCII representation of the files to a binary representation more convenient for programming. Files which represent tables are mapped into memory and accessed as large arrays. This means that the tables do not use up swap space, and it tends to be faster than going through the i/o interface.

Deleting Records

Unlike most RDBMS, specific record numbers are used when reading or writing records: e.g., record #10. These record numbers are directly manipulated by the user. Because of this, deleting records can cause problems; deleting a record

changes the record number of all subsequent records. This is often inconvenient, and consequently records may instead be *marked* for later deletion. Marking a record sets all of the fields to null values; a later *crunch* operation deletes all records which are null.

Subsets

The subset operation is pretty intuitive -- find all records which satisfy some condition, a boolean expression. This corresponds to the Where clause in SQL. However, Datascope expressions are more flexible than most SQL implementations.

Sorts

The sort operation simply orders the records according to some set of sort keys.

Grouping

A sorted table may be grouped -- this just means that consecutive records which are identical in some set of fields are separated into bins, and a few additional operations can be performed on these bins.

Joining Tables

The principal drawback of keeping data in a relational database is that you must regularly *join* tables in order to extract information from the database. In the *demo* database, given data recorded at station AAK and referenced in the *wfdisc* table, you must look into the *site* table to find out the location of AAK, and into *sitechan* to find out the orientation of the sensors, and into *sensor* and *instrument* to find out the response of the instrument. The most convenient way to do this is to construct a join of *wfdisc*, *sensor*, *instrument*, *site* and *sitechan*.

Conceptually, you can imagine forming a join of two (or more) tables as follows:

1. Find all combinations of a row from each table.
2. Keep only the combinations which satisfy some condition.

This approach would involve calculating the expression m1 * m2 * ... * mn times. For complex expressions, the join must in fact be performed in this fashion. Datas-

cope's theta join is performed in exactly this manner. However, this can be slow and expensive (try it), and it's often possible to use indexes to work much faster.

Natural joins among the tables of a database are those where the join condition is that fields with the same name have the same value. For instance, when joining *wfdisc* with *site*, the *sta* field should be the same in both the *wfdisc* row and the *site* row:

```
wfdisc.sta == site.sta
```

Forming the joined table which satisfies this condition can be done in a more efficient fashion than by considering every possible combination of rows from *wfdisc* and *site*. Instead, first sort *site* by *sta*, then proceed linearly through the wfdisc table. For each row of *wfdisc*, find the rows in site which have the same *sta* by searching the sorted list. Instead of m1 * m2 calculations, there are now more like m1 * log(m2) calculations -- much quicker (you may need to consider the time to compute the sorted list, of course). Datascope performs natural joins in exactly this fashion.

In addition, of course, the time ranges in *wfdisc* and *site* should match. In Datascope, this is taken to mean that the time ranges overlap. (In practice, this often means that one time range falls completely within the other; it is sometimes a database inconsistency if this is not true.)

Datascope provides a method of indexing time ranges. This is more difficult than it may at first appear. Consider a number of overlapping time ranges -- how should they be sorted? There is no unique ordering for overlapping time ranges. They might be sorted by starting time, by ending time, by the midpoint, etcetera. Some choice must be made, and Datascope chooses the one you probably thought of first; it sorts by starting time. However, if a time range index consisted of just starting times, on average the join expression would need to be computed on half of the indexed table for each record of the first table. To avoid this, Datascope first sorts by starting time, then computes a monotonically increasing envelope function of end times. The start times and this envelope function have the same sort order, so that the set of candidate matches is considerably restricted. Nonetheless, range indexes are more costly to compute and to join with than indexes on simple or compound keys.

Inferring Join Keys

If you don't specify the keys by which to join tables, an attempt is made to infer the keys based on the primary, alternate, and foreign keys specified for the two tables. The process is based on matching names for the keys of the two tables. It proceeds as follows:

1. If any foreign key in the first table is a part of the primary or alternate key in the second table, use it as the join key.
2. If any foreign key in the second table is a part of the primary or alternate key in the first table, use it as the join key.
3. Compare the primary keys of the two tables; if any fields match, use those as the join keys.
4. Compare the alternate keys of the two tables; if any fields match, use those as the join keys.
5. Compare the primary keys of the first table to the alternate keys of the second table; if any fields match, use those as the join keys.
6. Compare the primary keys of the second table to the alternate keys of the first table; if any fields match, use those as the join keys.

The process is complicated somewhat by the presence of ranges, for example a time range. A range matches another range with the same start and end names. (As a special concession to the CSS 3.0 database, time::endtime will also match ondate::offdate. The original decision to have the fields ondate and offdate was flawed.) A range will also match a single field which has the same name as the start of the range. For instance, time will match time::endtime.

Tables which are the result of joins inherit a subset of the keys possessed by their parents. This process is strictly ad hoc, and is *not* guaranteed to produce the correct subsequent joins. However, it is very often correct, and by careful choice of the order in which you join tables, you can usually force the correct join.

Inheritance of keys

The inheritance of keys in joined tables is based strictly on the names of the keys. The primary key for the joined table is the set of unique names from the primary keys of the two parent tables. The alternate and foreign keys are similarly constituted.

Once again, the presence of ranges complicates matters. If the parent tables had time and time::endtime as part of their primary key, the joined table will have both time and time::endtime as part of its primary key. However, because both *time*'s are unqualified, the time from the first table in the join will be used in later joins. This will probably not be the right choice.

Specifying Join Keys

To be absolutely certain you have the correct join keys, or to override the inferred join, you may specify join keys. Join keys are specified as two lists of fields or ranges; one list for each table. Often these two lists are identical. Ranges are indicated by the double colon separating the lower and upper boundary of the range. Note that the keys need not be just simple field names; they may also be expressions.

Speed and efficiency

Datascope always performs the joins two tables at a time, in the order you specify. An index is always used for the second table, while the first table is stepped through linearly. This can be useful for verifying that every record in the first table joins with one or more records in the second table, or identifying the records which don't join. (Finding the records which join with multiple records in the second table is more difficult).

However, your usual concern is what's fastest? A commercial DBMS will probably attempt to figure this out for you, and compute the join in the most efficient fashion. In Datascope, you must figure it out on your own. If it's really important, time the command line utilities. But as a rule of thumb, joins will be fastest if the second table has more rows and has the necessary index precomputed. If an index must be computed to perform the join, then it's faster (usually) if the second table has fewer rows.

An operation which is complementary to a join is to find all the records which do not join with a second table. Such records often indicate a database problem of some sort. The join operation (from the C interface) returns a list of these records, and the dbnojoin command and tcl and perl operation provide a view containing these records.

Once you have a joined view, you may want to take it apart. Datascope provides a few operations which do this. dbsever removes one table from a view, keeping only

rows which are unique in the output view. A related operation is dbseparate, which forms a view of all the records from a particular table which participate in the input view. A view may also be taken apart with dbunjoin, which creates new tables containing only the records which participate in a view. A corresponding operation is to return a list of all the records participating in the view for each table in the view.

A view is a somewhat static snapshot into a database It is a set of references to specific records in the database. Changes in the fields of existing rows of the database will be reflected in the view, if you get the value of a field. However, the view will not change as new records are added, nor because changing fields would result in a different view if the view were recomputed. A program which needs a more dynamic window into the database must either recompute the view frequently, or may find the dbmatches operation to be useful. dbmatches finds the collection of records in a table which would satisfy a certain join condition; it does not create a new joined view, it just returns the list of records. The user must index through those records; this is more lightweight and more convenient in many situations.

Summary

Datascope database operations are the guts of the database system; it's largely these operations which provide the power of the database. However, like arithmetic, there are not that many operations to learn: subset, sort, join, group and a few other closely related operations. All these operations are probably familiar from set theory in junior high school, or in recent years, first grade.

CHAPTER 5

Expression Calculator

Datascope has a very useful expression calculator. Expressions are used in many ways:

- to create a subset of a table
- as a sort key or a join key
- to calculate a value which doesn't appear directly in a table
- in the schema, to specify the legal range of a field.

Expressions have a familiar algebraic format, and are quite similar to expressions in c, FORTRAN and awk. You may find it instructive to play with expressions as you read this section, using the program dbcalc. dbcalc simply reads expressions from standard input, and evaluates them, printing the result. You need to give it a database table to use. Try

```
% dbcalc demo.wfdisc
```

And you can follow along, typing in the expressions below.

Basic Operators and Database Fields

Of course, the calculator understands the basic operators: '+', '-', '*' and '/'. Database fields are treated like named variables, and may be intermixed with literals. For example:

```
time
5/21/1987  5:04:37.000
(nsamp-1)/samprate + time
5/21/1987  5:05:28.996
endtime
5/21/1987  5:05:28.996
```

The specific results depend on the particular record selected, of course. For clarity, the precision of the printed values has been limited; floating point numbers are all internally double precision.

In addition to the familiar operators, the calculator understands '%' for modulus and '^' for exponentiation, and a few bitwise operators are understood:

```
10^5
   100000
1.3^2.1
   1.73
15 % 4
   3
25.3 % .25
   0.05
1 << 5
   32
28 >> 3
   3
25 & 3
   1
10 | 3
11
```

Data Types

The calculator understands a few basic data types, and converts between them as necessary. Thus a string may be converted to an integer, or an integer to a floating point number, or a number to a string, depending on the context:

```
10 + '13'
   23
10 + '13.5'
   23
10.0 + '13.5'
   23.5
10 + 13.5
   23.5
```

Each field in a database table has a specified type; the type of input literals is dictated by their format. A number without a decimal point is an integer, a number with a decimal point is floating point. Strings are enclosed in single or double quotes. The result of `10 + '13.5'` may have surprised you; because the first operand was an integer, the string was converted to an integer, not a floating point.

Some conversions can be done explicitly:

```
ceil(-3.5)
   -3
floor(-3.5)
   -4
int(-3.5)
   -3
```

String Operations

Strings have a few special operations; "." is the concatenation operator:

```
"a" . 'b'
   "ab"
sta . ':' . chan
   "KKL:HBIE"
```

```
dir . '/' . dfile
   "sac/1987141050437.00.KKL.HBIE"
```

The extfile function returns the path to an external file specified by the special fields *dir* and *dfile* in the record.

```
extfile()
    "ex/sac/1987141050437.00.KKL.HBIE"
extfile("wfdisc")
    "ex/sac/1987141050437.00.KKL.HBIE"
```

Notice that this is not necessarily the same as the concatenation of *dir*, '/', and *dfile*. If the path is not an absolute path, it is relative to the location of the file which contains the table.

The optional argument to the extfile function is the name of a table. In some cases, a joined view may contain multiple *dir* and *dfile* fields. (Consider a join of *wfdisc*, *sensor* and *instrument*, for example.) In this case, specifying a particular base table (*instrument*, for example) ensures that you get the external file specified in that table.

Pattern matching and pattern substitution are available:

```
chan
   "HBIE"
chan =~ /.*E/
   true
chan =~ /.*Z/
   false
chan =~ /E/
   false
sta =~ /AAK|CHM|KKL/
   true
time
   5/21/1987  5:04:37.000
time =~ /.* 5:0.:.*/
   true
patsub ( chan, "HB", "BH" )
```

```
   "BHIE"
patsub ( sta . "/" . chan, '([A-Z]+)/([A-Z]+)', '$2/$1')
   "HBIE/KKL"
```

The pattern matching expression is a standard UNIX regular expression. Read the sed man page for a complete description. The pattern must *match* the entire string on the left, not simply be a part of it. However, strings from fields are stripped of leading and trailing blanks before being used in calculations.

The pattern substitution pattern (second parameter of the *patsub* function) need only match part of the string (first parameter). As with sed and perl, matching sub-expressions (delimited by parentheses as shown above) can be referenced in the replacement string as $1 through $9.

Logical Operators

There are the typical (C form of) logical operators:

```
chan == 'HBIE'
   true
chan > 'HB'
   true
calib != NULL
   true
commid == NULL
   true
```

In simple expressions with a field on the left, followed by the operator '==' or '!=', NULL is understood as the null value for that field.

When a time field is compared to a string literal, the string may specify time in a variety of recognizable formats. Most common formats will be recognized properly. (See the man pages epoch(3) or epoch(1) for more examples).

```
time
   5/21/1987  5:04:37.000
time > "5:04:35 May 21 1987"
   true
```

```
time > “5/21/87" && time < “5/22/87"
   true
time < “5/21/87” || time > “5/22/87”
   false
time > “5:04 pm 5/21/87"
   false
time > “1987141"
   true
time > “1987 (141)”
   true
```

Another operator borrowed from c is the ’?’, which evaluates an expression and returns one value if the expression is true and a second value if the expression is false:

```
calib
    1.2
calib ? calib : 1.0
    1.2
chan
    “HBIE”
chan == “HBIZ” ? “vertical” : “horizontal”
    “horizontal”
```

While this syntax can be confusing, it takes the part of if .. else in the calculator. You can read

```
e1 ? e2 : e3
```

as

```
if ( e1 )
  e2
else
  e3
```

Assignments

Sometimes expressions can change the value of a field. The assignment operator is *:=* to make it difficult to mistakenly make an assignment. NULL is also understood as the null value for the field in this context.

```
commid := NULL
   -1
endtime := time + (nsamp-1)/samprate
         5/21/1987   5:05:28.996
```

As in C, the value of an assignment is the value assigned. Operators may be mixed in arbitrarily complex expressions. The precedence of operators is the same as C, and parentheses may be used to make the order of evaluation explicit.

Standard Math Functions

There are a number of math functions. The usual trigonometric functions:

```
sin(30)
   0.5
cos(45)
   0.707
tan(90)
   -2.61e+15
tan(0)
   0
tan(45)
   1
atan(sqrt(1-.5^2), .5)
   60
```

For convenience with the CSS database, arguments and results are in degrees, rather than radians.

There are also the familiar natural and common logarithms, square root, absolute value and min and max. Sign returns one, zero, or negative one, depending on the sign of the operand.

```
log(10)
   2.3
log10(10)
   1
exp(log(19.5))
   19.5
sqrt(2)/2.0
   0.707
abs(-5.3)
   5.3
min(-5, -3)
   -5
max(12, 30)
   30
sign(-5.3)
   -1
sign(0)
   0
sign(1.5e-30)
   1
```

Time Conversion

There are a variety of functions to convert among several common time formats. epoch converts a year/day of year integer (for example, day 35 of year 1973 is 1973035) -- sometimes termed a julian day-- to an epoch time: seconds since January 1, 1970. yearday takes an epoch time and converts back to the yearday format. strdate and strtime provide a standard conversion from epoch time to the more easily understood date and time format. date2e takes a year, month, day, and seconds of that day, and converts to an epoch time. The str2epoch function converts a string to an epoch time. now returns the current epoch time (as reported by the local system).

```
epoch ( 1987244 )
   9/01/1987  0:00:00.000
yearday ( “9/1/87" )
```

```
   1987141
strdate ( time )
   " 5/21/1987"
strtime ( time )
   " 5/21/1987  5:04:37.000"
date2e ( 1987, 5, 12, 2*3600+34*60+5.35 )
   5/12/1987  2:34:05.350
str2epoch("Jan 25, 1993 11:23:5.4")
   1/25/1993 11:23:05.400
now ()
   10/15/1994 23:04:46.000
```

Spherical Geometry

There are several functions related to distances around a spherical earth. Distances are expressed in angular degrees; azimuth is measured clockwise from geocentric north.

distance gives the angular distance between two points, while azimuth computes the direction from one point to the second. latitude and longitude compute the inverse operation -- the resulting latitude and longitude when moving a certain number of degrees in a particular direction.

```
distance(origin.lat, origin.lon, site.lat, site.lon )
    13.7234
azimuth(origin.lat, origin.lon, site.lat, site.lon )
    306.345
latitude(origin.lat, origin.lon, 13.7234, 306.345)
    56.4001
longitude(origin.lat, origin.lon, 13.7234, 306.345)
    58.6001
site.lat
    56.4
site.lon
    58.6
```

(The computed site.lat differs from the actual site.lat because of roundoff error in the distance and azimuth).

Seismic Travel Times

A seismic travel time calculator is built in. ptime and stime calculate the travel time for the first p and s arrival. (The arguments are distance in degrees, followed by depth in kilometers; for phase_arrival, the argument is velocity in degrees/second).

```
ptime(5.105, 155.166)
   74.8333
stime(5.105, 155.166)
   133.79
ptime(5.105, 155.166)+origin.time
   4/27/1992 10:52:55.502
stime(5.105, 155.166)+origin.time
   4/27/1992 10:53:54.459
parrival()
   4/27/1992 10:52:55.502
arrival("Pn")
   4/27/1992 10:52:55.502
sarrival()
   4/27/1992 10:53:54.460
arrival("Sn")
   4/27/1992 10:53:54.460
arrival("PcP")
   4/27/1992 10:59:52.549
phase_arrival(.6)
   4/27/1992 10:51:49.177
```

ptime and stime may always be evaluated. However, parrival, sarrival, and phase_arrival require a join of *origin* and *site*, because they use fields from both tables. parrival is shorthand for:

```
origin.time+ptime(distance(origin.lat,orgin.lon,site.lat,
site.lon), origin.depth)
```

and sarrival is shorthand for:

```
origin.time+stime(distance(origin.lat,origin.lon,site.lat,
site.lon), origin.depth)
```

and arrival is shorthand for:

```
origin.time+phasetime(phase,distance(origin.lat,origin.lon,
site.lat, site.lon), origin.depth)
```

phase_arrival(velocity) is shorthand for:

```
origin.time + distance(origin.lat, origin.lon, site.lat,
site.lon)/velocity
```

Note that the specified velocity is in degrees/second, not kilometers per second as you might have expected.

Seismic and Geographic Region functions

The Flinn-Engdahl seismic and geographic region functions are built-in. grn and srn take two arguments -- latitude and longitude in degrees -- and return the corresponding geographic or seismic region number, respectively. gregion and sregion take the same arguments, but return the names of these regions.

```
sta
   "HIA"
lat
   49.2667
lon
   119.742
grn(lat,lon)
   657
gregion(lat,lon)
   "E. USSR-N.E. CHINA BORDER REG."
srn(lat,lon)
   41
sregion(lat,lon)
   "EASTERN ASIA"
```

Conglomerate functions

A few functions operate on an entire table or a group.

```
max(lat)
    43.2669
min(lat)
    42.0778
count()
    20
sum(lat)/count()
    42.6158
```

min and max (with a single argument) find the minimum or maximum value of the expression over the entire range of the table or group. count returns the number of records, and sum computes the sum of an expression, calculated over all the rows of a table or group.

Whether to compute over an entire table or over a group is decided based on the presence of the field *bundle*, which is normally present only in a grouped view. For these grouped views, the computation is done over just a range of records, in the subsidiary table. You can force the calculation to range over the entire table instead by using min_table, max_table, count_table, or sum_table.

External functions

It's possible to execute a shell command and return a value. The syntax for this is borrowed from tcl/tk:

```
["date"]
    "Sat Oct 15 20:29:21 MDT 1994"
["wc" "-c" extfile()]
   "  88368 /jspc/wf/prod-
ucts/casia_waveforms_may92/wf/knetc/1992/138/210426/19921
382148.07.CHM.BHZ"
```

Note that the string literal arguments must be quoted; the list of expressions which make up the command line are evaluated just like any other expression.

CHAPTER 6

Programming with Datascope

While a great deal can be accomplished with just dbe, ultimately you will probably need to write specialized scripts and programs. You might want to generate reports offline, for instance. In addition, there are many kinds of problems which can't be solved by just manipulating sets. Finally, Datascope doesn't address the issue of external files (in seismology, the actual waveform data) beyond providing a way of indexing them.

You can attack programming problems from a variety of levels with Datascope. Some problems, (like generating a report) can easily be solved with Datascope's command line utilities. These provide the basic operations (subsets, joins, and sorts) at the command line. If you can get a table display from dbe which is what you want, then you can also write a shell script to generate the same information.

When the problem is almost solved by the command line utilities, you can often use standard shell tools like awk, sed or perl to finish the job.

Problems involving conversions to or from another RDBMS might be handled most easily with the perl interface, because perl is quite good at text manipulation, and also has interfaces to many of the standard RDBMS.

The command line utilities are also convenient for generating the input data set for more complex analysis. Rather than build a selection process into a program, use the command line utilities to select an interesting subset of data from the database,

read the resulting view directly in the program, and concentrate in the program on solving the problem. This approach is likely to lead to more general purpose programs.

Especially when a GUI interface is desired, consider tcl/tk. Antelope provides tcl/tk extensions which provide direct access to most of the functionality of Datascope and many other Antelope functions. The tcl interface is probably the easiest to learn, after the command line interface. Because tcl may be run interactively, you can (and should) develop an interactive development style: type in a few commands at a time inside atcl or awish, before saving them into a script file. The process of creating an application under X Windows is dramatically simpler with tk than with c. dbe is itself a tcl/tk script, as are several other Antelope programs.

There is also a complete perl interface into Datascope and many other Antelope functions. Perl is a jack-of-all-trades language, particularly prized by system administrators. It's fast, easy to write, and widely available. Many people prefer it to tcl/tk and c.

The Datascope library is written in common c, with a few yacc and lex parsers. Consequently, the interface has a distinctly c flavor, and the c interface is the best supported and tested. For problems which cannot be solved by simpler means, the c interface is preferred.

In some primarily numerical applications, (or because of familiarity with the language), FORTRAN may be the preferred language. FORTRAN's limitations with respect to dynamic memory allocation and string manipulation make it difficult to support all of the Datascope functionality. However, most of the c interface to Datascope has a FORTRAN counterpart of the same name, sometimes with slightly different parameters.

If you have a numerically complex problem, or you want extensive graphics support, and have access to it, you may consider Matlab. Kent Lindquist has a contributed matlab package which provides access to many Antelope functions including Datascope.

Sample Problem

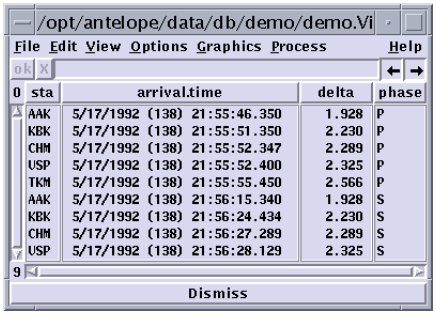
At this point, you should already be familiar with Datascope from exploration with dbe. This chapter introduces the programming interface by considering a simple

example problem. It will not go into detail about the usage of any particular Datascope call; for the complete story, please look up the man page.

The problem to be solved is to find all the arrivals associated with an event, ordered by arrival time, and for each arrival, to show the station, arrival time, phase, and distance to the event. This is easily done from within dbe in the *demo* database:

1. bring up the *origin* table.
2. select one origin, by entering its origin id (*orid*) in the entry area and selecting subset from the menu under the orid column header. (Use *orid* 645)
3. join this subset to the *assoc* table, then to the *arrival* table, and finally to the *site* table.
4. sort by *arrival.time*.
5. Use the arrange dialog to show *sta, arrival.time, .delta, phase*.

You should get a display like the following:



This problem is very simple, but serves to introduce various approaches to the problem. (As an exercise, and to get a better appreciation of what you gain by using Datascope, you might try solving this problem working directly against the text files using c, FORTRAN, and/or perl. Or create the SQL to generate this report from a standard RDBMS).

The examples of implementations in shell, tcl, perl, c and fortran in the remainder of this chapter are also available in the examples section of the Antelope distribution in the directory $ANTELOPE/example/datascope.

At the command line

The command line utilities provide the basic join, sort, and subset functions of dbe as commands that can be joined together into pipes. The commands are named after their function, and have the same names as their corresponding library routines: dbjoin, dbsort, and dbsubset. The output of dbjoin, dbsort, and dbsubset is binary; don't try to display it on your terminal or window. dbselect reads this binary output, and displays selected fields (or expressions).

The output of dbjoin, dbsort, and dbsubset is a simple view -- an array of database pointers with a little bit of header information. This output can be saved to disk and read later. Following the table naming conventions, or by using the -n option, allows a saved view to be read like another table of the database. However, because views contain references to specific record numbers in the base tables, they can be invalidated by changes to original tables. Datascope detects changes by checking modification dates, and will not open out of date views.

Converting the process outlined in dbe above to the command line is straightforward:

1. Begin with the *origin* table, and select only *orid* 645:
   `dbsubset demo.origin "orid == 645" | \`
2. Join this result with the *assoc*, *arrival*, and *site* tables, by piping the results to dbjoin:
   `dbjoin - assoc arrival site | \`
3. Sort by *arrival.time*:
   `dbsort - arrival.time | \`
4. Print out the fields of interest:
   `dbselect - sta arrival.time assoc.delta assoc.phase`

Here's the result:

```
AAK      706139746.35000    1.928 P
KBK      706139751.35000    2.230 P
CHM      706139752.34740    2.289 P
USP      706139752.40005    2.325 P
TKM      706139755.45010    2.566 P
AAK      706139775.33963    1.928 S
KBK      706139784.43397    2.230 S
```

```
CHM       706139787.28949    2.289 S
USP       706139788.12868    2.325 S
```

dbselect does not print epoch times in an easily understandable format; for something easier, specify **`strtime(arrival.time)`** instead of **`arrival.time`**. (To get raw epoch times in dbe, deselect the "Option->Readable times").

The command line utilities are by far the easiest and most concise of the programming interfaces, though they are somewhat limited. Generally, they provide the same kinds of manipulation as dbe. In two areas, however, the command line utilities are (currently) more flexible:

- dbjoin allows you to explicitly specify the join keys between two tables. This makes it possible to join the event and the origin tables on the condition that **`event.prefor == origin.orid`**, for example.
- dbselect allows tables to be grouped by some set of fields, and allows printing a header and footer around each such group.

Database pointers

Before going on to the tcl, c, and FORTRAN interfaces, it is useful to introduce a few concepts and utilities which are used in all three interfaces.

First, most Datascope library calls use a construct called a database pointer. In c, this is declared as a struct, Dbptr. In FORTRAN, it is an array of four integers, and in tcl, it is a list of four integers. A database pointer is used to address a range of database parts, from a specific field in a record in a table, to a whole record, to a whole table, to a whole database. The integers which make up a database pointer index structures internal to Datascope. However, these various integers may be manipulated directly by the user. The various integers are

- database #

  The database number is just the index corresponding to the order in which databases were opened. A typical application may only open one database, in which case database number would be zero.

- table #

  Table number is just the sequence number of the table in the schema file; as views are created, they receive higher numbers.

- field #

  Field number is the sequence number of the field in the table.

- record #

  Record number is the sequence number of the record in the table.

Note that all these numbers follow the c convention of being zero based, starting at zero, rather than the FORTRAN convention of starting at one.

Typically, the user will directly manipulate only the record number and use dblookup to find table and field numbers. The various numbers are dynamic; except possibly for record number, they will not vary during a particular execution of the program, but they may very well change from one execution to the next (if the schema file changes, for instance). That is, the database pointer for a particular field in a particular record of a particular database may be different the next time the program is run.

These indexes may also take on a few special values. dbALL, dbSCRATCH, dbNULL, or dbINVALID. When the field number is dbALL, the database pointer refers to the entire record. Similarly, when the record number is dbALL, the database pointer refers to the entire table. (However, it is not valid to have field number refer to a field and record number be dbALL).

The record number may be dbSCRATCH or dbNULL, in which case the database pointer refers to two special records for a table; the “scratch” record, and the null record. You may assemble a new record in the scratch record before adding it to the table, and you may read null values for fields or records from the null record.

One last special case must be mentioned; a database pointer which specifies a contiguous range of records from a table. In this case, the field index does not refer to a field, but rather is the number of the last record of the range, plus one. A field named *bundle* contains a database pointer which refers to a range of records.

A few programming utilities

Programming in tcl, c, or FORTRAN with Datascope requires becoming familiar with a small set of utility functions, in addition to the Datascope library routines. These utility functions relate to:

- error handling

- time conversions
- associative arrays
- lists
- parameter files

In tcl, associative arrays and lists are part of the language; in c or FORTRAN there are a set of special routines to manipulate them. The relevant man pages are error_handling(3), epoch(3), arr(3), tbl(3), and pf(3).

Error Handling

Libraries should not, as a rule, print out error messages. On the other hand, it's quite difficult to pass adequate information back up the calling sequence and having a set of return codes is very limiting and difficult to accomplish. Consequently, there are a set of routines which maintain an error log. When a library routine encounters a problem, it leaves a verbose message on the error log (by calling elog_log(3)), and then returns a generic error return code. The calling program must manage the error register. Typically, it will want to print the messages in the error register using complain(3).

Programs often benefit from a standard error reporting routine, and the error handling library provides this also. These routines should be initialized by calling elog_init(3). die(3) prints an error message, and exits; complain prints an error message and returns. Both routines print and clear the current error log.

Time conversion

Internally, all time calculations are based on an epoch time, quite similar to the time returned by the UNIX time(3). This time is the nominal seconds since January 1, 1970. In Datascope, it is represented in a double precision number, which results in an accuracy to at least the millisecond range for times of interest. However, a time presented in seconds is difficult to appreciate, so time is often converted between epoch seconds and some more readable format like *June 22, 1992 13:34:45.345*. There are a set of routines for conversion between the two formats. The central routines might be str2epoch, which converts a fairly arbitrary time-date string to epoch seconds, and strtime, which converts an epoch time to a standard format a bit like the example above. Read the man page epoch(3) to learn more, and use the program epoch(1) to perform conversions on the fly.

Associative Arrays

Associative arrays, i.e. arrays indexed by a character string rather than an integer, are quite handy for a variety of problems. One variety of associative arrays are implemented in the arr(3) routines. In this case, the arrays are implemented as balanced (red-black) binary trees. (In contrast, the associative arrays in tcl are implemented as hash tables.) Much of Datascope's internals are implemented using these associative arrays, and Datascope sometimes returns an associative array. Read the man page for more detail. Arrays are not directly accessible from FORTRAN.

Lists

Associative arrays and lists are complementary. Lists and arrays are used extensively in Datascope's internals. Some arguments to Datascope routines are lists, and Datascope sometimes returns lists. Datascope indexes are sorted lists. Lists are declared Tbl *, and the list routines all have the suffix tbl: settbl, pushtbl, gettbl, poptbl, newtbl, etc. Lists are managed as one dimensional arrays (vectors); the array contents are commonly (but need not always be) pointers to strings. The tbl routines provide a variety of common list manipulations; see tbl(3) for more detail. Some list operations are available in FORTRAN.

Parameter files

It's often convenient to keep some configuration data for programs in a separate text file. This allows the program to be more flexible, but avoids the problem of the overloaded command line. The Datascope library doesn't use parameter files directly, but several of the command line utilities and dbe do use parameter files. Basically, the parameter files allow a variety of configuration parameters to be specified by name in a free-format fashion in a central file. This parameter file can also be personalized. Parameter files are read and used as associative arrays (key-value pairs), where the values may be simple strings (scalars), associative arrays or lists. The depth of nesting of arrays and lists has no set limit. Read pf(3) for more detail.

Overview of tcl, perl, c, and fortran solutions

The solutions in tcl, perl, c and FORTRAN to the simple problem above all look quite similar; the differences relate primarily to the different syntax of the languages.

1. Each program begins by opening the database using dbopen. This operation returns a database pointer which specifies the entire database.
2. From this database pointer, other pointers which specify the origin, the assoc, the arrival and the site table are generated, using dblookup.
3. dbsubset is applied to the origin table, returning a view which contains only the one origin.
4. The resulting view is joined using dbjoin, first with assoc, then with arrival, and finally with site.
5. The joined view is sorted using dbsort.
6. Finally, selected fields from each record are fetched using dbgetv, and printed.

Overall, the process is quite similar to the process in dbe or at the command line, but more of the hidden details become evident. Rather than specifying tables directly by name, database pointers are used. The new calls are dbopen, dblookup, dbquery, and dbgetv. dbopen initializes Datascope to allow tables from a database to be used; it returns a database pointer which specifies a database. dblookup is used to further qualify a database pointer to specify a particular table. dbquery is used generally to find ancillary information about a database, table, or field -- in this example, to find the number of records in a table.

Tcl/Tk interface

Tcl is a simple interpretive language designed and written by John Ousterhout. It is widely used on a variety of platforms. It was intended as a kind of glue to put together more complex operations, something like UNIX shells, but with more flexibility. Tk is an x-windows toolkit built on tcl. It allows creating applications with windows and other widgets from a script or interactively.

The source code for tcl/tk is freely available, and so the language can easily be embedded in your own application. Usually, this means adding a few specialized commands to the basic language and then writing the application in the resulting script language. Datascope takes this approach, providing two shells: *atcl* (no x windows interface) and *awish* (a windowing shell).

A program in tcl is a script, which may be run at the command line directly. Alternatively, you can just run atcl or awish, and type in the lines. This latter approach may be more instructive, and is a useful approach when writing your own scripts; try them out directly. Here's the script:

```
#!/bin/sh
```

```
# This comment extends to the next line for tcl \
exec atcl -f $0 $*

package require Datascope
set db [dbopen demo r+]
set db [dblookup $db 0 origin 0 0]
set db [dbsubset $db {orid == 645}]
set db [dbjoin $db [dblookup $db 0 assoc 0 0]]
set db [dbjoin $db [dblookup $db 0 arrival 0 0]]
set db [dbjoin $db [dblookup $db 0 site 0 0]]
set db [dbsort $db arrival.time]
loop i 0 [dbquery $db dbRECORD_COUNT] {
  puts [dbgetv $db 0 $i sta arrival.time assoc.delta
assoc.phase]
}
```

If you're familiar with shell programming, this will all look pretty familiar. The first three lines are a trick to get awish running regardless of where it's installed on your system. In the body of the program, the square brackets replace the shell's back-tick, and cause a command to be executed before the rest of the line, with the results of the execution replacing that part of the line. The curly braces correspond to single quotes in the bourne shell. These replacements allows embedding quotes within quotes, and commands within commands in a tcl command.

Here are the results:

```
AAK      706139746.35000    1.928 P
KBK      706139751.35000    2.230 P
CHM      706139752.34740    2.289 P
USP      706139752.40005    2.325 P
TKM      706139755.45010    2.566 P
AAK      706139775.33963    1.928 S
KBK      706139784.43397    2.230 S
CHM      706139787.28949    2.289 S
USP      706139788.12868    2.325 S
```

Once again, there is no conversion from epoch seconds to a readable format. However, the standard time conversion routines are available inside tcl.

In tcl, an error return from a Datascope library call raises an exception, which prints a message, and stops the execution of a script. This is quite different from c or FORTRAN, where you must yourself test the return code. To override this behavior, use the tcl catch command.

Variables need not be declared in tcl. Database pointers are kept as lists in a variable; try **`puts $db`** to see the contents. Generally, the arguments are the same as the arguments in c, but since all tcl variables resolve to strings, there is no need to put quotes around names like demo, origin, and arrival.time above.

Documentation on the tcl interface is limited to usage lines in the man page for dbwish(1). It will often be useful to read the c man page when more detail about the arguments or results is needed.

If you need to learn tcl/tk, a good place to start is John Ousterhout's book, *Tcl and the Tk Toolkit*. Once you've begun writing programs, you'll probably find the dbhelp program to be quite useful, in much the same way the man pages are useful when programming in c.

The perl interface

The perl interface is quite similar, but with a perl flavor. In perl, the database pointers are kept in perl lists. This interface requires the 5.0 version of perl, not the older 4.036 version.

Here's the perl routine:

```
#! /usr/bin/perl

use Datascope ;

@db = dbopen ( "demo", "r" ) ;
@db = dblookup(@db, "", "origin", "", "" ) ;
@db = dbsubset(@db, "orid == 645" ) ;
@db = dbjoin(@db, dblookup(@db, "", "assoc", "", "") );
@db = dbjoin(@db, dblookup(@db, "", "arrival", "", "") ) ;
```

```
@db = dbjoin(@db, dblookup(@db, “”, “site”, “”, “”) ) ;
@db = dbsort(@db, “arrival.time” ) ;

$nrecords = dbquery(@db, “dbRECORD_COUNT”) ;
for ( $db[3] = 0 ; $db[3] < $nrecords ; $db[3]++ ) {
    ($sta, $time, $delta, $phase) = dbgetv (@db,
        qw(sta arrival.time assoc.delta assoc.phase));
    print “$sta $time $delta $phase\n” ;
}
```

The results are identical with the tcl results.

The c interface

Most routines in the c interface have a return code; typically zero indicates success, and a negative value, usually dbINVALID, indicates failure. Some routines return a database pointer, rather than an integer return code. In these cases, an error is indicated by individual components of the database pointer taking the value dbINVALID. (Except for dblookup, all the components take the value dbINVALID). Error messages are left on the error log. To print them, use complain(3), die(3), or clear_register(3).

Here’s the program in c:

```
#include <stdio.h>
#include “db.h”
int main()
{
    Dbptr db, dbassoc, dbarrival, dbsite ;
    int n  ;
    double arrival_time, delta ;
    char sta[25], phase[10] ;
    Tbl *sortkeys ;
    if ( dbopen ( “demo”, “r+”, &db ) < 0 )
     die ( 0, “Can’t open table knet.origin.” ) ;
    db = dblookup ( db, 0, “origin”, 0, 0 ) ;
    db = dbsubset ( db, “orid == 645", 0 ) ;
   dbassoc = dblookup ( db, 0, “assoc”, 0, 0 ) ;
    db = dbjoin ( db, dbassoc, 0, 0, 0, 0, 0 ) ;
   dbarrival = dblookup ( db, 0, “arrival”, 0, 0
```

```
) ;
   db = dbjoin ( db, dbarrival, 0, 0, 0, 0, 0 ) ;
    dbsite = dblookup ( db, 0, “site”, 0, 0 ) ;
    db = dbjoin ( db, dbsite, 0, 0, 0, 0, 0 ) ;
     sortkeys = strtbl(“arrival.time”, 0 ) ;
     db = dbsort ( db, sortkeys, 0, 0 ) ;
     dbquery ( db, dbRECORD_COUNT, &n ) ;
      for ( db.record = 0 ; db.record < n ;
db.record++ ) {
        dbgetv ( db, 0,
            “sta”, sta,
            “arrival.time”, &arrival_time,
            “assoc.delta”, &delta,
            “assoc.phase”, phase,
            0 ) ;
     printf ( “%-5s %17.3lf %10.3lf %-2s \n”,
          sta, arrival_time, delta, phase ) ;
        }
    return 0;
}
```

The output from this program is the same as the shell and tcl versions, with some minor differences in the number of significant digits printed, and the number of blanks between columns.

Disregarding the syntactical differences because of the different languages, the significant difference between this and the tcl version is the extra arguments to **dbjoin**. These extra, optional arguments provide some added flexibility, allowing you to specify the join keys explicitly, and possibly returning a list of records which don’t join. In most cases, you’ll set these extra arguments to zero, but refer to the man page for more information about them. Another difference is the way the sort keys are specified to dbsort; the keys are packed into a list, and this list is passed to **dbsort**. Both of these differences derive from the ability in tcl to have a variable argument list with defaults.

The FORTRAN interface

The FORTRAN interface is as similar to the c interface as was practical. The names are the same, but routines which return database pointers are changed to subroutines with an extra argument added for the return values. Possibly all the routines should have been changed to subroutines, since the rest return integers, and have to be declared in the include file, since their names don’t start with the right letters (i-

n). It seems to be impossible to get the FORTRAN compiler to shut up about these unused “variables” if you never call them.

Here’s the solution in fortran:

```
#include “db.i”
integer db(4), dbt(4)
real *8 time, arrtime, delta
character*15 sta, phase
if (dbopen(“demo”,“r+”,db ) .lt. 0)
     *    call die ( 0, “Can’t open database” )
      call dblookup( db, db, “”, “origin”, “”, “” )
call dbsubset ( db, db, “orid == 645", ”“ )
call dblookup ( dbt, db, “”, “assoc”, “”, “” )
call dbjoin( db, db, dbt, 0, 0, 0, 0, “” )
call dblookup ( dbt, db, “”, “arrival”, “”, “” )
call dbjoin( db, db, dbt, 0, 0, 0, 0, “” )
call dblookup ( dbt, db, “”, “site”, “”, “” )
call dbjoin( db, db, dbt, 0, 0, 0, 0, “” )
call strtbl(keys, “arrival.time”, 0 )
call dbsort ( db, db, keys, 0, “” )
call dbquery ( db, dbRECORD_COUNT, n )
do i=0, n-1
    db(4) = i
    ierr = dbgetv ( db, 0,
     *“sta”, sta,
     *“arrival.time”, arrtime,
     *“assoc.delta”, delta,
     *“assoc.phase”, phase,
     *0 )
    write (*,10) sta, arrtime, delta, phase
10          format ( a10, f17.3, x, f10.3, x, a3  )
end do
end
```

The result from this program differs from the c version programs only in the number of spaces between fields.

Summary

The example problem is quite simple, of course, but shows the use of the primary operations from several interfaces. It did not address, however, the issues of modifying a field (dbputv), adding a record (dbaddv), or creating a new database by copying the various records from a view into a new table (dbunjoin). There are man pages for these routines, and the calls follow the general outline of the calls used above: dbputv and dbaddv are quite similar to dbgetv. dbquery has many more codes than dbRECORD_COUNT used above; some codes return character strings, lists or associative arrays. Almost any information which can be read from the schema file may also be obtained from dbquery with the proper code; dbhelp exploits this to make it easy to explore a schema.

There are detailed man pages for all the c interface library calls; the man pages for the other interfaces are a bit more skimpy. To see other calls which might be of interest consult the manual pages.

In the examples above, very little error checking was performed. Real programs should take more care.

The programs described above may be found in the antelope distribution in the directory $ANTELOPE/example/datascope. You'll also find a Makefile there which can be used to produce the executables. The Makefile is fairly simple:

```
BIN= xc xf xperl xsh xtcl
ldlibs=$(DBLIBS)
include $(ANTELOPEMAKE)
```

To learn how it works, please consult the Antelope Software Development Tutorial.

CHAPTER 7

Datascope Utilities

There are a number of specialized programs that add functionality to Datascope. This chapter introduces the utilities; refer to the man pages for detailed information.

dbverify

provides overall consistency checks on a database. It uses the range expressions from the schema file to check that each field has a legal value, and it can be used to perform other simple consistency checks on a database, for instance, checking to see that tables join properly. Many of these additional tests are configurable in a parameter file, so you may add your own special tests.

dbcheck

checks each line (record) of each file in a database for the correct length. Datascope requires that files which represent tables be composed of lines which all have the same length. If you edit these files with a text editor, it is easy to introduce errors by adding or deleting a blank, or converting several blanks to a tab. dbcheck looks for problems like this.

dbdiff

compares two databases, with output resembling that of sdiff. Comparing databases is difficult, and dbdiff does not do a complete job, but is very useful in some situations regardless.

dbdoc

prepares tbl/troff style documentation directly from the schema. This can be useful for generating hardcopy which is more readable than the schema. For interactive use, dbhelp is more convenient.

dbset

simplifies global changes across multiple tables. For instance, you might want to change a station name, say from *AAK* to *AKA*, in every table. dbset makes this easy.

dbfixids

The id fields in a database typically have no intrinsic meaning; they are arbitrary integers which provide a simple key to a table, and are often used to join two tables. Nevertheless, it is sometimes useful to renumber these ids. This must be done carefully, since a particular id may be present in several tables, and all the ids must be changed in a consistent manner.

dbcrunch

Because views and indexes refer directly to a record number, many programs (like dbe) do not usually delete records immediately; instead, all fields in a record are set to the null value: marked for later removal. dbcrunch removes such null records from a database or table.

dbnextid

When a new record is added to certain tables, a unique id must be generated. As explained previously, these new ids are generally found by consulting the *lastid* table, which saves the value of the previously largest integer used, and then using the next larger integer. (If the lastid table is not present, Datascope looks through the entire table which defines the id in question, and creates a lastid table). dbnextid provides command line access to this function.

dbcp

Just as it's useful to copy a file, it's often convenient to create a new copy of a database, either to modify it, or to prevent it from being modified. dbcp copies a database to a new location. It can also be used to create a descriptor file referencing the original database, or to copy a single database table.

dbremark

The CSS database has a special *remark* table which may be used to add a comment to particular records of selected tables. dbremark provides a method of printing or adding a comment at the command line.

dbaddv

is a command line program for adding records to a database. It has an interactive mode, or can add a single record directly from the command line, or multiple records by reading from stdin.

dbcalc

allows expressions to be evaluated against a single record of a table or view. This is most useful for testing the syntax of the expression.

dbconvert

is a general purpose tool for converting from one schema to another. It's a bit slow, but can be very useful when your first attempt at a schema left out some important fields or made some fields too small. You can create your new schema (with dbdesign) and then use dbconvert to convert any old databases to the new schema.

dbdesign

is a GUI program which simplifies the process of creating new schema files. It can also be used to edit old schemas, or to create schema files which extend an existing schema.

dbinfer

is an ad hoc program which reads a single fixed field length record and creates a schema file for a corresponding table, using the whitespace to define fields. This may be useful as a first cut at the eventual schema file.

dbdestroy

removes all the tables of a database; with an option, it will also remove all the external files (like waveform files).