BARRODALE COMPUTING SERVICES LTD.

DBXten Extension for PostgreSQL (Linux Version)

Programmer's Guide

Version 1.10.0.3, October 7, 2011

DBXten Extension for PostgreSQL (Linux Version) Programmer's Guide

© Barrodale Computing Services Ltd.

http://www.barrodale.com

Table of Contents

Chapter 1: Tuple Data and the BCS DBXten Extension – An Introduction 1
Storage and Representation of Tuple Data 2
“Strong” Entities and “Weak” Entities 2
Representation of Parent and Child Entities in a Database .. 4
Some Queries Involving Parents and Children 5
Issues When *N* Becomes Large 6
Storing Child Tuples in Blocks 6
Retrieving Tuples from Blocks 9
What are the Benefits of the Tuple Block Implementation? 10
But What About First Normal Form? 10
Features of Tuples that Can Be Exploited – What Tuple Features Make Tuple Block Storage Particularly Suitable? 11
Tuple Block Schemas for Various Applications 13
Instrument Measurements 13
Airline Flight 14
GIS Object 15
Stock Market 16
Delivery Tracking 17
Inventory 18
Implementing Tuple Blocks – DBXten 18
Chapter 2: Installing the BCS DBXten Extension 19
Installing the Software 19
Setting up the License Key 21
Installing the “Cube” Extension 22
Building Sample Programs and Testing the Installation 23
Determining the DBXten Software Version Number 24
Chapter 3: The BCS DBXten Database Extension – the DSChip Data Type 25
The DSChip Data Type 25
Units Support 27
Data Types that are Supported Inside a DSChip 28
Date/Time Conversion Functions 28
Chapter 4: Getting Data into DSChip's 29
Using the DBXten SQL API to Insert Data 29
Creating an Empty DSChip 29

Adding Tuples to a DSChip 31
Indexing DSChip's... 35
Other Functions.. 37
Using a Utility Loader to Insert Data 38
Using the DBXten C API to Insert Data.. 39
General Structure of Loading Programs............................. 39
A Sample C DSChip Loading Program 39
Using the DBXten Java API to Insert Data.................................... 45
A Sample Java DSChip Loading Program 45
Chapter 5: Retrieving Data from DSChip's 49
Options for Extracting Data 50
Other Processing Paths 52
Using the DBXten SQL API to Extract Data.................................. 53
Determining What Columns are in a DSChip 54
Listing Values for DSChip Column(s) 54
Listing Distinct Values for DSChip Column(s) 54
Determining Whether a DSChip has a Particular Column(s)... 55
Determining How Many Columns a DSChip has................ 55
Determining the Number of Tuples in a DSChip 55
Determining the Maximum Value for Columns in a DSChip56
Determining the Minimum Value for Columns in a DSChip 56
Getting Column Values from Single-Tuple DSChip's 56
Listing Compression Information for a DSChip................... 57
Converting a DSChip Range to a Cube............................. 58
Generating a Bounding Cube from a DSChip 58
Doing a Region of Interest Query to Select DSChip's 58
Extracting Tuples from a DSChip (no tuple filtering)........... 60
Counting Matches without Extracting 62
Tuple Filtering DSChip's into New DSChip's...................... 62
Tuple Filtering DSChip's into a Stream of Bytes................. 64
Tuple Filtering DSChip's into a Set of Tuples..................... 64
Using the DBXten C API to Extract Data....................................... 66
Performing Tuple Filtering on the Client 66
Performing Tuple Filtering on the Server............................ 69
Using the DBXten Java API to Extract Data.................................. 73
Chapter 6: Updating Data in the Database 77
Chapter 7: Database Management Issues 79
Database Security 79

Chapter 8: Troubleshooting Guide 81
Connection Errors 81
C Client Library Errors 81
General Operational Errors 82
Corrupt or Misinterpreted Binary Data 82
Bad ASCII Data 83
Bad DateTime Data 85
Appendix A: Complete List of BCS DBXten Extension User-Defined Routines (UDRs) 91
Appendix B: The C API 93
List of C API Constants 93
List of C API Functions 93
Appendix C: The Java API 99
List of Java API Functions 99
Appendix D: A Tutorial 104
Appendix E: CSV File Reader Utility 105
Downloading the CSV File Reader Utility 105
Compiling the CSV File Reader Utility 106
Running the CSV File Reader Utility 106
Appendix F: NetCDF File Reader Utility 110
Overview 110
The Run-Control File 110
Downloading the NetCDF File Reader Program 111
Compiling 112
Running the NetCDF File Reader Utility 112
Appendix G: CSV File Reordering Utility 115
Overview 115
Using the CSV File Reordering Utility 115
Appendix H: Tile Optimization Utility 119
Overview 119
Usage 120
Limitations 120
Example usage 121
Using tileOptimizer in conjunction with reorderCsv and csvChipLoader 121

List of Figures

Figure 1: Traditional Implementation of a Parent-Child Entity Relationship. 4
Figure 2: Alternative Implementation of a Parent-Child Entity Relationship. 7
Figure 3: Looking Inside a Tuple Block. 7
Figure 4: Logical View of a Set of Tuple Blocks. 8
Figure 5: Logical View of a Set of Tuple Extents. 9
Figure 6: Instrument Measurements application schema. 13
Figure 7: Airline Flight application schema. 14
Figure 8: GIS Object application schema. 15
Figure 9: Stock Market application schema. 16
Figure 10: Delivery Tracking application schema. 17
Figure 11: Inventory application schema. 18
Figure 12: Syntax of a DSChip schema. 30
Figure 13: Two-dimensional DSChip's. 37
Figure 14: DSChip Extraction Processing paths. 50
Figure 15: DSChip's A and B overlap the specified X and Y ranges. 51
Figure 16: Syntax of a `columnNames` parameter. 53
Figure 17: Syntax of a `rangeSpec` parameter. 53

Documentation Conventions

This section defines the conventions used in this document. The conventions include typographical conventions and icon conventions.

Typographical Conventions

This manual uses the following typographical conventions:

Convention	Meaning
KEYWORD	Programming language keywords (i.e., SQL, C keywords) appear in a serif font.
italics ***italics*** `italics`	*New terms, emphasized words, and variable values appear in italics.*
`User input`	`Computer generated text (e.g., error messages) and user input appear in a non-proportional font.`
<POSTGRESQLDIR>	The directory where the PostgreSQL server was installed. This value varies depending on the flavor of Linux. Some possible values are `/opt/postgres`, `/usr/local/pgsql`, and `/usr/lib/postgresql/`*version*.
<DBXTENDIR>	The directory where DBXten is installed. By default, this is `/opt/DBXten`.
(ehportsopa) '	An apostrophe is used in the plural form of data types (e.g., DSChip's)

Icon Conventions

This manual uses the following icon conventions to highlight passages in the manual:

Icon	Label	Description
	Warning:	Identifies paragraphs that contain vital instructions, cautions, or critical information.
	Important:	Identifies paragraphs that contain significant information about the feature or operation that is being described.
	Tip:	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described.

What's New in This Version?

The following table lists the features that have been added to this version of the BCS DBXten Extension:

Feature	Manual Sections Where Feature is Described.
Tile Optimization utility	Appendix H (page 119)
CSV File Reordering utility	Appendix G (page 115)
Utility executables now supplied with DBXten	Appendices E, F, G, H

Chapter

1

Chapter 1: Tuple Data and the BCS DBXten Extension – An Introduction

One of the many ways that data about an object can be represented is as a "tuple" – an ordered set of values, each of which describes some aspect of the object. Tuples in turn are often then stored as lines in spreadsheets or rows in database tables. This chapter reviews how tuples are often stored in a database and then offers an alternative implementation that, for a wide class of applications, is often logically more appropriate and physically more efficient.

Storage and Representation of Tuple Data

Datasets consisting of a collection of tuples[1] occur naturally in many different types of computer applications. Some examples are:

Application Area	Tuple Columns
Instrument measurement	date, time, location, measured value
Airline flight	date, time, airline, flight number, latitude, longitude, altitude, heading, speed
GIS object	feature id, shape point number, latitude of shape point, longitude of shape point
Stock market	date, time, stock code, bid value, ask value
Delivery tracking	order item number, date, time, location, activity
Inventory	product code, date, time, inventory
Human Resources	employee number, employee name, department number, salary
	department number, department name

The conventional approach to storing tuple data is to store each tuple as a row in a suitably-defined database table, for example:

```
INSERT INTO inventory_table(product_code, date, time, store_id,
inventory) VALUES('P1234','2008-01-29','12:43:13','S123',45);
```

"Strong" Entities and "Weak" Entities

Sometimes tuples represent an instance of a real-world standalone entity (called a "Strong Entity"), but often the entity represented by a tuple is not standalone – we call such an entity a "Weak Entity."

The "Human Resources" example above illustrates two instances of strong entities: "Employee" and "Department". Strong entities have the following characteristics:

1) They may or may not take part in relationships. In the Human Resources case the two entities are related (employees work for departments) but it is easy to imagine applications that involve just one of these entities.

2) The existence of one instance of a strong entity does not depend on the existence of instances of any other entity. So for example an employee can be removed without necessitating the removal of departments, and a

[1] A tuple (or row) is a fixed grouping of elements (columns), each of a particular type. Each row of a spreadsheet, for example, can be considered to be a tuple.

department can be removed without necessitating the removal of employees (assuming employees would be moved to a different department).

All of the other examples above are examples of weak entities. Weak entities have the following characteristics:

1) They sit at the *many* side of a *1-to-many* relationship.

2) They are the children of a single (strong) parent entity.

3) Their existence depends on the existence of the parent entity. Removal of the parent entity logically necessitates the removal of its (weak) children.

A special class of weak entities consists of ones that take part in no relationships other than ones through their parent. In this discussion we will restrict ourselves to this special class.

For the examples above, the following table shows how weak entities are related to their parent.

Application Area	**Weak Entity Tuples (columns as listed above)**	**Parent Entity (and columns)**
Instrument measurement	Instrument measurement	Instrument (instrument type, model, serial number)
Airline flight	Flight path	Airline flight (airline, flight number, departure city, departure time, arrival city, arrival time)
GIS object	Shape points	GIS feature (id, type, name)
Stock market	Stock quotes	Stock (abbreviation, company name)
Delivery tracking	Delivery history	Orders (order number)
Inventory	Inventory history	Products (product code, product name)

The distinction between strong and weak entities is critical to the suitability of DBXten. This will be described in a later section, but first we will discuss how parent and child entities (of either the strong or weak type) are represented in a database.

Representation of Parent and Child Entities in a Database

As discussed in the previous section, both strong and weak entities can take part in parent-child relationships (and weak entities ***always*** do).The traditional way to represent entities and their parent-child relationship in a database is to store child entity tuples in one table, parent entity tuples in another table, and to use primary and foreign key relationships to link the children with their parent. For example, following the Instrument measurement example above we might generate the following database objects:

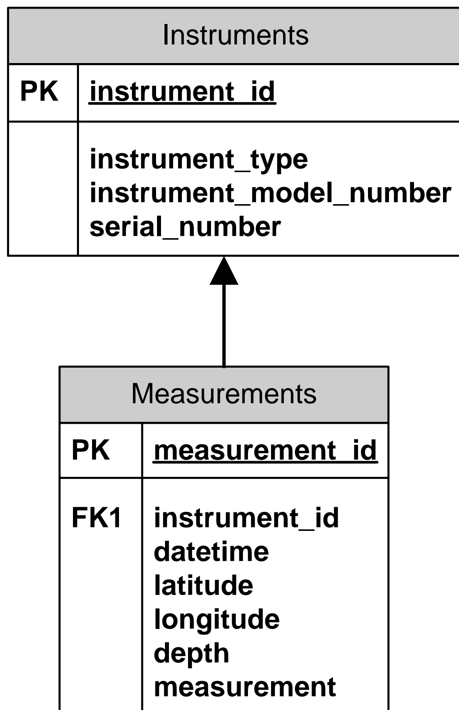


Figure 1: Traditional Implementation of a Parent-Child Entity Relationship[2].

```
CREATE TABLE instruments (instrument_id INTEGER PRIMARY KEY,
                        instrument_type INTEGER,
                        instrument_model_number VARCHAR(30),
                        serial_number VARCHAR(30));
CREATE TABLE measurements (measurement_id INTEGER PRIMARY KEY,
                        instrument_id INTEGER REFERENCES
                              instruments(instrument_id),
                        datetime TIMESTAMP,
```

[2] In this figure, "PK" denotes a primary key (uniquely identifying) column and "FK1" denotes a foreign key column. Each foreign key value in the child table identifies a primary key value from the parent table. The (unique) tuple in the parent table that has that value is the child's parent.

```
                           latitude FLOAT,
                           longitude FLOAT,
                           depth FLOAT,
                           measurement FLOAT);
```

Some Queries Involving Parents and Children

Assuming that we have tuples stored in the instruments and measurements tables shown in the previous section, the following is a list of queries and other operations that we might want to perform with respect to those tuples:

Q1) Insert a series of measurements for a particular instrument.

Q2) Replace one or more measurements for a particular instrument.

Q3) Delete one or more measurements for a particular instrument.

Q4) Remove an instrument and all its measurements.

Q5) Find all the measurements taken from a particular instrument within a particular time period and geographic area (represented by northing and easting values).

Q6) Find which instruments have produced a measurement between 99.3 and 99.7. What were the measurement values and when did they occur?

These queries can all be expressed quite easily using SQL. For example, Q5 can be written as:

```
SELECT measurement
FROM instruments i, measurements m
WHERE i.instrument_id = m.instrument_id
AND i.serial_number = idOfInstrument
AND m.datetime BETWEEN time1 AND time2
AND contains(areaOfInterest,m.northing,m.easting);
```

Q6 can be written as:

```
SELECT i.serial_number, m.measurement, m.datetime
FROM instruments i, measurements m
WHERE i.instrument_id = m.instrument_id
AND m.measurement BETWEEN 99.3 and 99.7;
```

Efficient execution of these queries would require indices to be built on i.serial_number (for Q5), m.instrument_id (for Q5 and Q6), m.datetime (for Q5), and m.measurement (for Q6). In addition, a spatial index on m.northing/m.easting would help with Q5.

As will be discussed in the next section, efficiency becomes a real concern once the number of measurements (the *N* in the 1-N instrument-measurement relationship) becomes large. And the indexes, so crucial in allowing queries to be answered quickly, actually become part of the problem.

Issues When *N* Becomes Large

The database schema described in the previous section works quite well when the number of children ***N*** of a parent remains relatively small (on the order of tens or hundreds). When ***N*** becomes large, however, one is likely to notice the following:

1) The space taken by the child table ("measurements", in the example above) becomes very large and starts to consume a disproportionate amount of resources (disk space, CPU cycles, DBA time, etc.). Activities that might be performed by a DBA for smaller tables over the lunch hour need to be postponed until the weekend.

2) The total space consumed by indexes also becomes large.

3) The total overhead involved in inserting and deleting rows (including the time spent logging, updating indexes, and checking foreign key constraints, etc.) becomes large.

4) If child tuples are generated at a fast rate (as measurements from a scientific instrument might be) the database might not be able to "keep up".

In the next section we will see how an alternative representation can be used to address these issues.

Storing Child Tuples in Blocks

An alternative to the design presented above, where we stored one measurement per row of the measurements table, is to store multiple measurements in each row, as is done by DBXten.

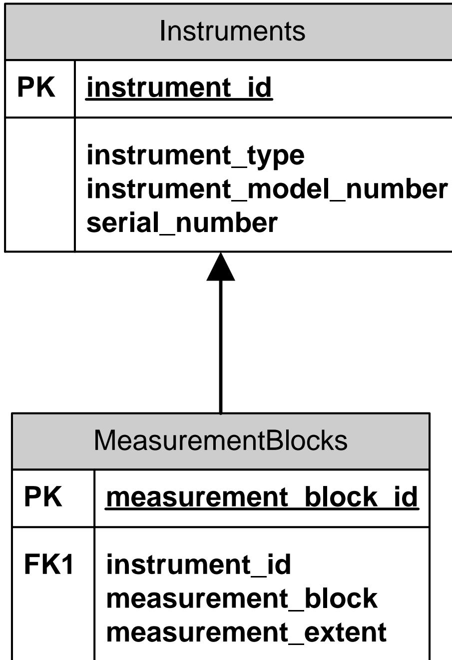


Figure 2: Alternative Implementation of a Parent-Child Entity Relationship.

In the above diagram, "measurement block" is a binary large object (blob) that contains all the information from a group of measurements. Ignoring for the moment how to get information into and out of this blob, consider the blob to be a physical implementation of a logical table consisting of rows of measurements:

Measurements
datetime latitude longitude depth measurement

Figure 3: Looking Inside a Tuple Block.

A sequence of these measurement blocks may look like the following[3]:

	Datetime	Latitude	Longitude	Depth	Measurement
Block 1:	2008-02-01 00:12:23	46.343	-127.386	14.34	10.2
	2008-02-01 00:12:25	46.344	-127.385	16.82	11.9
	2008-02-01 00:12:27	46.345	-127.383	18.85	10.7
	2008-02-01 00:12:29	46.346	-127.382	21.22	11.7
	2008-02-01 00:12:31	46.347	-127.381	23.66	10.9
	2008-02-01 00:12:33	46.349	-127.379	26.05	11.4
	…	…	…	…	…
	2008-02-01 00:13:43	46.392	-127.335	104.82	10.7
Block 2:	2008-02-01 00:13:45	46.394	-127.334	106.83	10.7
	2008-02-01 00:13:47	46.395	-127.332	108.90	13.1
	2008-02-01 00:13:49	46.396	-127.331	110.99	10.7
	2008-02-01 00:13:51	46.398	-127.330	113.32	11.4
	2008-02-01 00:13:53	46.399	-127.328	115.38	10.2
	2008-02-01 00:13:55	46.400	-127.327	117.65	10.9
	…	…	…	…	…
	2008-02-01 00:14:59	46.439	-127.289	188.40	10.3
Block 3:	2008-02-01 00:15:01	46.441	-127.287	190.88	9.7
	2008-02-01 00:15:03	46.442	-127.286	193.22	11.9
	2008-02-01 00:15:05	46.443	-127.285	195.65	11.5
	2008-02-01 00:15:07	46.444	-127.283	197.86	10.5
	2008-02-01 00:15:09	46.446	-127.282	200.02	11.2
	2008-02-01 00:15:11	46.447	-127.281	202.38	10.7
	…	…	…	…	…
	2008-02-01 00:16:15	46.486	-127.241	274.68	11.3

Figure 4: Logical View of a Set of Tuple Blocks.

[3] This is a ***logical*** view of the contents of the measurement tuple blocks. We'll discuss the physical representation later in Chapter 3.

Along with each measurement_block in the new MeasurementBlocks table we can also store a tuple extent value called "measurement_extent." These tuple extents store the minimum and maximum values from the corresponding tuple blocks:

	Datetime	Latitude	Longitude	Depth	Measurement
Extent 1:	2008-02-01 00:12:23	46.343	-127.386	14.34	10.2
	2008-02-01 00:13:43	46.392	-127.335	104.82	11.9
Extent 2:	2008-02-01 00:13:45	46.394	-127.334	106.83	10.2
	2008-02-01 00:14:59	46.439	-127.289	188.40	13.1
Extent 3:	2008-02-01 00:15:01	46.441	-127.287	190.88	9.7
	2008-02-01 00:16:15	46.486	-127.241	274.68	11.9

Figure 5: Logical View of a Set of Tuple Extents.

Again we're ignoring for the moment how to get data into and out of a tuple extent and how they are stored internally. Suffice it to say, though, that these tuple extent values can be indexed (with a multidimensional index) in such a way that the blocks containing particular component values (latitude, longitude, datetime, depth, measurement) can be efficiently identified. For example, an index on the above tuple extents would tell us that if we were looking for measurements with values greater than 12 then we would just have to look inside block 2.

Retrieving Tuples from Blocks

With this alternative design we can re-express the Q5 and Q6 queries presented earlier with the following pseudo-SQL[4]:

Q5 can be written as:

```
SELECT extractFromBlock("measurement", measurement_block,
                                 filterExpression)
FROM instruments i, measurementBlocks m
WHERE i.instrument_id = m.instrument_id
AND i.serial_number = idOfInstrument
AND overlap(filterExpression, measurement_extent);
```

[4] The actual SQL used with DBXten is slightly more complicated and will be explained in Chapter 5.

Q6 can be written as:

```
SELECT i.serial_number,extractFromBlock("measurement,datetime",
measurement_block,filterExpression)
FROM instruments i, measurementBlocks m
WHERE i.instrument_id = m.instrument_id
AND overlap(filterExpression, measurement_extent);
```

In both queries, filterExpression is used twice – once in the "overlap" clause to eliminate from consideration any tuple blocks where all the tuples have component values that fall outside the area of interest, and once in the "SELECT" clause to eliminate from the remaining blocks any individual tuples where the component values do not fall within the area of interest.

Example

Consider the tuple blocks shown above and suppose, for example, that the filter expression says that we only want rows where the measurement value is between 12.0 and 14.0. Then the overlap clause would restrict our search to block 2 and the SELECT clause would further restrict the results to just row 2 of that block (assuming no rows hidden in the "…" rows qualify).

What are the Benefits of the Tuple Block Implementation?

At first glance it may appear that the alternative implementation offers no advantage over the traditional one. However,

- The alternative implementation consumes less index space, since there is just one index entry per block rather than per tuple.
- Since there are fewer rows there is less row overhead.
- Looking at the tuple block example above one can see that there is a lot of redundancy in each column of the blocks (for example the dates are all the same and the time values differ from one another consistently by 2 seconds). Hence there is a lot of potential for compression – DBXten can exploit that by applying any of its many compression algorithms. Less disk space usage translates into faster access times.

But What About First Normal Form?

Database purists may point out that the alternative design violates "First Normal Form" (and hence higher normal forms as well) since each row in the measurement_block table (in our example) represents a repeating group of measurement entity values. This is really only a concern, however, if individual measurements take part in relationships with other entities in the database. For example if one of the columns in the measurement tuple pointed to rows in another table, then the design could lead to problems in enforcing referential

integrity[5]. If the columns do not take part in relationships – i.e., the entity is "weak" in the way described above – then there is no penalty in using the alternative design.

Features of Tuples that Can Be Exploited – What Tuple Features Make Tuple Block Storage Particularly Suitable?

This section summarizes the features of tuples that make them particularly well-suited to being stored using the tuple block alternative implementation that DBXten uses.

The DBXten tuple block alternative is a particularly good choice when:

1) The tuple values are of known, limited, precision. This is not a requirement, but DBXten can exploit limited precision if it's told to. For example, if it's known that measurement values are accurate to just three significant digits then a lot of space can be saved if we don't try to preserve seven digit precision.

2) The tuple data are primarily stored once and retrieved often. If your application does not involve lots of updating and deletion of tuples then DBXten is a particularly good choice.

3) The tuple values have some natural ordering (e.g., the time value is increasing). This can increase the redundancy within a block and hence increase the storage savings. It may also reduce the number of blocks that need to be read to answer most queries (see point 5 as well).

4) The specific columns that make up a tuple change over time. With DBXten it is possible to store tuple blocks having different structures in the same column of the same table. So when an instrument starts recording a new type of measurement this new measurement type can be accommodated without performing an expensive reorganization of the measurements table.

5) There is redundancy in the tuple data. As explained above, redundancy can be exploited to achieve high compression. The following is a list of some of the more simple compression techniques used by DBXten:

   i) **Run-length Encoding**: if a significant portion of the values in a list are duplicates of an adjacent value, a run-length encoding strategy

[5] Suppose for example that we used a tuple block scheme to store tuples of employee information, and one of the columns in the employee tuple was a department identifier. Then each block could point to multiple departments and it would be hard to enforce the rule that each employee must be in exactly one department.

may be employed. See http://en.wikipedia.org/wiki/Run-length_encoding for a general description of run-length encoding. The DBXten implementation uses signed one byte integer counts; -128 to -1 representing runs of unrepeated values of length 128 to 1 respectively, 0 to 127 representing runs of repeated values from 2 to 129 respectively. The run counts are stored separately from the run values so that the run values can be further compressed by another compression strategy.

ii) **Arithmetic Series Encoding**: if a list can be expressed as an arithmetic series of the form $a_i = r + (i-1)*d$ for positive integer i and some constants r and d, the list may be reduced to two values, the start value r and d.

iii) **Arithmetic Cycle Encoding**: if a list can be expressed as a repeating arithmetic series of the form $a_i = r + ((i+p) \bmod q)*d$ for integer $i > 0$ and positive integer constants p and q, and some constants r and d, then the list may be reduced to four values: p, q, r and d.

iv) **Fixed Point Encoding**: If a list can be expressed in the form $a_i = r + s_i * d$ where all s_i are integers expressible in one byte, or all s_i are integers expressible in two bytes, or all s_i are integers expressible in three bytes, then the list may be reduced to a byte value specifying how many bytes s_i requires, the value r, the value d, and the values s_i.

v) **Delta Encoding**: If a list can be expressed in the form $a_i = a_{i-1} + s_i * d$ for $i > 1$ and s_i are integers expressible in one byte, or all s_i are integers expressible in two bytes, or all s_i are integers expressible in three bytes, then the list may be reduced to a byte value specifying how many bytes s_i requires, the value a_1, the value d, and the values s_i.

vi) **Float Encoding**: If a list of floating point values can be expressed as 32-bit floating point values instead of 64-bit floating point values while not violating the precision requirements, the list may be reduced to the 32-bit floats.

These and other more sophisticated compression algorithms are applied appropriately and automatically by DBXten.

6) Tuple values can be, or are already naturally, localized – it's possible to arrange tuples in blocks in a way that will minimize the number of

blocks needed to answer most queries. For example, if measurements arrive in time order and there is a high correlation between insertion time and position (as in the example above), then queries that specify a time or spatial region of interest will naturally restrict the number of blocks that need to be examined.

Tuple Block Schemas for Various Applications

This section illustrates how each of the weak entity tuple scenarios described earlier can be implemented using tuple blocks.

Note that these references do not have the "_extent" columns in the child tables. These columns were useful in the above text for explaining the logistics of extracting desired data from tuple blocks. However, with DBXten these columns are not strictly necessary since indexes can be built on the tuple block columns themselves.

Instrument Measurements

Schema

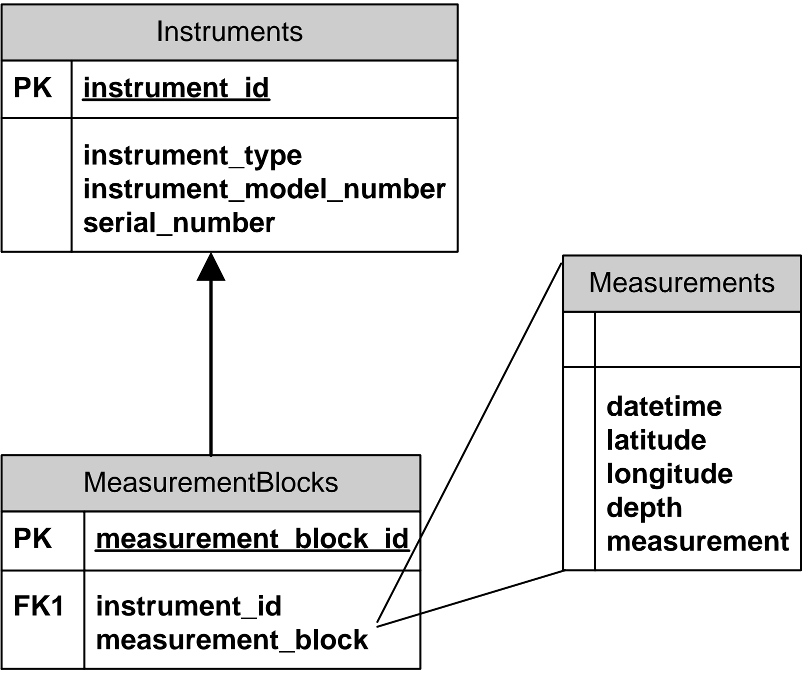


Figure 6: Instrument Measurements application schema.

Airline Flight

Schema

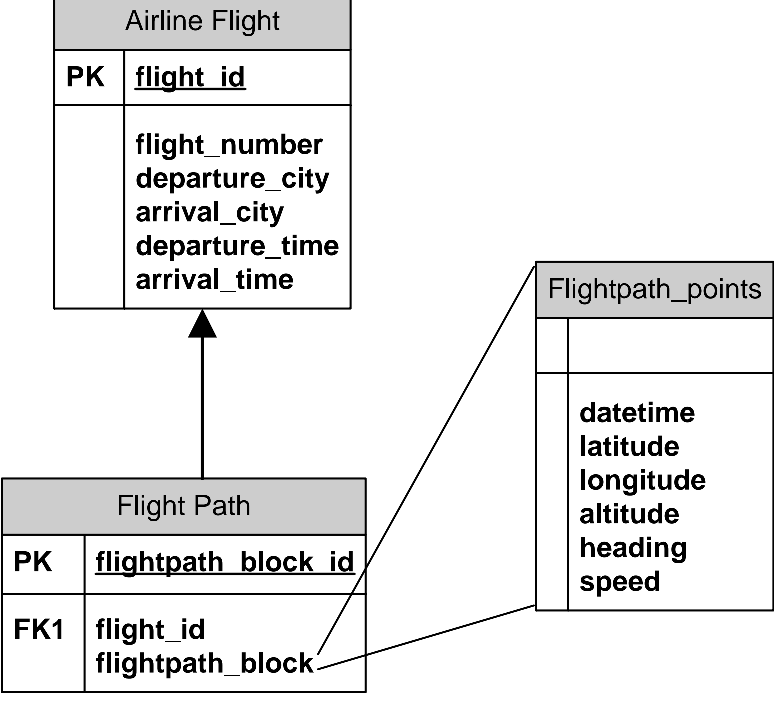


Figure 7: Airline Flight application schema.

GIS Object

Schema

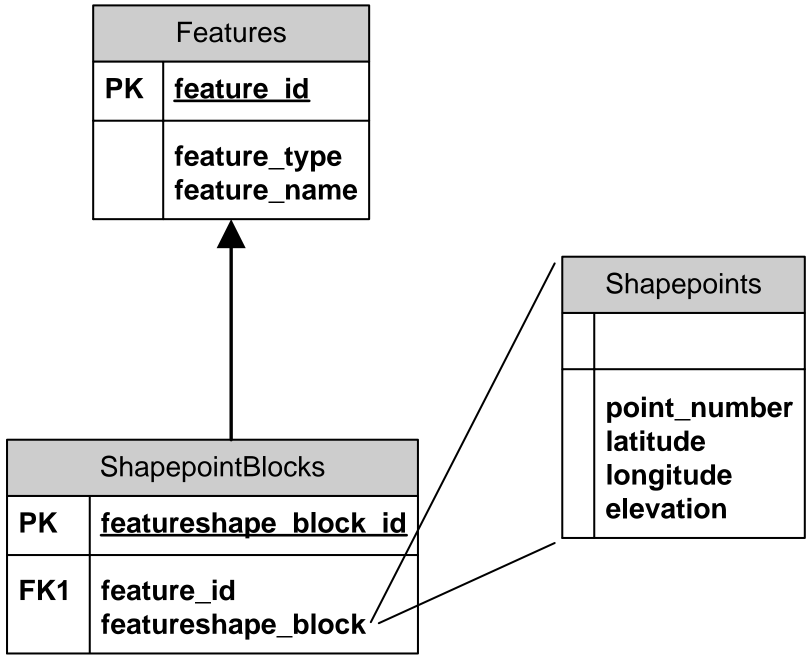


Figure 8: GIS Object application schema.

Stock Market

Schema

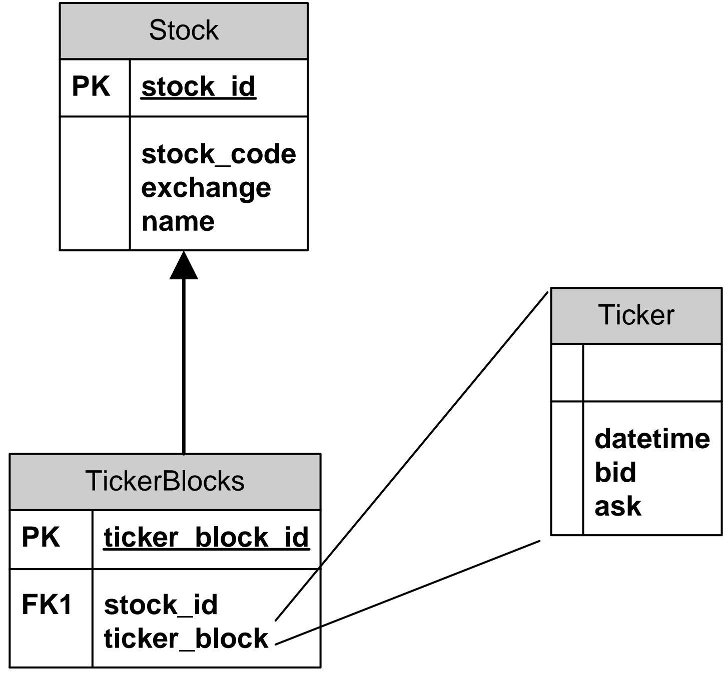


Figure 9: Stock Market application schema.

Delivery Tracking

Schema

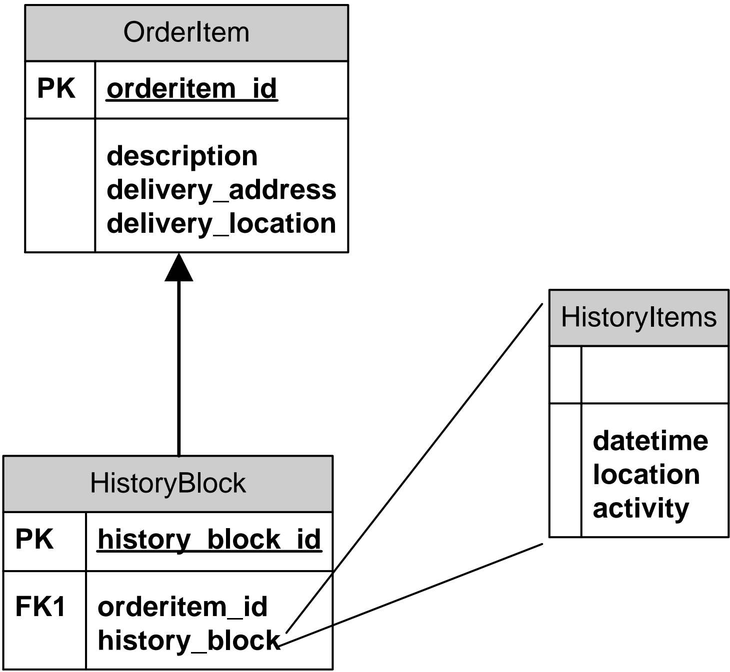


Figure 10: Delivery Tracking application schema.

Inventory

Schema

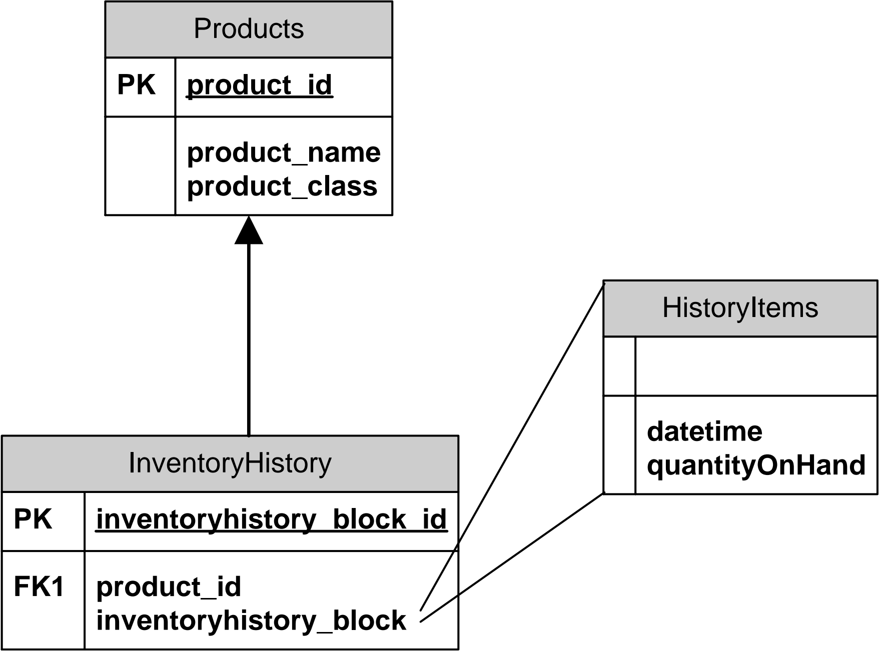


Figure 11: Inventory application schema.

Implementing Tuple Blocks – DBXten

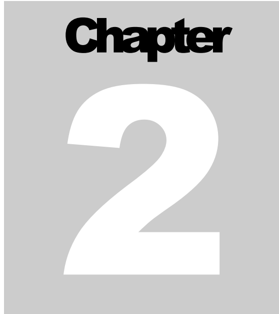
Logically a tuple block is nothing more than a table inside another table. We have described several applications where the natural way to model a portion of the application data is in a table-in-table fashion. There are many more. DBXten provides a natural, and efficient, way of implementing tables-in-tables.

Specifically, the DBXten database extension consists of:

1) a data type, DSChip, for storing a block of tuples.

2) utilities for loading tuple blocks from flat files or netCDF files.

3) API's for selecting one or more tuples from one or more blocks and returning them as a set of database rows.



Chapter 2: Installing the BCS DBXten Extension

This chapter describes how to install the BCS (Barrodale Computing Services) DBXten Extension software onto a Linux server machine and perform some simple operations to test the installation. This section assumes that you have a PostgreSQL installation already available. If you don't, you can contact BCS to receive a custom installation script that will install PostgreSQL for you.

Installing the Software

DBXten is packaged as a zip file, with a name of `DBXten_versionNumber.zip`[6]. The file can be unzipped and placed anywhere, but in the following we assume that it is being unzipped into the `/opt` directory.

1. cd into directory `/opt`

   ```
   $ cd /opt
   ```

2. Unzip the file.

   ```
   $ unzip DBXten_versionNumber.zip
   ```

3. cd into the DBXten directory.

   ```
   $ cd DBXten
   ```

4. Copy the shared object library to its proper spot (as root).

   ```
   $ cp lib/DataSeriesChip.so <POSTGRESQLDIR>/lib
   ```
   [7]

---

[6] Evaluation versions have names of the form DBXtenEval-*yyyy-mm-dd_versionNumber*.zip, where *yyyy-mm-dd* refers to the expiration date of the evaluation period.

[7] This may be called /usr/lib/pgsql on some systems. On Ubuntu systems running PostgreSQL 8.4 the directory is /usr/lib/postgresql/8.4/lib.

5. If the version being installed is not an evaluation version, then set up the license key as described in the next section (Setting up the License Key).

6. Create a PostgreSQL user "`demo`" which will own the example database and run the example programs. You may need to run this command as a privileged user such as "`postgres`". The following commands can be found in *<POSTGRESQLDIR>*`/bin`, which is assumed to be in `$PATH`.

   ```
   $ createuser -S -d -R demo -P
   ```

   Enter "demo" as the password.

   You may need to edit the `pg_hba.conf` file to ensure that the `demo` user can connect to PostgreSQL.

7. Create the `demo` database, owned by user `demo`

   ```
   $ createdb -O demo demo
   ```

8. Register the DBXten extension into the `demo` database[8].

   ```
   $ psql -e demo -U postgres <
            registration/DBXten.sql
   ```

9. Install the "Cube" Extension as described below in "Installing the "Cube" Extension".

10. "Make" the examples and test the installation as described below in "Building Sample Programs and Testing the Installation".

[8] This step will have to be repeated for each database into which you wish to install the DBXten extension.

Setting up the License Key

A license key is needed for each computer on which the PostgreSQL server runs[9]. The license key is provided to the BCS DBXten Extension by adding the following lines to the `.bashrc` file in the `postgres` account[10]:

```
export DBXTEN_LICENSE_KEY
DBXTEN_LICENSE_KEY=license_key_value
```

If there is a line in the file that reads

```
# User specific aliases and functions
```

place the license key statements right below that line.

Next, restart the PostgreSQL server. A simple way to do this is to log into the `root` account, cd to the `/etc/init.d` directory and execute the following commands:

```
./postgresql stop
./postgresql start
```

The license key is dependent on your machine's hostname and IP address. If either of these change, you will need to contact Barrodale Computing Services Ltd. for a new license key.

[9] A license key is not not needed for an evaluation copy.

[10] If the `postgres` account runs with some shell other than bash, then the means for setting the DBXTEN_LICENSE_KEY environment variable will be different. For example, if csh or tcsh is used, then "`setenv DBXTEN_LICENSE_KEY` *`license_key_value`*" will need to be placed in the `.cshrc` file in the `postgres` account.

Installing the "Cube" Extension

DBXten makes use of the open-source PostgreSQL "Cube" Extension to quickly and efficiently find tuple blocks that "overlap" a region of interest[11]. This extension includes a "cube" (n-dimensional box) data type and an operator class that allows a GiST[12] index to be built on a cube. You can install the cube extension in either of two ways:

1) Follow the instructions in the README.cube file located in the contrib/cube directory of the PostgreSQL installation package, or

2) Use the cube code packaged with DBXten by following these steps:

    a. Copy the `cube.so` shared object library to its proper spot.

    ```
    $ cd /opt/DBXten[13]
    $ cp lib/cube.so <POSTGRESQLDIR>/lib[14]
    ```

    b. Register the Cube extension into the `demo` database[15].

    ```
    $ $ psql -e demo -U postgres <
                registration/cube.sql
    ```

---

[11] By "region of interest" we don't necessarily mean something geospatial. "Region of interest" in this context is more general – e.g., "I'm interested in any tuples where column1 is between *A* and *B*, column2 is between *C* and *D*, etc."

[12] GiST stands for ***G***enera***li***zed ***S***earch ***T***ree. See http://en.wikipedia.org/wiki/GiST for an explanation of GiST indexes.

[13] "cd" to the same directory used in step 3 of the "Installing the Software" section.

[14] This directory may be called /usr/

[15] This step will have to be repeated for each database into which you wish to install the DBXten extension.

Building Sample Programs and Testing the Installation

Note that the sample programs assume that no password is needed to access the `demo` account on the demo database.

1. cd into the `examples/sql` directory.

```
$ cd <DBXTENDIR>/examples/sql/chapter2/sql
```

2. Create the `table_of_chips`, `instruments`, and `measurementBlocks` tables.

```
$ psql demo –U demo < table_of_chips.sql
$ psql demo –U demo < measurements.sql
```

3. Create and populate the `xyvals` table.

```
$ cd ..
$ ./populate400.sh
```

4. cd into the `examples/c` directory.

```
$ cd <DBXTENDIR>/examples/c
```

5. Build the C executable programs. Depending on the permissions for directory `<DBXTENDIR>/examples/c` you may have to "su" before running this command.

```
$ make
```

6. Execute the `loadTable` program to load the `measurementBlocks` table. First make sure that `<POSTGRESQLDIR>/lib` is included in `$LD_LIBRARY_PATH`.

```
$ ./loadTable
```

7. Execute the `fetch1` and `fetch2` programs to extract data from the `xyvals` table.

```
$ ./fetch1
$ ./fetch2
```

8. cd into the `examples/java` directory.

```
$ cd <DBXTENDIR>/examples/java
```

9. Build the class files.

```
$ make
```

10. Execute the loading class.

```
$ make runLoad
```

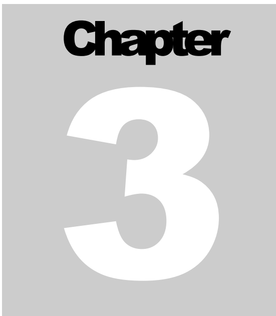
11. Execute the extracting class.

```
$ make runFetch
```

Determining the DBXten Software Version Number

The DSGetVersion function can be used to return the version number of the DBXten extension:

```
demo=> SELECT DSGetVersion();
 dsgetversion
--------------
 1.5.0.0
(1 row)

demo=>
```



Chapter 3: The BCS DBXten Database Extension – the DSChip Data Type

This chapter discusses the representation of tuple blocks inside the BCS DBXten Database Extension. In particular, it defines the DSChip data type, which is the data type used by DBXten to store a block of tuples.

The DSChip Data Type

In the Instrument Measurement example provided earlier, the tuples being stored had the following columns:

Column Name	Data Type	Precision
datetime	date and time	1.0 (i.e., precise to the whole second)
latitude	float	0.001
longitude	float	0.001
depth	float	0.02
measurement	float	0.1

If we were to store the tuples shown earlier into three DSChip instances (ignoring the … rows) and then do a SELECT from the table into which these instances were stored, we would get the output similar to the following[16]:

[16] This of course isn't very useful output. Generally DSChip contents are SELECTed in other ways, as described in Chapter 5. But this simple SELECT statement does illustrate the sort of information stored inside a DSChip.

```
> SELECT measurement_block FROM measurements;

measurement_block
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--------------------

maxtuples=7,filledtuples=7,numcolumns=5;datetime,date,1;latitude
,float,0.001;longitude,float,0.001;depth,float,0.01;measurement,
float,0.1;2008-02-01 00:12:23,46.343,-127.386,14.34,10.2;2008-
02-01 00: 12:25,46.344,-127.385,16.82,11.9;2008-02-01
00:12:27,46.345,-127 .383,18.85,10.7;2008-02-01
00:12:29,46.346,127.382,21.22,11.7;20 08-02-01 00:12:31,46.347,-
127.381,23.66,10.9;2008-02-01 00:12:33 ,46.349,-
127.379,26.05,11.4;2008-02-01 00:13:43,46.392,-127.335,
104.82,10.7

maxtuples=7,filledtuples=7,numcolumns=5;datetime,date,1;latitude
,float,0.001;longitude,float,0.001;depth,float,0.01;measurement,
float,0.1;2008-02-01 00:13:45,46.394,-127.334,106.83,10.7;2008-
02-01 00 :13:47,46.395,-127.332,108.90,13.1;2008-02-01
00:13:49,46.396,-1 27.331,110.99,10.7;2008-02-01
00:13:51,46.398,-127.330,113. 32,1 1.4;2008-02-01
00:13:53,46.399,-127.328,115.38,10.2;2008-02-0100
:13:55,46.400,-127.327,117.65,10.9;2008-02-01 00:14:59,46.439,-1
27.289,188.40,10.3

maxtuples=7,filledtuples=7,numcolumns=5;datetime,date,1;latitude
,float,0.001;longitude,float,0.001;depth,float,0.01;measurement,
float,0.1;2008-02-01 00:15:01,46.441,-127.287,190.88,9.7;2008-
02-01 00: 15:03,46.442,-127.286,193.22,11.9;2008-02-01
00:15:05,46.443,-12 7.285,195.65,11.5;2008-02-01
00:15:07,46.444,-127.283,197.86,10. 5;2008-02-01
00:15:09,46.446,-127.282,200.02,11.2;2008-02-01 00:
15:11,46.447,-127.281,202.38,10.7;2008-02-01 00:16:15,46.486,-12
7.241,274.68,11.3

(3 rows)
```

The text in the output above indicates the information that is stored inside a DSChip:

maxtuples=*value*,	This is the number of tuples that can be put in the DSChip
filledtuples=*value*,	This is the number of tuples currently in the DSChip
numcolumns=*value*;	This is the number of columns in a DSChip tuple.
columnName[="*unit*"**],columnType,precision;**	The DSChip column definitions – one for each of the ***numcolumns*** columns.
;**col1,col2**,…;	The tuples themselves – there are ***filledtuples*** of these.

Units Support

DBXten will allow a string to be stored with each numeric[17] column, the string denoting the units for values stored in the column. The intent is that the strings will obey the form of the UDUNITS-2 package from Unidata. These strings are not interpreted by DBXten; it is the user's responsibility to ensure that the data in them is meaningful. Here is how one of the `measurement_block` tuples shown above would appear if unit names were used:

```
maxtuples=7,filledtuples=7,numcolumns=5;datetime,date,1;latitude
="degrees_north",float,0.001;longitude="degrees_east",float,0.00
1;depth="meters",float,0.01;measurement="degC",float,0.1;2008-02
-01 00:12:23,46.343,-127.386,14.34,10.2;2008-02-01 00: 12:25, 46
.344,-127.385,16.82,11.9;2008-02-01 00:12:27,46.345,-127.383,1 8
.85,10.7;2008-02-01 00:12:29,46.346,127.382,21.22,11.7;20 08-02-
01 00:12:31,46.347,-127.381,23.66,10.9;2008-02-01 00:12:33,46.34
9,-127.379,26.05,11.4;2008-02-01 00:13:43,46.392,-127.335,104.82
,10.7
```

[17] Numeric columns are columns of type integer and float as described in the "Data Types that are Supported Inside a DSChip" section.

Data Types that are Supported Inside a DSChip

The data types that are available for columns within a DSChip are:

1) integer – 32 bit integer
2) int8 – 64 bit integer
3) float[18] – 64 bit floating point
4) string – 8 bit character strings
5) date – dates (in Greenwich Mean Time (GMT)) with an optional time component

The ***date*** type supports basically the same syntax as used by the PostgreSQL (see http://www.postgresql.org/docs/8.3/static/datatype-datetime.html). However, note that it is only able to support dates and times between Jan 1, 1902 and Dec 31, 2037.

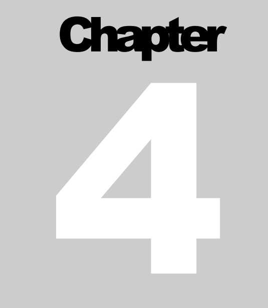
Date/Time Conversion Functions

For testing and data analysis purposes it is often useful to be able to convert between date/time values in string format and date/time values expressed as seconds since midnight, January 1 1970, GMT. The following SQL functions are provided with this in mind:

```
FUNCTION DSGMTTimeToDouble(char) RETURNS double precision

FUNCTION DSLocalTimeToDouble(char) RETURNS double precision

FUNCTION DSDoubleToGMTTime(double precision) RETURNS char

FUNCTION DSDoubleToLocalTime(double precision) RETURNS char
```

Similarly-defined Java (see page 103) and C (see page 97) client-side library functions are provided as well.

[18] "double" can be used as a synonym for float.



Chapter 4: Getting Data into DSChip's

There are four general mechanisms that can be used to insert data into DSChip's:

1) by using SQL,
2) by using one of the utility loaders supplied with DBXten,
3) by writing and running a C program written using the DBXten C API,
4) by writing and running a Java program written using the DBXten Java API, or
5) by using *Draw and Load* (DaL), the graphical data loading utility available free on the BCS Website.

Using the DBXten SQL API to Insert Data

This section describes the SQL functions that can be used to create or fill DSChip's. Most of these functions aren't used directly or invoked explicitly through SQL; rather they are involved either implicitly or from C or Java programs.

Creating an Empty DSChip

```
FUNCTION DSChipNew(schema CHAR) RETURNS DSChip
```

This function returns an empty (tuple-less) DSChip having a specified schema.

The format of the *schema* parameter is:

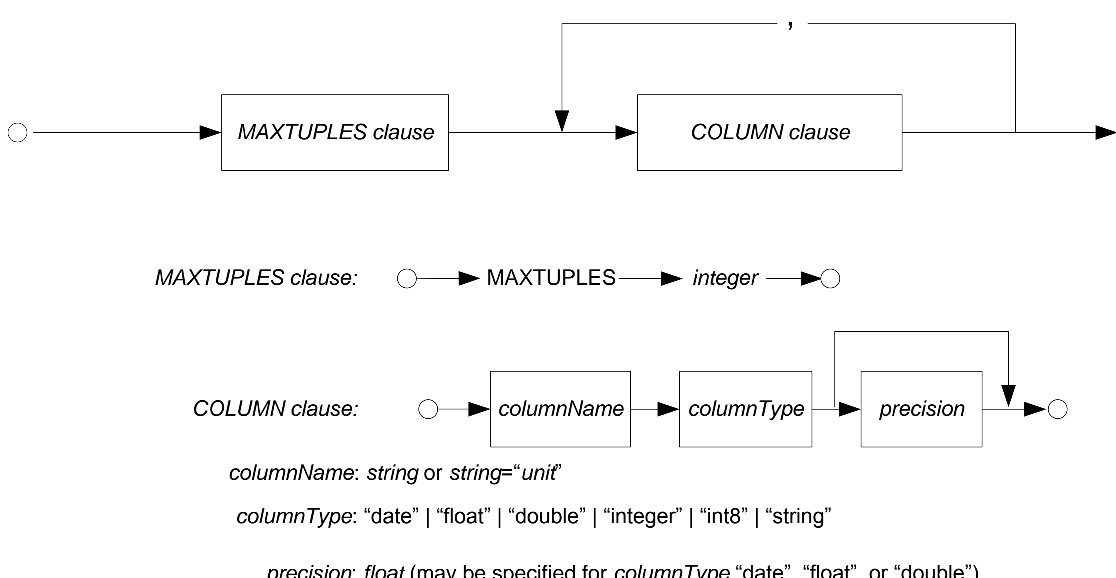
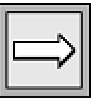


Figure 12: Syntax of a DSChip schema.



The precision value, which can optionally be specified for dates and floating point numbers (float/double) indicates how much precision should be maintained when storing the data. A precision value of "0", which is the default, indicates that data should be stored to full precision. A precision of "0.01", for example, indicates that the input data is only accurate to two decimal places and so the values that are later extracted from the DSChip are only guaranteed to agree with the input data values to the second decimal place. The example in the "Adding Tuples to a DSChip" section below illustrates this feature.

Example

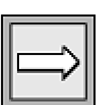
Run this, and all other sample sql files, as user "`demo`" in the "`demo`" database.

```
demo=> SELECT DSChipNew('maxtuples 200,datetime date 1,latitude
float 0.001,longitude="degrees_east" float 0.001,intColumn
integer');

                                                      dschipnew
-----------------------------------------------------------------
--------------------------------------------------
maxtuples=200,filledtuples=0,numcolumns=4;datetime,date,1;latitu
de,float,0.001;longitude="degrees_east",float,0.001;intColumn,in
teger,0
(1 row)

demo=> CREATE TABLE mytable(chipcol DSChip);
CREATE TABLE
demo=> INSERT INTO mytable SELECT
DSChipNew('maxtuples 200, datetime date 1,latitude float 0.001,
        longitude float 0.001,intColumn integer');
INSERT 0 1
demo=> SELECT * FROM mytable;
                                                       chipcol
-----------------------------------------------------------------
--------------------------------------------------
maxtuples=200,filledtuples=0,numcolumns=4;datetime,date,1;latitu
de,float,0.001;longitude,float,0.001;intColumn,integer,0
```

Adding Tuples to a DSChip

```
FUNCTION DSChipAppendRow(existingChip DSChip,
      valuesAsString CHAR) RETURNS DSChip
```

This function adds a new tuple to an existing DSChip, returning a new DSChip. The *valuesAsString* parameter is a comma-separated list of tuple values, with the columns specified in the same order as they were when specifying the schema when creating the DSChip.



Note that datetime values have the format

YYYY-MM-DD hh:mm:ss[.nnnnnn],

e.g., "2008-01-23 10:34:45", "2008-01-23 14:34:45", "2008-01-23 14:34:45.1", "2008-01-23 14:34:45.123456", etc.

Example

The following example uses SQL to load data into the `instruments` and `measurementBlocks` tables described earlier. This SQL can be found in the `examples/sql/measurements.sql` file in the distribution.

Create the parent table (`instruments`) and insert some rows into it. (This is just conventional, non-DBXten, SQL.)

```
demo=> CREATE TABLE instruments (
                  instrument_id INTEGER PRIMARY KEY,
                  instrument_type INTEGER,
                  instrument_model_number VARCHAR(30),
                  serial_number VARCHAR(30));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
      index "instruments_pkey" for table "instruments"
CREATE TABLE

demo=> INSERT INTO instruments
            VALUES(1,101,'ABC','SerialNum1');
INSERT 0 1

demo=> INSERT INTO instruments
            VALUES (2,102,'DEF','SerialNum2');
INSERT 0 1

demo=> INSERT INTO instruments
            VALUES (3,103,'GHI','SerialNum3');
INSERT 0 1

demo=> INSERT INTO instruments
            VALUES (4,104,'JKL','SerialNum4');
INSERT 0 1
```

Create the child table (`measurementBlocks`).

```
demo=>CREATE TABLE measurementBlocks (
            measurement_block_id SERIAL PRIMARY KEY,
            instrument_id INTEGER REFERENCES
                        instruments(instrument_id),
            measurement_block DSChip);
NOTICE:  CREATE TABLE will create implicit sequence
      "measurementblocks_measurement_block_id_seq" for
      serial column
      "measurementblocks.measurement_block_id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
      index "measurementblocks_pkey" for table
      "measurementblocks"
CREATE TABLE
```

Insert a row into the child table. This row contains a foreign key value pointing to the instruments table and an empty `measurement_block` DSChip value. Note that line feeds have been inserted into this text to break up the DSChipNew input value – if executing this command yourself, don't include the line feeds.

```
demo=> INSERT INTO measurementBlocks(
            instrument_id,
            measurement_block)
            SELECT 1,
                  DSChipNEW('maxtuples 100,datetime date
                        10,latitude float 0.001,longitude
                        float 0.001,intColumn integer');
INSERT 0 1
```

Insert tuples into the DSChip.

```
demo=> UPDATE measurementBlocks
            SET measurement_block =
                  DSChipAppendRow(measurement_block,
                  '2008-01-01 12:30:31,49.7,-127.5,45');
UPDATE 1

demo=> SELECT * FROM measurementBlocks;

measurement_block_id | instrument_id |
measurement_block
----------------------+----------------+------------------
-------------------------------------------------------------
-------------------------------------------------------------
------------------+--------------------
                     1 |             1 |
maxtuples=2,filledtuples=1,numcolumns=4;datetime,date,10;l
atitude,float,0.001;longitude,float,0.001;intColumn,intege
r,0;2008-01-01 12:30:30,49.700,-127.500,45 |
(1 row)

demo=> UPDATE measurementBlocks
            SET measurement_block =
                  DSChipAppendRow(measurement_block,
                  '2008-01-01 12:30:35,49.51234,-
                  127.7238,24');
UPDATE 1

demo=> SELECT * FROM measurementBlocks;

measurement_block_id | instrument_id |
measurement_block
----------------------+----------------+------------------
-------------------------------------------------------------
-------------------------------------------------------------
------------------+--------------------
                     1 |             1 |
maxtuples=2,filledtuples=2,numcolumns=4;datetime,date,10;l
atitude,float,0.001;longitude,float,0.001;intColumn,intege
r,0;2008-01-01 12:30:30,49.700,-127.500,45;2008-01-01
12:30:40,49.512,-127.724,24 |
(1 row)
```

Note in this example how the specified precision of 0.001 for latitude and longitude result in the truncation of 49.51234 and -127.7238 to 49.512 and -127.724, respectively. Similarly, the precision of "10" for datetime resulted in the conversion of the seconds components from 31 and 36 to 30 and 40, respectively.

Example

The following example uses SQL to add and fill another DSChip, but it does this using a single SQL statement. (Note again that line feeds have been added to improve readability).

```
demo=> INSERT INTO measurementBlocks(
                         measurement_block_id,
                         instrument_id,
                         measurement_block)
           SELECT 2,1,
             DSChipAppendRow(
               DSChipAppendRow(
                 DSChipNew('maxtuples 2,datetime date 10,
                               latitude float 0.001,
                               longitude float 0.001,
                               intColumn integer'),
                   '2008-01-01 12:30:31,49.7,-127.5,45'
               ),
                   '2008-01-01 12:30:35,
                     49.51234,-127.7238,24');
INSERT 0 1
```

Indexing DSChip's

Indexes can be created on a DSChip by first casting the DSChip to a cube data type (which was described above on page 22) and then indexing the cube with a GiST index.

There are two functions that can be used to cast a DSChip into a cube:

```
FUNCTION DSAsCubeString (existingChip DSChip,
      columnSpec CHAR) RETURNS TEXT
```

This function returns the text form of an *n*-dimensional cube built on the *n* columns listed in the *columnSpec* parameter. The columnSpec parameter has the structure shown in Figure 12 (page 30).

```
FUNCTION DSAsGeoCubeString (existingChip DSChip,
      columnSpec CHAR) RETURNS TEXT
```

This function is the same as `DSAsCubeString` except that any columns named "lat", "latitude", "lon", or "longitude" are first converted to a Cartesian coordinate of a point on the unit circle. This prevents issues with coordinate wrap-around.

GiST indexes can be built using SQL similar to the following:

```
demo=> CREATE INDEX measurementBlock_idx ON
           measurementBlocks
       USING
            gist((DSAsCubeString(measurement_block,
                  'datetime,latitude,longitude')::cube));
CREATE INDEX
```

Note that the choice of columns to include in the `columns` parameter depends on the sorts of queries that are envisioned. The goal of using the GiST index is to eliminate as many DSChip's as possible from consideration so that only those DSChip's that actually contain tuples of interest need to be unpacked and examined further. There is no point in including a column in the index if most of the DSChip's have values for that column that are in the region of interest for most queries. Consider the following diagram where we have just two columns, X and Y, in each tuple (we are limiting the tuples to two columns simply for ease of illustration; the argument generalizes to any number of columns). Each box in the diagram represents a DSChip and the boundaries of the boxes represent the extent of X and Y values that are contained in the respective DSChip.

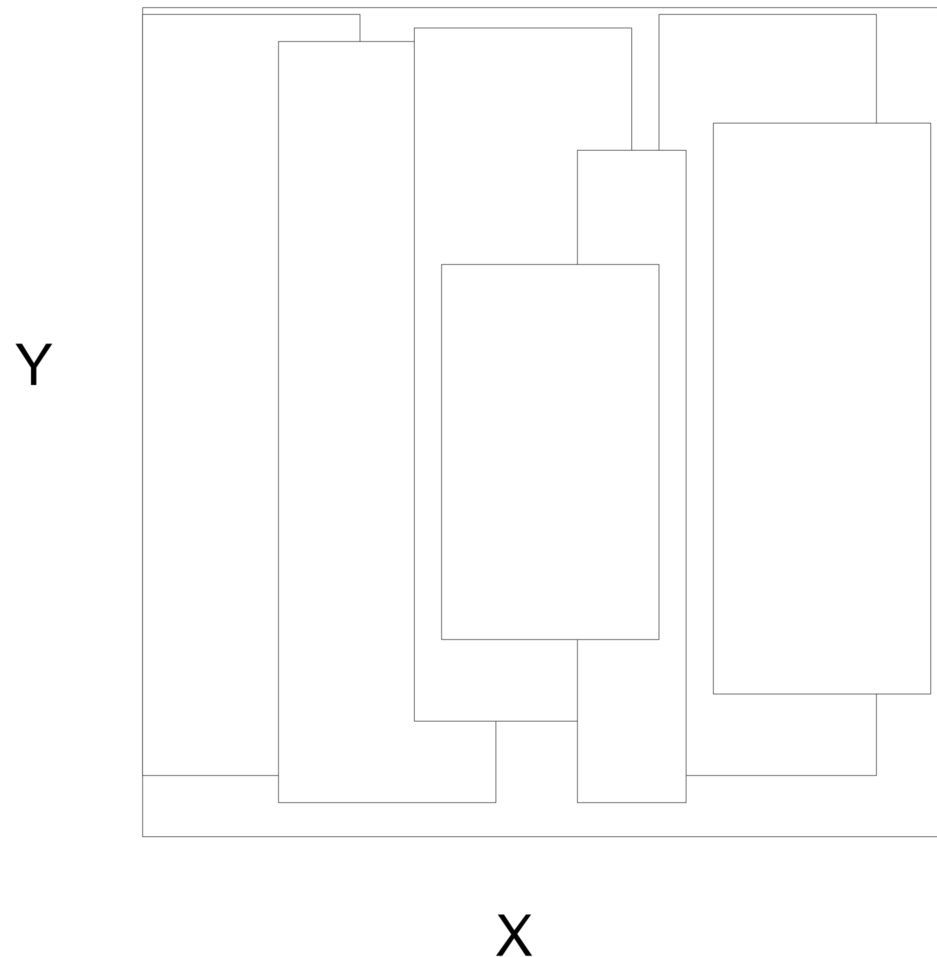


Figure 13: Two-dimensional DSChip's.

In this example, "Y" is not a good candidate for a GiST index column, since for almost any small range of Y values almost all of the DSChip's will contain tuples in that range. On the other hand, "X" is a good candidate since for every small range of X values only a relatively small number of DSChip's will contain values in that range.

Other Functions

```
FUNCTION DSChip_recv(valueAsBytes bytea) RETURNS DSChip
```

This function is used to create a DSChip from binary data. It is used by Java and C programs; its use is illustrated in the examples in the sections "A Sample C DSChip Loading Program" (page 39) and "A Sample Java DSChip Loading Program" (page 45).

```
FUNCTION DSChip_in(valueAsString cstring) RETURNS DSChip
```

This function is used to create a DSChip from its textual ascii representation. It is used by C programs.

```
FUNCTION DSChip_in(valueAsString char) RETURNS DSChip
```

This is a variant of DSChip_in used to create a DSChip from its textual Java-compatible ASCII representation. It is used by Java programs.

Using a Utility Loader to Insert Data

DBXten is shipped with two pre-built applications for loading DSChip's from files.

The CSV File Reader utility can be used to load a flat, delimited, text file, i.e., a text file with one tuple per line, using a constant delimiter[19] to separate the columns. This utility is described in Appendix E on page 105.

If the input data is in a netCDF file, then the netCDF File Reader utility can be used. This utility is described in Appendix F on page 110.

[19] The name ***csv***Loader implies that the delimiter is a comma, but this is just the default; it can be any single character.

Using the DBXten C API to Insert Data

If the input data is not already in a form that one of the prebuilt utility loaders handles, then the C or Java API's can be used to write a custom loading application. This section describes the C API; the Java API is described in the next section (page 45).

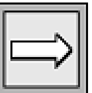
General Structure of Loading Programs

The general mechanism for loading data into DSChip's is the following:

1) A table-like C object representing a DSChip is created.
2) A schema for the object is defined by adding column information to the C object.
3) Individual data values are stored in the object.
4) The object is converted to its compressed-binary form.
5) The compressed binary form is stored in a table as a DSChip object.

A Sample C DSChip Loading Program

This section will guide you through a sample C program for loading data into DSChip's. This program can be found in `examples/c/loadTable.c` in the DBXten distribution.



Before running this program, run the SQL provided in the example on page 31. This will create and populate the tables used by the example.

First you will need to include PostgreSQL's libPQ library definitions, and the definitions for DBXten.

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>  /* Postgresql  libPQ definitions */
#include <dschip_exports.h>  /* definitions for DBXten */
```

Later on, we'll want to create and use a prepared statement to insert our DSChip's. The STMT_TAG string will be the name of our prepared statement.

```
#define STMT_TAG  "STMT_1"
```

The section below defines the metadata for the DSChip's that we will be loading. In a more sophisticated program, this information would likely be generated dynamically.

```
/* column metadata */
#define NUM_CHIP_COLUMNS (4)
static char *columnNames  [NUM_CHIP_COLUMNS] =
      {"Datetime", "Latitude", "Longitude", "intColumn"};
static char *unitNames [NUM_CHIP_COLUMNS] = {
      NULL,
      "degrees_north",
      "degrees_east",
      NULL
};
static double precisions [NUM_CHIP_COLUMNS] =
      { 10, 0.001, 0.001, 0 };
static int columnTypes [NUM_CHIP_COLUMNS] =
      { DSVarTypeDATE, DSVarTypeDOUBLE, DSVarTypeDOUBLE,
            DSVarTypeINT };
```

The next section is a static repository of the data to be stored in the DSChip's. In a more sophisticated program, this data would be read from an input data file of some type instead of being statically defined in the program.

We are building input data for a DSChip that will initially have 2 (`NUM_CHIP_ROWS`) tuples but which might grow to 200 (`MAX_ROWS_PER_CHIP`) tuples.

```
/* actual data */
#define MAX_ROWS_PER_CHIP  (200)
#define NUM_CHIP_ROWS  (2)
static char *datetimes[NUM_CHIP_ROWS] =
          {"2008-01-01 12:30:31", "2008-01-01 12:30:35"};
static double latitudes [NUM_CHIP_ROWS] = {49.7, 49.51234};
static double longitudes [NUM_CHIP_ROWS] = {-127.5, -127.7238};
static double intvals [NUM_CHIP_ROWS] = {45, 24};
```

Note in the above that the date values are initially stored as character strings. They will be converted into the DBXten date type later.

The next section builds a DSChip with a schema determined by the variables in the metadata section. The argument to DSChipCreate must be greater than or equal to the number of rows that actually get stored in the DSChip. If it is greater, the DSChip will consume more space in memory than it strictly needs to, but it won't increase the amount of space the DSChip takes when actually stored in the database itself. As mentioned above, we are allowing for a capacity of 200 (`MAX_ROWS_PER_CHIP`) tuples although initially we are inserting just 2 (`NUM_CHIP_COLUMNS`).

```
DSChip *BuildChip() {
     DSChip *chip;
     int i;

     chip = DSChipCreate(MAX_ROWS_PER_CHIP);
     for( i = 0; i <; i++ ) {
```

```
        DSChipAddVarWithUnit(chip, columnNames[i],
           unitNames[i], columnTypes[i], precisions[i]);
    }
    return chip;
}
```

The `PopulateChip` function populates a DSChip with a set of discrete values taken from some variables. The use of DSChipClearRows() is not strictly needed in this application as the DSChip is initially empty, but it would be necessary if `PopulateChip` was to be called a second time.

```
void PopulateChip(DSChip *chip)
{
    DSChipVar *columns[NUM_CHIP_COLUMNS];
    int i;

    DSChipClearRows(chip);

    for( i = 0; i < NUM_CHIP_COLUMNS; i++ ) {
        columns[i] = DSChipGetVarByPos(chip, i);
    }

    for( i = 0; i < NUM_CHIP_ROWS; i++ ) {
        DSChipVarSetDouble(columns[0], i,
                 DSGMTStringToDouble(datetimes[i]));
        DSChipVarSetDouble(columns[1], i, latitudes[i]);
        DSChipVarSetDouble(columns[2], i, longitudes[i]);
        DSChipVarSetInt(columns[3], i, intvals[i]);
    }
}
```

At the beginning of the "for" loop above we convert the specified date/time values to the fractional-seconds-since-1970 representation used internally for dates/times in DBXten[20].

The `CheckResult` function checks the result of a libPQ call to determine if it failed. In a real application, this would be more elaborate, to give a more precise indication of which statement actually failed.

```
void CheckResult( PGconn *conn, PGresult *result) {
   if( result == NULL ) {
      fprintf(stderr,"null result: %s\n",
         PQerrorMessage(conn));
         exit(-1);
   }
   else if( PQresultStatus(result) != PGRES_COMMAND_OK) {
       fprintf(stderr, "operation failed: %s",
                PQresultErrorMessage(result));
```

[20] The fractional-seconds-since-1970 representation is more compact than a textual representation, and it allows for better compression of series of values.

```
        exit(-1);
    }
}
```

The `PrepareToInsert` function creates a prepared statement that will later be used to insert the DSChip (along with the two other columns in the measurementBlocks table). Prepared statements tend to run faster than directly executed statements because the statement doesn't need to be reparsed by the server each time. The `DSChip_recv` function is a DBXten SQL registered function that converts an array of bytes into a `DSChip`.

```
void PrepareToInsert(PGconn *conn, char *table_name,
                     int measurement_block_id,
                     int instrument_id,
                     char *column_names)
{
      char sqlText[256];
      int nParams = 1;
      sprintf(sqlText,
        "insert into %s(instrument_id,%s)"
        "values(%d,DSChip_recv($1::bytea))",
            table_name, column_names, instrument_id);
      CheckResult(conn, PQprepare(conn,  STMT_TAG,
                  sqlText, nParams, NULL));
}
```

Note that in the prepared statement that is built, the instrument_id value is constant. This is appropriate since all the tuples that we will be inserting have the same instrument_id. In a more sophisticated program you may wish to change the sprintf statement above to

```
      sprintf(sqlText,
        "insert into %s(instrument_id,%s)"
        "values($1,DSChip_recv($2::bytea))",
      table_name, column_names);
```

If this were done then other changes to the program would have to be made as well, particularly in the `StoreChipInTable` function shown below and in the main program, also shown below. The modified program can be found in `examples/c/loadTable_modified.c` in the DBXten distribution.

The function below converts a DSChip into a contiguous array of bytes and then loads the bytes into the desired table using the earlier created prepared statement.

```
#define NUM_TABLE_COLUMNS (1)

void StoreChipInTable(DSChip *chip, PGconn *conn) {
    int chipLength;
    void *bytes;
```

```
    char *column_data[NUM_TABLE_COLUMNS];
    int lengths[NUM_TABLE_COLUMNS];
    int isBinary[NUM_TABLE_COLUMNS];
    int resultFormatIsBinary = 1;

    lengths[0] =  DSChipToByteLength(chip);
    column_data[0] = (char *)DSMalloc(lengths[0]);
    DSChipToBytes(chip, column_data[0]);

    isBinary[0] = 1;
    CheckResult(conn, PQexecPrepared(conn, STMT_TAG,
            NUM_TABLE_COLUMNS,
         (const char *const *)column_data,
                lengths, isBinary, resultFormatIsBinary));
    DSFree(column_data[0]);
}
```

The next section establishes a connection to the PostgreSQL server in function `CreateConnection`, and disposes of it in function `CloseConnection`.

```
PGconn *CreateConnection()
{
    PGconn *conn;
    char *connectionString =
            "host=localhost dbname=demo user=demo";

    conn = PQconnectdb(connectionString);
    if( PQstatus(conn) == CONNECTION_BAD ) {
        fprintf(stderr, "unable to connect to database\n");
        exit(-1);
    }
    return conn;
}

void CloseConnection(PGconn *conn)
{
    PQfinish(conn);
}
```

Finally, here is the main driver program. The program:

1) creates a database connection,
2) builds a prepared statement to insert a row,
3) constructs a DSChip,
4) executes the prepared statement to load the DSChip and other columns into a database table row,
5) closes the database connection.

```
int main(int argc, char *argv) {
    PGconn *conn;
    DSChip *chip;
    int measurement_block_id=1;
    int instrument_id=1;
```

```
    conn = CreateConnection();
    PrepareToInsert(conn, "measurementBlocks",
           measurement_block_id, instrument_id,
            "measurement_block");
    chip = BuildChip();
    PopulateChip(chip);
    StoreChipInTable(chip, conn);
    CloseConnection(conn);
}
```

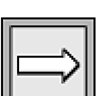
Using the DBXten Java API to Insert Data

This section will guide you through a sample Java program for loading data into DSChip's. This program can be found in `examples/java/LoadTable.java` in the DBXten distribution. It can be run by issuing the command "`make runLoad`" in that directory.

A Sample Java DSChip Loading Program

The following Java LoadTable program has the same high level structure as the C version listed earlier (page 39) with the following exceptions:

- the prepared statement is referenced by an object rather than a name, and
- there is no need for a CheckResults function because errors are handled by Java's exception mechanism.



Before running this program, run the SQL provided in the example on page 31. This will create and populate the tables used by the example.

```
import java.sql.*;
import com.barrodale.dschip.*;

public class LoadTable {
```

The standard way to connect to a database with JDBC is by using the DriverManager.getConnection method. Using this method allows an application the ability to connect to a database without knowing what type of database it is. However, the method only knows about drivers that have already been loaded by the JRE (Java runtime engine), therefore the PostgreSQL driver class is explicitly loaded by the program.

```
    final String serverName = "carbon";
    final String databaseName = "demo";
    final String userName = "postgres";
    final String password = "";


    private Connection createConnection() throws SQLException
    {
        String url = "jdbc:postgresql://" + serverName + "/" +
                   databaseName;
        try {
            Class.forName("org.postgresql.Driver");
        } catch( Exception e1) {
            throw new RuntimeException(e1.toString());
        }
        return
            DriverManager.getConnection(url,userName,password);
```

```
}

private PreparedStatement prepareToInsert(
    Connection conn,
    String table_name,
    String column_name,
    int instrument_id) throws SQLException
{
    return conn.prepareCall("INSERT INTO " + table_name +
            "(instrument_id," + column_name +
            ") values(" + instrument_id + "," +
            "DSChip_recv(?))");
}
```

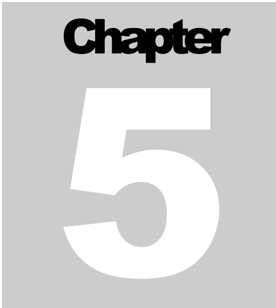
Note that in this method, we are using specialized calls for each type of column. These are just convenience methods that in turn call a single public method.

```
private DSChip buildChip() {
    DSChip chip = new DSChip(200);
    chip.addDateColumn("Datetime",10);
    chip.addDoubleColumn("Latitude",”degrees_north”,0.001);
    chip.addDoubleColumn("Longitude",”degrees_east”,0.001);
    chip.addIntColumn("intColumns");
    return chip;
}

private final Timestamp [] datetimes = new Timestamp [] {
  Timestamp.valueOf("2008-01-01 12:30:31"),
  Timestamp.valueOf("2008-01-01 12:30:35")
};

private final double [] latitudes = new double [] {
   49.7, 49.51234
};

private final double [] longitudes = new double [] {
   -127.5, -127.7238
};

private final int [] intvals = new int [] {
   45, 24
};
```

In the C program, the code makes a DSChipVar type to access individual elements of a DSChip. In the Java version, the elements are accessed through the DSChip, requiring the use of a column index argument.

```
private void populateChip(DSChip chip) {
    chip.clearRows();
    for(int i = 0; i < datetimes.length;  i++ ) {
        chip.setDate(i, 0, datetimes[i]);
        chip.setDouble(i, 1, latitudes[i]);
```

```
            chip.setDouble(i, 2, longitudes[i]);
            chip.setInt(i, 3, intvals[i]);
        }
    }


    private void storeChipInTable(DSChip chip,
                    PreparedStatement ps)
      throws java.sql.SQLException
    {
        byte [] bytes = chip.toBytes();
        ps.setBytes(1, bytes);
        ps.execute();
    }

    public LoadTable(int instrument_id) {
        try {
           DSChip chip =  buildChip();
           Connection connection = createConnection();
           PreparedStatement ps = prepareToInsert(connection,
                  "measurementBlocks",
                  "measurement_block",
                  instrument_id);
           populateChip(chip);
           storeChipInTable(chip, ps);
           ps.close();
           connection.close();
        } catch( java.sql.SQLException e1 ) {
            System.out.println(e1.toString());
        }
    }

   public static void main(String args[]) {
      int instrument_id = 1;
      new LoadTable(instrument_id);
    }
}
```



Chapter 5: Retrieving Data from DSChip's

There are three general mechanisms that can be used to extract data from DSChip's:

1) by using SQL,

2) by writing and running a C program written using the DBXten C API, or

3) by writing and running a Java program written using the DBXten Java API.

Each of these mechanisms is described in the following sections. Many of the examples in these sections use a table called "`xyvals`", which contains a single column of type DSChip. The schema for this DSChip contains three columns: `X` (integer), `Y` (integer), and `Z` (float). The script `populate400.sh` in the directory `<DBXTENDIR>/examples` can be used generate data for, and load, the `xyvals` table.

Options for Extracting Data

The following diagram illustrates the possible processing paths involved in retrieving data using DBXten.

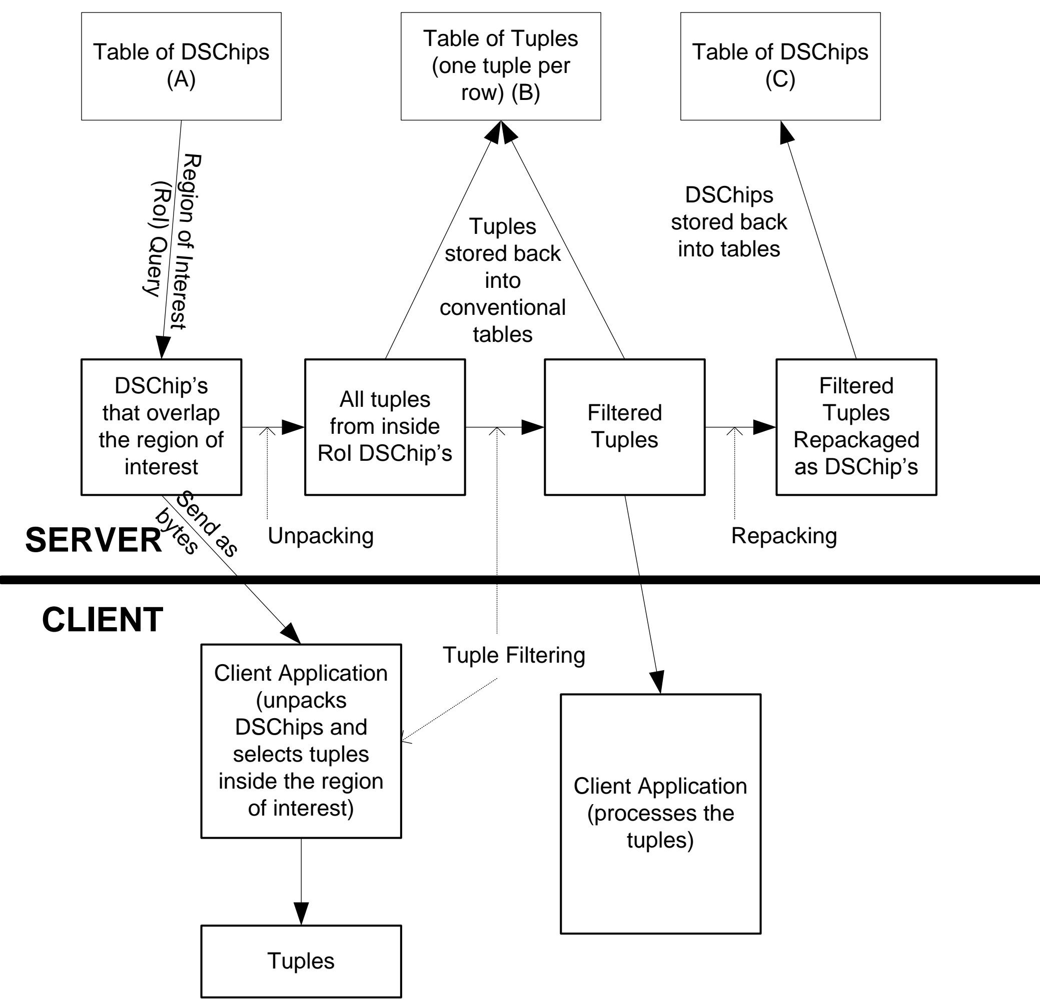


Figure 14: DSChip Extraction Processing paths.

The first step in extracting data from DSChip's is to determine which DSChip's "overlap" the "region of interest." This is shown as the "Region of Interest" query in the diagram above where DSChip's are selected from Table A.

Region of Interest Query explained

Suppose that the DSChip's have three integer columns x, y, and z and that we are interested in all tuples where x = xval, y is between ymin and ymax, and z can be anything. Then our region of interest can be defined as:

- x is between x_{val} and x_{val}
- y is between y_{min} and y_{max}
- z is between -4and +4

Similarly each DSChip has minimum and maximum values:

- x is between $Chip_{Xmin}$ and $Chip_{Xmax}$
- y is between $Chip_{Ymin}$ and $Chip_{Ymax}$
- z is between $Chip_{Zmin}$ and $Chip_{Zmax}$

So as a first step we restrict ourselves to the DSChip's where

- $Chip_{Xmax} \geq x_{val}$ and $Chip_{Xmin} \leq x_{val}$, and
- $Chip_{Ymax} \geq y_{min}$ and $Chip_{Ymin} \leq y_{max}$

In the following diagram, all three DSChip's satisfy the test on the Y dimension, but only DSChip's A and B satisfy the test on the X dimension, so it is these two DSChip's that are selected.

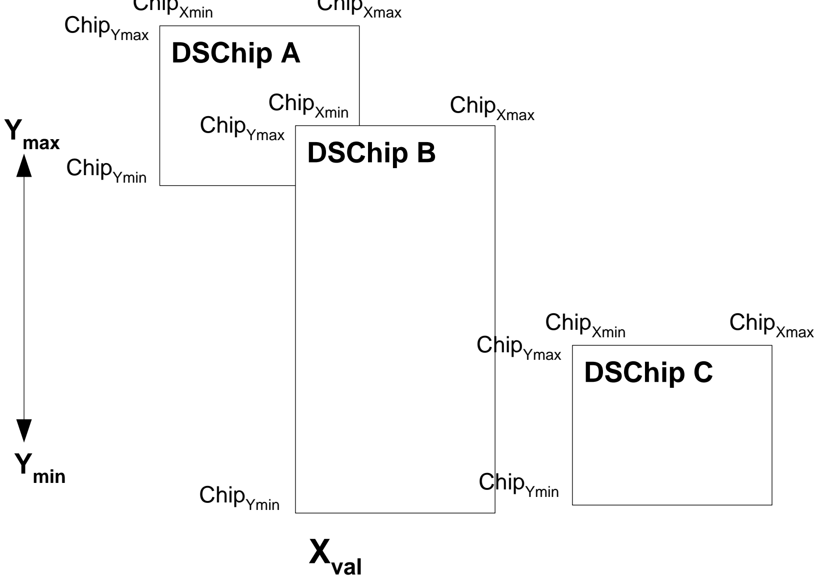


Figure 15: DSChip's A and B overlap the specified X and Y ranges.

"Tuple Filtering" explained

The second step is to unbundle the selected DSChip's and select from those DSChip's just the tuples that satisfy the query. This is referred to as "Tuple Filtering" in the diagram above. Suppose for example that in the example above that X_{val} is 100.0, Y_{min} is 200.0 and Y_{max} is 250.0. Suppose further that DSChip A has (x,y) tuples:

- TA1 = (100.0, 240.0)
- TA2 = (100.0, 245.0)
- TA3 = (100.0, 255.0)
- TA4 = (120.0, 225.0)

and that DSChip B has tuples:

- TB1 = (100.0, 251.0)
- TB2 = (100.0,199.0)

Then, tuples TA1, TA2, and TA3 in DSChip A satisfy the X and Y query constraints, but no tuple in DSChip B does.

As shown in Figure 14 (page 50), tuple filtering can be done on either the server or the client. If network bandwidth is an issue or if the tuples are needed on the server (e.g., to be inserted into another table as illustrated in Figure 14) then the tuple filtering should be done on the server. Otherwise, tuple filtering on the client is preferred since it places less of a load on the database server and allows greater opportunities for parallelism. The examples that follow will illustrate both approaches.

Other Processing Paths

Figure 14 (page 50) shows a number of other processing paths in addition to the Region of Interest Query and Tuple Filtering:

- Once the Region of Interest query has been performed, for example, we may wish to just extract from the DSChip's (and not perform any further tuple filtering). This is shown by the line labeled "Unpacking" in the Figure.

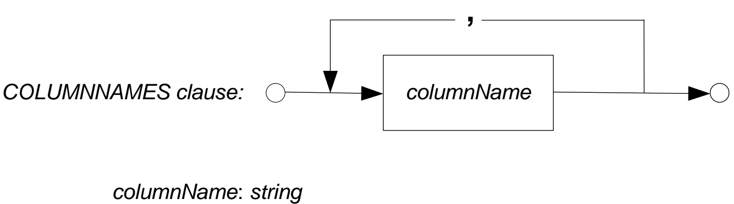
- Once we have filtered the tuples, we may wish to repack just the filtered tuples back into new DSChip's. This is shown by the line labeled "Repacking" in the Figure.

- Once we have tuples or DSChip's we may wish to load them back into other tables in the database (such as Tables B or C in the Figure).

Using the DBXten SQL API to Extract Data

The function signatures listed below refer to character string parameters called `columnNames` and `rangeSpec`.

The `columnNames` parameter has the following syntax:

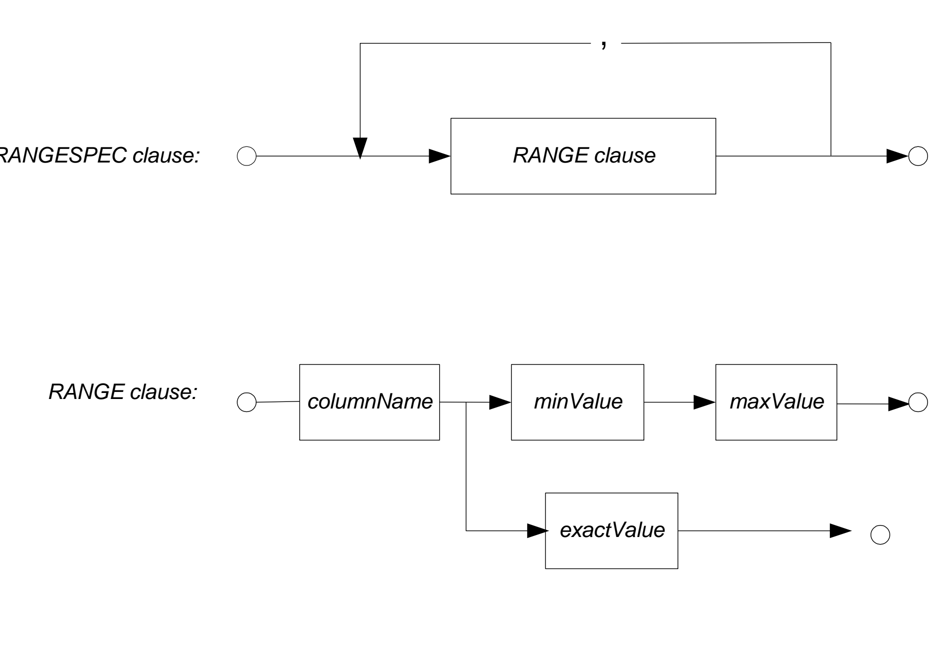


columnName: *string*

Figure 16: Syntax of a `columnNames` parameter.

Note that an empty string (‘’) can be used as shorthand for specifying all the columns in a DSChip.

The `rangeSpec` parameter has this syntax:



minValue: *some value from the domain of the column, or ‘*’ to match anything*

maxValue: *some value from the domain of the column, or ‘*’ to match anything*

exactValue: *some value from the domain of the column*

Figure 17: Syntax of a `rangeSpec` parameter.

Determining What Columns are in a DSChip

FUNCTION DSChipSchema(*existingChip* DSChip) RETURNS char

The following example uses SQL to list the schema for the "`chip`" DSChip column in the "`xyvals`" table. In this example we are assuming that every DSChip in this column has the same schema (that need not be the case, though[21]), so we use a "`LIMIT 1`" clause to limit the number of rows returned to one.

Example

```
demo=> SELECT DSChipSchema(chip) FROM xyvals LIMIT 1;

                   dschipschema
--------------------------------------------------
 maxtuples 25, x integer, y integer, val float 0.01
(1 row)
```

Example

```
demo=> SELECT DSChipSchema(measurement_block) FROM
            measurementBlocks limit 1;
                                        dschipschema
----------------------------------------------------------------
----------------------------
 maxtuples 2, datetime date 10, latitude float 0.001, longitude
float 0.001, intColumn integer
(1 row)
```

Listing Values for DSChip Column(s)

FUNCTION DSChipToStrings(*existingChip* DSChip, *columnNames* char) RETURNS setof char

Example

```
demo=> SELECT
measurement_block_id,DSChipToStrings(measurement_block,
      'latitude,longitude') FROM measurementBlocks
      WHERE measurement_block_id = 1;

 measurement_block_id | dschiptostrings
----------------------+----------------
                    1 | 49.512,-127.500
                    1 | 49.700,-127.724
 (2 rows)
```

Listing Distinct Values for DSChip Column(s)

FUNCTION DSDistinct(*existingChip* DSChip, *columnNames* char) RETURNS setof char

Example

[21] See the discussion on page 11 in the section "Features of Tuples that Can Be Exploited – What Tuple Features Make Tuple Block Storage Particularly Suitable?"

```
demo=> SELECT
measurement_block_id,DSDistinct(measurement_block,'latitude')
FROM measurementBlocks;

 measurement_block_id | dsdistinct
----------------------+------------
                    1 | 49.512
                    1 | 49.700
                    2 | 49.512
                    2 | 49.700
(4 rows)
```

Determining Whether a DSChip has a Particular Column(s)

FUNCTION DSHasVariables(*existingChip* DSChip, *columnNames* char) RETURNS boolean

Example

```
demo=> SELECT measurement_block_id,
      DSHasVariables(measurement_block,'latitude,longitude')
      FROM measurementBlocks;

 measurement_block_id | dshasvariables
----------------------+----------------
                    1 | t
                    2 | t
(2 rows)

demo=> SELECT measurement_block_id,
      DSHasVariables(measurement_block,'latitude,x')
      FROM measurementBlocks;

 measurement_block_id | dshasvariables
----------------------+----------------
                    1 | f
                    2 | f
(2 rows)
```

Determining How Many Columns a DSChip has

FUNCTION DSNumVars(*existingChip* DSChip) RETURNS integer

Example

```
demo=> SELECT measurement_block_id,
                  DSNumVars(measurement_block)
            FROM measurementBlocks;

 measurement_block_id | dsnumvars
----------------------+-----------
                    1 |         4
                    2 |         4
(2 rows)
```

Determining the Number of Tuples in a DSChip

FUNCTION DSNumFilledRows(*existingChip* DSChip) RETURNS integer

Example

```
demo=> SELECT measurement_block_id,
                  DSNumFilledRows(measurement_block)
            FROM measurementBlocks;

 measurement_block_id | dsnumfilledrows
----------------------+-----------------
                    1 |               2
                    2 |               2
(2 rows)
```

Determining the Maximum Value for Columns in a DSChip

```
FUNCTION DSMaxAsString(existingChip DSChip, columnNames char)
RETURNS TEXT
```

Example

```
demo=> SELECT DSMaxAsString(measurement_block,
      'latitude,longitude') FROM measurementBlocks;

  dsmaxasstring
-----------------
 49.700,-127.500
 49.700,-127.500
```

Determining the Minimum Value for Columns in a DSChip

```
FUNCTION DSMinAsString(existingChip DSChip, columnNames char)
RETURNS TEXT
```

Example

```
demo=> SELECT DSMinAsString(measurement_block,
      'latitude,longitude') FROM measurementBlocks;

  dsminasstring
-----------------
 49.512,-127.724
 49.512,-127.724
```

Getting Column Values from Single-Tuple DSChip's

The functions described in this section can be used to extract individual columns from a single-tuple DSChip. Single-tuple DSChip's are created by the DSChipToRowChip and DSChipExtractToRowChip functions described later.

```
FUNCTION DSGetInteger(chip DSChip, columnPosition integer)
RETURNS Integer
```

This function returns a 32-bit-integer-typed column from the first tuple of a DSChip. It will generate an error message if the column is not integer-typed. Column positions start at 1.

```
FUNCTION DSGetInt8(chip DSChip, columnPosition integer)  RETURNS
bigint
```

This function returns a 64-bit-integer-typed column from the first tuple of a DSChip. It will generate an error message if the column is not integer-typed. Column positions start at 1.

```
FUNCTION DSGetDouble(chip DSChip, columnPosition integer)
RETURNS Double Precision
```

This function returns a double-typed column from the first tuple of a DSChip. It will generate an error message if the column is not double-typed. Column positions start at 1.

```
FUNCTION DSGetText(chip DSChip, columnPosition integer)
RETURNS TEXT
```

This function returns a string-typed column from the first tuple of a DSChip. It will generate an error message if the column is not string-typed. Column positions start at 1.

```
FUNCTION DSGetDate(chip DSChip, columnPosition integer)
RETURNS Timestamp
```

This function returns a Datetime-typed column from the first tuple of a DSChip. It will generate an error message if the column is not Datetime-typed. Column positions start at 1.

Listing Compression Information for a DSChip

FUNCTION DSCompressInfo(*existingChip* DSChip) RETURNS TEXT

Example

```
demo=> SELECT measurement_block_id,
                DSCompressInfo(measurement_block)
            FROM measurementBlocks;

 measurement_block_id |
dscompressinfo
----------------------+----------------------------------------
-----------------------------------------------------------------
------------
                    1 | Total 158  |  datetime arithmetic 17 |
latitude arithmetic 17 |  longitude arithmetic 17 |  intColumn
arithmetic 9
                    2 | Total 158  |  datetime arithmetic 17 |
latitude arithmetic 17 |  longitude arithmetic 17 |  intColumn
arithmetic 9
(2 rows)
```

The output format of DSCompressInfo is a string of the form:

```
Total size_of_entire_DSChip_in_bytes ( | column_name
compression_type column_size_in_bytes)*
```

Note that the sum of the *column_size_in_bytes* is less than the *size_of_entire_DSchip_in_bytes* because the *column_size_in_bytes* doesn't include meta-data such as precisions and the column name.

Converting a DSChip Range to a Cube

```
FUNCTION DSRangeToCube(rangeSpec char) RETURNS text
```

Example

```
demo=> SELECT DSRangeToCube('datetime 1970-01-01 00:00:00 1970-
01-01 00:00:01, latitude 49.0 49.01');

                   dsrangetocube
---------------------------------------------------
 (28800.000000,49.000000),(28801.000000,49.010000)
(1 row)
```

Generating a Bounding Cube from a DSChip

```
FUNCTION DSAsCubeString(existingChip DSChip, columnNames char)
RETURNS text
```

Example

```
demo=> SELECT measurement_block_id,
                  DSAsCubeString(measurement_block,
                              'datetime,latitude')
            FROM measurementBlocks;

 measurement_block_id |                        dsascubestring
----------------------+----------------------------------------
--------------------
                    1 |
(1199219425.000000,49.511500),(1199219445.000000,49.700500)
                    2 |
(1199219425.000000,49.511500),(1199219445.000000,49.700500)
(2 rows)
```

Doing a Region of Interest Query to Select DSChip's

As discussed earlier (page 51), a Region of Interest query is used to select those DSChip's that *may* contain tuples of interest; it doesn't actually do any unpacking of DSChip's or selection of any specific tuples from a DSChip. A Region of Interest query typically makes use of the following DBXten features:

1) a cube-valued functional index built on the DSChip columns of interest (possibly in addition to other columns that are not of interest), and

2) some cube, or set of cubes, to be used in comparison, and

3) the PostgreSQL "`&&`" operator.

In the following examples we assume that an index has been built on the `x` and `y` columns of the `chip` DSChip in table `xyvals` using function `DSAsCubeString(chip,'x,y'::bpchar)`.

Example

In this first example we wish to find all DSChip's that overlap the box defined by `x` being between 26 and 39 and `y` being between 31 and 53:

```
CREATE INDEX xyvals_idx ON xyvals
      USING gist((DSAsCubeString(chip,'x,y')::cube));
SELECT chip FROM xyvals
   WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
            && DSRangeToCube('x 26 39,y 31 53')::cube;
```

Example

In this example we wish to find all DSChip's that overlap the line defined by `y` being between 31 and 53. We don't care about `x` but we include it in our test (using wildcards) so that the index is used:

```
SELECT chip FROM xyvals
   WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
            && DSRangeToCube('x * *,y 31 53')::cube;
```

Example

We could have also got the same results with the following query, but it wouldn't have used the index:

```
SELECT chip FROM xyvals
   WHERE DSAsCubeString(chip,'y'::bpchar)::cube
            && DSRangeToCube('y 31 53')::cube;
```

Example

The cube that we are comparing to doesn't have to be a single cube and it doesn't have to be a literal. In the next example we compare our DSChip's to a set of cubes stored in a different table.

```
SELECT chip FROM xyvals, referenceCubes
   WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
            && referenceCubes.cube and
                  referenceCubes.otherAttibute = otherValue;
```

Example

The last example selects DSChip's from the `measurementBlocks` table. This example shows how to use an index that includes a date column. Note that the date/times in the datetime range string are assumed to be in Greenwich Mean Time.

```
CREATE INDEX measurementBlocks_cube_idx ON measurementBlocks
      USING gist((dsascubestring(measurement_block,
            'datetime,latitude,longitude'::bpchar)::cube));
SELECT measurement_block_id, measurement_block
      FROM measurementBlocks
   WHERE DSAsCubeString(measurement_block,
            'datetime,latitude,longitude'::bpchar)::cube
            && DSRangeToCube('datetime 2008-01-01 12:00:00
2008-01-01 14:00:00,latitude * *,longitude * *')::cube;
```

Extracting Tuples from a DSChip (no tuple filtering)

```
FUNCTION DSChipToRowChip(existingChip DSChip) RETURNS
      setof DSChip
```

The examples above can all be modified to return a set of 1-tuple DSChip's instead of the fuller, as-stored DSChip's simply by replacing the DSChip column ("`chip`" or "`measurement_block`") with "`DSChipToRowChip(DSChipCol)`". The benefit of doing this is that the individual tuple columns can then be isolated, using the DSGetInteger, DSGetDouble, etc. functions.

Example

```
demo=> SELECT DSGetInteger(onetuple,1) AS x,
       DSGetInteger(onetuple,2) AS y,
       DSGetDouble(onetuple,4) AS val FROM
   (SELECT DSChipToRowChip(chip) AS onetuple FROM xyvals
       WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
                && DSRangeToCube('x 26, y 50 52')::cube) AS
                        tupleset ;

 x  | y  | val
----+----+------
 20 | 50 | 0.89
 20 | 51 | 0.18
 20 | 52 | 0.79
 20 | 53 |  0.2
 20 | 54 | 0.05
 20 | 55 | 0.31
.
.
.
 28 | 57 | 0.35
 28 | 58 | 0.22
 28 | 59 | 0.64
 29 | 50 | 0.98
 29 | 51 | 0.17
 29 | 52 | 0.91
 29 | 53 | 0.39
 29 | 54 | 0.22
 29 | 55 | 0.58
 29 | 56 | 0.62
 29 | 57 | 0.38
 29 | 58 | 0.55
 29 | 59 | 0.45
(500 rows)
```

Note that the query above returns all the tuples from all of the DSChip's that *overlap* the region on interest; many of these tuples do not themselves fit inside the region of interest. To eliminate those outliers, extra WHERE clause components could be added, as illustrated by the following example.

Example

```
demo=> SELECT DSGetInteger(onetuple,1) AS x,
             DSGetInteger(onetuple,2) AS y,
             DSGetDouble(onetuple,4) AS val FROM
       (SELECT DSChipToRowChip(chip) AS onetuple FROM xyvals
             WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
                   && DSRangeToCube('x 26, y 50 52')::cube)
             AS tupleset
                WHERE DSGetInteger(onetuple,1) = 26 and
                   DSGetInteger(onetuple,2) between 50 and 52;

 x  | y  | val
----+----+------
 26 | 50 |  0.5
 26 | 51 | 0.58
 26 | 52 | 0.15
 26 | 50 | 0.66
 26 | 51 | 0.49
 26 | 52 | 0.32
 26 | 50 | 0.09
 26 | 51 | 0.85
 26 | 52 | 0.86
 26 | 50 | 0.65
 26 | 51 | 0.78
 26 | 52 |  0.8
 26 | 50 | 0.65
 26 | 51 | 0.61
 26 | 52 | 0.39

(15 rows)
```

A more efficient way of performing this tuple filtering will be shown later in the section Tuple Filtering DSChip's into a Set of Tuples (page 64).

Counting Matches without Extracting

```
FUNCTION DSChipNumMatches(existingChip DSChip, rangeSpec char)
RETURNS integer
```

The following example counts the number of contained tuples falling inside the region of interest for each of the DSChip's that overlap the region of interest. The total number of tuples in each DSChip is shown as well.

Example

```
demo=> SELECT DSNumFilledRows(chip),
                DSChipNumMatches(chip,'x 26, y 50 52')
            FROM xyvals
            WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
                && DSRangeToCube('x 26, y 50 52')::cube;


 dsnumfilledrows | dschipnummatches
-----------------+------------------
             100 |                3
             100 |                3
             100 |                3
             100 |                3
             100 |                3
(5 rows)
```

This example shows results consistent with those of the previous two examples. A total of 5 DSChip's, containing a total of 500 tuples, overlap the region of interest. Three tuples in each DSChip fall inside the region of interest, for a total of 15 tuples.

Tuple Filtering DSChip's into New DSChip's

Tuples that have been extracted from DSChip's through tuple filtering can be repacked into new DSChip's using the following function:

```
FUNCTION DSChipExtractToChip(existingChip DSChip, rangeSpec
            char, columnNames char) RETURNS DSChip
```

This function always creates one DSChip for every DSChip that it takes as input. The new DSChip's may have fewer tuples than the corresponding input DSChip (in fact in general they will) and they may have no tuples at all. Which tuples from the input DSChip go into the output DSChip is controlled by the rangeSpec. The columnNames parameter can then be used to determine which columns go into the output DSChip.

Example

In the first example there are three DSChip's that satisfy the WHERE clause, so the query produces three new DSChip's. The columnNames parameter requests that the new DSChip's have just two columns (X and Y) instead of three (X, Y, and vals).

```
demo=> SELECT DSChipExtractToChip(chip,'x 26, y 40 52','x y')
            FROM xyvals
                  WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
                        && DSRangeToCube('x 26, y 40 52')::cube;
                                   dschipextracttochip
-----------------------------------------------------------------
-------------------------
maxtuples=25,filledtuples=5,numcolumns=2;x,integer,0;y,integer,0
;26,40;
26,41;26,42;26,43;26,44

maxtuples=25,filledtuples=5,numcolumns=2;x,integer,0;y,integer,0
;26,45;
26,46;26,47;26,48;26,49

maxtuples=25,filledtuples=3,numcolumns=2;x,integer,0;y,integer,0
;26,50;
26,51;26,52
(3 rows)
```

Example

In the second example there are again three DSChip's that satisfy the WHERE clause, so the query produces three new DSChip's. The columnNames parameter requests that the new DSChip's have the same three columns as the input, but the rangeSpec parameter results in no tuples being put into the new DSChip's.

```
demo=> SELECT DSChipExtractToChip(chip,'x 99','x y val')
            FROM xyvals
            WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
                && DSRangeToCube('x 26, y 40 52')::cube;
                           dschipextracttochip
-----------------------------------------------------------------
----------

maxtuples=25,filledtuples=0,numcolumns=3;x,integer,0;y,integer,0
;val,fl
oat,0.01

maxtuples=25,filledtuples=0,numcolumns=3;x,integer,0;y,integer,0
;val,fl
oat,0.01

maxtuples=25,filledtuples=0,numcolumns=3;x,integer,0;y,integer,0
;val,fl
oat,0.01
(3 rows)
```

Tuple Filtering DSChip's into a Stream of Bytes

Tuples that have been extracted from DSChip's through tuple filtering can be sent to a client using the following function:

```
FUNCTION DSChipExtractToBytes(existingChip DSChip, rangeSpec
char, columnNames char) RETURNS bytea
```

This function returns one byte array for every DSChip that it takes as input. It is normally called from a Java or C client program as illustrated in the example in Performing Tuple Filtering on the Server (which starts on page 69).

Tuple Filtering DSChip's into a Set of Tuples

The following function is like DSChipToRowChip described earlier (page 60) in that it extracts tuples from DSChip's. However it also does tuple filtering (through a rangeSpec parameter) and is able to return just a subset of the tuple columns (as specified in the columnNames parameter).

```
FUNCTION DSChipExtractToRowChip(existingChip DSChip,
rangeSpec char, columnNames char) RETURNS setof DSChip
```

Example

Compare the output of the following example with the output produced by the DSChipToSet function in the example above (page 60).

```
demo=> SELECT DSGetInteger(onetuple,1) AS x,
            DSGetInteger(onetuple,2) AS y,
            DSGetDouble(onetuple,4) AS val FROM
            (SELECT DSChipExtractToRowChip(chip,
                  'x 26, y 40 52','x y z val') AS onetuple
                  FROM xyvals
                  WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
                        && DSRangeToCube('x 26, y 50 52')::cube)
                              AS tupleset ;
x  | y  | val
----+----+------
 26 | 50 |  0.5
 26 | 51 | 0.58
 26 | 52 | 0.15
 26 | 50 | 0.66
 26 | 51 | 0.49
 26 | 52 | 0.32
 26 | 50 | 0.09
 26 | 51 | 0.85
 26 | 52 | 0.86
 26 | 50 | 0.65
 26 | 51 | 0.78
 26 | 52 |  0.8
 26 | 50 | 0.65
 26 | 51 | 0.61
 26 | 52 | 0.39
(15 rows)
```

Note that since we are not retrieving the z column, we could have also written the query as follows (the bolded sections identify the two changes):

```
demo=> SELECT DSGetInteger(onetuple,1) AS x,
            DSGetInteger(onetuple,2) AS y,
            DSGetDouble(onetuple,3) AS val FROM
            (SELECT DSChipExtractToRowChip(chip,
                  'x 26, y 40 52','x y val') AS onetuple
                  FROM xyvals
                  WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube
                        && DSRangeToCube('x 26, y 50 52')::cube)
                              AS tupleset ;
```

Using the DBXten C API to Extract Data

This section will guide you through two sample C programs for fetching data from DSChip's. These program can be found in the `examples/c/` directory in the DBXten distribution.

Performing Tuple Filtering on the Client

The first program (fetch1.c) performs tuple filtering on the client. Its basic structure is as follows:

1) Build a query to do a Region of Interest query to select DSChip's.
2) Send the query to the PostgreSQL database server.
3) For each DSChip in the query results:
    a. Deserialize/decompress the DSChip
    b. for each row (tuple) in the DSChip
        i. process the row (including doing tuple filtering)

First you will need to include PostgreSQL's libPQ library definitions, and the definitions for DBXten.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libpq-fe.h>  /* Postgresql  libPQ definitions */
#include <dschip_exports.h>  /* definitions for DBXten */
```

Later on, we'll want to create and use a prepared statement to insert our DSChip's. The STMT_TAG string will be the name of our prepared statement.

```
#define STMT_TAG  "STMT_1"
```

The next section establishes a connection to the PostgreSQL server in function `CreateConnection`, and disposes of it in function `CloseConnection`.

```
PGconn *CreateConnection()
{
    PGconn *conn;
    char *connectionString =
            "host=localhost dbname=demo user=demo";

    conn = PQconnectdb(connectionString);
    if( PQstatus(conn) == CONNECTION_BAD ) {
```

```
        fprintf(stderr, "unable to connect to database\n");
        exit(-1);
    }
    return conn;
}

void CloseConnection(PGconn *conn)
{
    PQfinish(conn);
}
```

The `CheckResult` function has to accept values of `PGRES_TUPLES_OK` as well as `PGRES_COMMAND_OK`.

```
static void CheckResult( PGconn *conn, PGresult *result)
{
   int status;
   if( result == NULL ) {
      fprintf(stderr,"null result: %s\n",
         PQerrorMessage(conn));
         exit(-1);
   }
   else {
       status = PQresultStatus(result);
       if( status != PGRES_TUPLES_OK && status !=
                   PGRES_COMMAND_OK) {
          fprintf(stderr, "operation failed: %s",
                   PQresultErrorMessage(result));
          exit(-1);
       }
   }
}
```

The next function builds a prepared statement. For a program that performs a single fetch, it is not necessary to use a prepared statement, but it is useful in the case of repeated queries.

```
static void PrepareToFetch(PGconn *conn, char *table_name,
                              char *column_name)
{
    char sqlText[256];
    sprintf(sqlText, "SELECT %s FROM %s WHERE "
                     "DSAsCubeString(chip,'x,y'::bpchar)::cube"
                     "&& DSRangeToCube($1)::cube",
                column_name, table_name);
    CheckResult(conn,
                   PQprepare(conn, STMT_TAG, sqlText, 1, NULL));
}
```

The following function generates the textual representation of the cube value that the program uses as a Region of Interest key.

```
static void GenerateKeyText(char *queryText, double xRange[],
```

```
                              double yRange[])
{
    sprintf(queryText, "x %f %f, y %f %f",
    xRange[0], xRange[1], yRange[0], yRange[1]);
}
```

The next function extracts values from a single DSChip. It performs its own filtering of rows (tuple filtering) using the xRange and yRange parameters.

```
static void ProcessASingleChip(DSChip *chip, double xRange[],
 double yRange[])
{
   int chipRows;
   int i;
   DSChipVar *xVar, *yVar, *valVar;

   chipRows = DSChipGetNumRows(chip);
   xVar = DSChipGetVarByName(chip, "x");
   yVar = DSChipGetVarByName(chip, "y");
   valVar = DSChipGetVarByName(chip, "val");
   for( i = 0; i < chipRows; i++ ) {
      double x, y;
       x = DSChipVarGetDouble(xVar, i);
       y = DSChipVarGetDouble(yVar, i);
/* perform tuple filtering */
       if( x >= xRange[0] && x <= xRange[1] &&
           y >= yRange[0] && y <= yRange[1] ) {
          printf("(%f,%f) = %f\n", x, y,
                 DSChipVarGetDouble(valVar, i));
       }
   }
}
```

The following function performs the actual query. Notice that the final argument to PQexecPrepared is a 1, indicating that the fetched values should come back in binary form. Only use PQexec* functions that allow you to specify the form of the returned values; functions that don't will return values converted to text.

```
#define NUM_TABLE_COLUMNS (1)

static void PerformQuery(PGconn *conn, char *keyText)
{
    int chipLength;
    void *bytes;
    char *key_data[NUM_TABLE_COLUMNS];
    int lengths[NUM_TABLE_COLUMNS];
    int isBinary[NUM_TABLE_COLUMNS];
    int nResults;
    int i;
    PGresult *res;

    lengths[0] =  strlen(keyText);
```

```
    key_data[0] = keyText;
    isBinary[0] = 0;

    res = PQexecPrepared(conn, STMT_TAG, 1,
         (const char *const *)key_data,
         lengths, isBinary, 1);
    CheckResult(conn, res);
    nResults = PQntuples(res);
    for( i = 0; i < nResults; i++ ) {
        void *chipData;
        int chipDataLen;
        DSChip *chip;
        chipData = PQgetvalue(res, i, 0);
        chipDataLen = PQgetlength(res, i, 0);
        chip = DSChipFromBytes(chipData, chipDataLen);
        ProcessASingleChip(chip);
    }
    PQclear(res);
}
```

Finally, here is the main program, which specifies the Region of Interest (X between 26 and 39; Y between 31 and 53).

```
static double xRange[] = { 26, 39};
static double yRange[] = { 31, 53};

int main(int argc, char *argv) {
    PGconn *conn;
    char queryText[256];
    DSChip *chip;

    conn = CreateConnection();

    PrepareToFetch(conn, "xyvals", "chip");
    GenerateKeyText(queryText, xRange, yRange);
    PerformQuery(conn, queryText);
    CloseConnection(conn);
    return 0;
}
```

Performing Tuple Filtering on the Server

The second program (fetch2.c) performs tuple filtering on the server. Its basic structure is as follows:

1) Build a query that does both a Region of Interest selection as well as tuple filtering.

2) Send the query to the PostgreSQL database server.

First you will need to include PostgreSQL's libPQ library definitions, and the definitions for DBXten.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libpq-fe.h>
#include <dschip_exports.h>
```

Later on, we'll want to create and use a prepared statement to insert our DSChip's. The STMT_TAG string will be the name of our prepared statement.

```
#define STMT_TAG  "STMT_1"

#define NUM_ARGUMENTS (1)
```

The next section establishes a connection to the PostgreSQL server in function `CreateConnection`, and disposes of it in function `CloseConnection`.

```
static PGconn *CreateConnection()
{
    PGconn *conn;
    char *connectionString =
            "host=localhost dbname=demo user=demo";

    conn = PQconnectdb(connectionString);
    if( PQstatus(conn) == CONNECTION_BAD ) {
        fprintf(stderr, "unable to connect to database\n");
        exit(-1);
    }
    return conn;
}

static void CloseConnection(PGconn *conn)
{
    PQfinish(conn);
}
```

The `CheckResult` function has to accept values of `PGRES_TUPLES_OK` as well as `PGRES_COMMAND_OK`.

```
static void CheckResult( PGconn *conn, PGresult *result)
{
   int status;
   if( result == NULL ) {
      fprintf(stderr,"null result: %s\n",
         PQerrorMessage(conn));
         exit(-1);
   }
   else {
       status = PQresultStatus(result);
       if( status != PGRES_TUPLES_OK && status !=
                   PGRES_COMMAND_OK) {
          fprintf(stderr, "operation failed: %s",
                   PQresultErrorMessage(result));
          exit(-1);
       }
```

```
   }
}
```

The next function builds a prepared statement. For a program that performs a single fetch, it is not necessary to use a prepared statement, but it is useful in the case of repeated queries. Note the difference between the sqlText value used here and the one in the fetch1.c example.

```
static void PrepareToFetch(PGconn *conn, char *table_name, char
*column_name)
{
    char sqlText[256];

     sprintf(sqlText,
          "SELECT DSChipExtractToBytes(%s, $1,'x,y,val')"
          " FROM %s WHERE "
              "DSAsCubeString(chip, 'x,y'::bpchar)::cube "
              "&& DSRangeToCube($1)::cube",
                column_name, table_name);
    CheckResult(conn,
            PQprepare(conn,  STMT_TAG, sqlText, 1, NULL));
}
```

The following function generates the textual representation of the cube value that the program uses as a Region of Interest key.

```
static void GenerateKeyText(char *queryText, double xRange[],
    double yRange[])
{
    sprintf(queryText, "x %f %f, y %f %f",
    xRange[0], xRange[1], yRange[0], yRange[1]);
}
```

The next function extracts value from a single DSChip. Unlike the fetch1.c example shown previously, it *does not* perform its own filtering of rows (tuple filtering) (having left tuple filtering to the `DSChipExtractToBytes` function).

```
static void ProcessASingleChip(DSChip *chip)
{
   int chipRows;
   int i;
   DSChipVar *xVar, *yVar, *valVar;

   chipRows = DSChipGetNumRows(chip);
   xVar = DSChipGetVarByName(chip, "x");
   yVar = DSChipGetVarByName(chip, "y");
   valVar = DSChipGetVarByName(chip, "val");
   for( i = 0; i < chipRows; i++ ) {
       double x, y, z;
       x = DSChipVarGetDouble(xVar, i);
       y = DSChipVarGetDouble(yVar, i);
       z = DSChipVarGetDouble(valVar, i);
       printf("(%f,%f) = %f\n", x, y, z);
```

```
    }
}
```

The following function performs the actual query. Notice that the final argument to PQexecPrepared is a 1, indicating that the fetched values should come back in binary form. Only use PQexec* functions that allow you to specify the form of the returned values; functions that don't will return values converted to text.

```
static void PerformQuery(PGconn *conn, char *keyText, double
xRange[],
    double yRange[])
{
    int chipLength;
    void *bytes;
    char *key_data[NUM_ARGUMENTS];
    int lengths[NUM_ARGUMENTS];
    int isBinary[NUM_ARGUMENTS];
    int nResults;
    int i;
    PGresult *res;

    lengths[0] =  strlen(keyText);
    key_data[0] = keyText;
    isBinary[0] = 0;

    lengths[0] =  strlen(keyText);
    key_data[0] = keyText;
    isBinary[0] = 0;

    res = PQexecPrepared(conn, STMT_TAG, NUM_ARGUMENTS,
         (const char *const *)key_data,
         lengths, isBinary, 1);
    CheckResult(conn, res);
    nResults = PQntuples(res);
    for( i = 0; i < nResults; i++ ) {
        void *chipData;
        int chipDataLen;
        DSChip *chip;
        chipData = PQgetvalue(res, i, 0);
        chipDataLen = PQgetlength(res, i, 0);
        chip = DSChipFromBytes(chipData, chipDataLen);
        ProcessASingleChip(chip);
    }
    PQclear(res);
}

static double xRange[] = { 26, 39};
static double yRange[] = { 31, 53};
```

Finally, here is the main program, which specifies the Region of Interest (X between 26 and 39; Y between 31 and 53).

```
int main(int argc, char *argv) {
    PGconn *conn;
    char queryText[256];
    DSChip *chip;

    conn = CreateConnection();

    PrepareToFetch(conn, "xyvals", "chip");
    GenerateKeyText(queryText, xRange, yRange);
    PerformQuery(conn, queryText, xRange, yRange);
    CloseConnection(conn);
    return 0;
}
```

Using the DBXten Java API to Extract Data

This class is the Java version of the fetch1.c program listed earlier.

```
/*
 * Fetch.java
 *
 * Created on Feb 1, 2008, 9:33:15 AM
 *
 */

import java.sql.*;
import com.barrodale.dschip.*;

public class Fetch {

    final String serverName = "carbon";
    final String databaseName = "demo";
    final String userName = "demo";
    final String password = "";

    private Connection createConnection() throws SQLException
    {
        String url = "jdbc:postgresql://" + serverName + "/" +
                    databaseName;
        try {
            Class.forName("org.postgresql.Driver");
        }catch( Exception e1) {
            throw new RuntimeException(e1.toString());
        }
        return DriverManager.getConnection(url, userName,
                          password);
    }
```

The prepareToFetch method builds a prepared statement. It calls the dschip_send method, which is used in order to prevent the DSChip from being converted into text before being returned to the client. The WHERE clause takes advantage of a cube-valued functional index built on the DSChip column.

```
private PreparedStatement prepareToFetch(Connection conn,
              String table_name,
              String column_name) throws SQLException
{
    return conn.prepareCall("SELECT  dschip_send(" +
        column_name + ") FROM " + table_name +
        "  WHERE DSAsCubeString(chip,'x,y'::bpchar)::cube" +
        " && DSRangeToCube(?)::cube" );
}
```

The performQuery method launches the query, and constructs DSChip's from returned byte arrays.

```
private void performQuery(PreparedStatement ps, String
                    keyText, double [] xRange,
                          double [] yRange)
  throws java.sql.SQLException
{
    ps.setString(1, keyText);
    ResultSet rs = ps.executeQuery();
    while( rs.next()) {
        byte [] chipData = rs.getBytes(1);
        DSChip chip = new DSChip(chipData);
        processASingleChip(chip, xRange, yRange);
    }
    rs.close();
}
```

The processASingleChip method determines the indexes of the pertinent columns and then filters the rows before printing them.

```
private void processASingleChip(DSChip chip, double []
        xRange, double [] yRange)
{
    int xColumn = -1, yColumn = -1, valsColumn = -1;
    for(int i = 0, n = chip.getNumColumns(); i < n; i++ ){
        String columnName = chip.getColumnName(i);
        if( "x".equals(columnName)) xColumn = i;
        else if("y".equals(columnName)) yColumn = i;
        else if("val".equals(columnName)) valsColumn = i;
    }
  if( xColumn == -1|| yColumn == -1 || valsColumn == -1 )
  {
        return;
  }
  for(int i = 0, n = chip.getFilledRows(); i < n; i++ ) {
        double x = chip.getDouble(i, xColumn);
        double y  = chip.getDouble(i, yColumn);
        if( x >= xRange[0] && x <= xRange[1] &&
                y >= yRange[0] && y <= yRange[1] ) {
            System.out.println("(" + x + "," + y + ") = " +
                    chip.getDouble(i, valsColumn));
        }
  }
```

```
    }
```

The generateKey method generates an argument to be passed to DSRangeToCube, which in turn generates the cube string key.

```
    private String generateKey(double [] xRange, double []
                  yRange) {
          StringBuilder b = new StringBuilder();
          b.append("x ");
          b.append( xRange[0]);
          b.append( " ");
          b.append( xRange[1]);
          b.append( ", y ");
          b.append( yRange[0]);
          b.append( " ");
          b.append( yRange[1]);
          b.append( ")");
          return b.toString();
    }
```

The following is the equivalent of the main routine in the fetch1.c program.

```
    public Fetch() {
        try {
           double [] xRange = { 26, 39 };
           double [] yRange = { 31, 53 };
           Connection connection = createConnection();
           PreparedStatement ps =
                  prepareToFetch(connection, "xyvals",
                                  "chip");
           String key = generateKey(xRange, yRange);
           performQuery(ps, key, xRange, yRange);
           ps.close();
           connection.close();
        } catch( java.sql.SQLException e1 ) {
                  System.out.println(e1.toString());
        }
    }

   public static void main(String args[]) {
      new Fetch();
    }
}
```

Chapter 6: Updating Data in the Database

Currently, DBXten supports the ability to add tuples to partially filled DSChip's (see Adding Tuples to a DSChip on page 31).

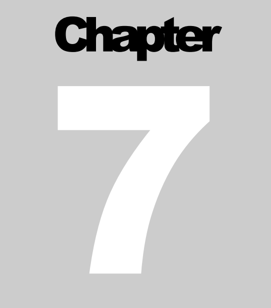
In addition, it is of course possible to delete (entire) DSChip's that satisfy some criteria:

Example

```
demo=> DELETE FROM measurementBlocks
WHERE DSAsCubeString(measurement_block,
                'datetime,latitude,longitude'::bpchar)::cube
        && DSRangeToCube('datetime 2008-01-01 12:00:00 2008-01-
01 14:00:00,latitude * *,longitude * *')::cube;

DELETE 2
```

Possible future features include:

- enlarging the capacity of an existing DSChip.
- deleting all the tuples that satisfy some given rangeSpec from one or more DSChip's. (e.g., delete all the DSChip's where x is between 1 and 10 and y = 40.)
- replacing one or more columns with a specified set of values in all DSChip's that satisfy some given rangeSpec. (e.g., set Z=10 and Y=40 in all DSChip's where X=3.)
- apply a simple function to one or more columns in all DSChip's that satisfy some given rangeSpec. (For example, add .3 to Z and subtract .5 from Y in all DSChip's where time is between *a* and *b*.) This might have applicability in instrument data post-processing.

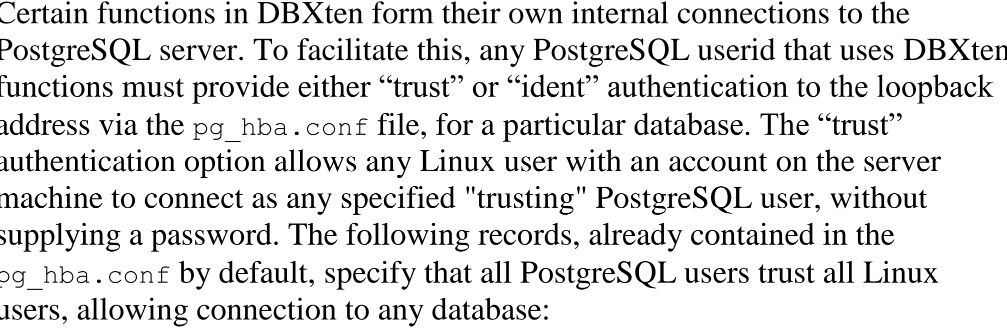


Chapter 7: Database Management Issues

This chapter describes the PostgreSQL database management issues that must be addressed when using the BCS DBXten Extension.

Database Security

The default installation of PostgreSQL and DBXten provides a minimal set of access rights. With this set, only client applications that are running on the server machine are allowed to connect to a database. Connection access is controlled by a file of access records called `<POSTGRESQLDIR>/data/pg_hba.conf`, defined by PostgreSQL. The PostgreSQL documentation[22] contains information on how to modify access records.



Certain functions in DBXten form their own internal connections to the PostgreSQL server. To facilitate this, any PostgreSQL userid that uses DBXten functions must provide either "trust" or "ident" authentication to the loopback address via the `pg_hba.conf` file, for a particular database. The "trust" authentication option allows any Linux user with an account on the server machine to connect as any specified "trusting" PostgreSQL user, without supplying a password. The following records, already contained in the `pg_hba.conf` by default, specify that all PostgreSQL users trust all Linux users, allowing connection to any database:

```
# TYPE  DATABASE  PG USER  IP-ADDRESS  IP-MASK         METHO
      D
local   all       all                                  trust
# IPv4-style local connections:
host    all       all      127.0.0.1   255.255.255.255 trust
```

To enhance security, replace these lines with the following:

[22] See http://www.postgresql.org/docs/7.4/interactive/client-authentication.html.

```
# TYPE   DATABASE   PG USER   IP-ADDRESS   IP-MASK         METHOD
local    db1,…      all                                    trust
local    all        all                                    someother
# IPv4-style local connections:
host     all        all       127.0.0.1    255.255.255.255 someother
```

where "db1,…" is a list of BCS DBXten Extension-enabled databases and "*someother*" is one of the other methods described in http://www.postgresql.org/docs/8.2/interactive/auth-methods.html.

Extra security can be achieved, at the expense of some run-time cost, by using the "ident" method of authentication. To use the "ident" method, replace these `pg_hba.conf` lines:

```
# TYPE   DATABASE   PG USER   IP-ADDRESS   IP-MASK         METHOD
local    all        all                                    trust
# IPv4-style local connections:
host     all        all       127.0.0.1    255.255.255.255 someother
```

with the following:

```
# TYPE   DATABASE   PG USER   IP-ADDRESS   IP-MASK     METHOD
local    db1,…      all                                ident map
local    all        all                                someother
# IPv4-style local connections:
host     all        all       127.0.0.1    255.255.255.255 someother
```

and include the following lines in a file called pg_ident.conf, stored in the same directory as `pg_hba.conf` (`<POSTGRESQLDIR>/data`):

```
# MAPNAME      IDENT-USERNAME    PG-USERNAME
map            postgres          postgres
map            postgres          pguser1
…
map            postgres          pusern
map            linuxuser1        pguser1
…
map            linuxusern        pusern
```

where `pguser`$_1$, …, `pguser`$_n$ are PostgreSQL userids and (`linuxuser`$_1$,`pguser`$_1$), …, (`linuxuser`$_n$,`pguser`$_n$) identify the equivalences between Linux userids and PostgreSQL userids.

Chapter 8: Troubleshooting Guide

This chapter provides guidance on resolving error messages that might be encountered while using the BCS DBXten Extension.

Connection Errors

"Unable to open connection to '*database_name*'. Check security permissions as a possible problem."
There is likely a problem with the `pg_hba.conf` file.

C Client Library Errors

"attempt to access tuple *num*, out of range"
This can be caused by specifying an invalid tuple index as the second parameter to one of the DSChipVarGet* functions.

"attempt to seek past last column"
This can be caused by specifying an invalid column index as the second parameter to the DSChipGetVarByPos function.

"attempt to set tuple *num*, out of range"
This can be caused by specifying an invalid tuple index as the second parameter to one of the DSChipVarSet* functions. Note that tuple indices start at 0, not 1.

"attempt to set string column with numeric value"
You are likely calling DSChipVarSet**String** and passing as the first argument a DSChipVar that is bound (through DSChipGetVarByName/Pos) to an **integer** or **floating point** column.

"attempt to get string column as numeric value"
You are likely calling DSChipVarGet**String** and passing as the first argument a DSChipVar that is bound (through DSChipGetVarByName/Pos) to an **integer** or **floating point** column.

"attempt to set numeric column with string value."
You are likely calling DSChipVarSet**Double** or DSChipVarSet**Int** and passing as the first argument a DSChipVar that is bound (through DSChipGetVarByName/Pos) to a **string** (char *) column.

"attempt to get numeric column as string value."

You are likely calling DSChipVarGet **Double** or DSChipVarGet**Int** and passing as the first argument a DSChipVar that is bound (through DSChipGetVarByName/Pos) to a **string** (char *) column.

General Operational Errors

`"attempt to add too many columns to DSChip"`
There is currently a limit of 100 columns in a DSChip tuple.

`"license expired"`
The temporary demonstration license key provided to you has expired. Contact BCS for a new license key.

`"failed to allocate` *`num`* **`bytes"`**
This is a general memory allocation failure.

`"text for appended tuple only had` *`num`* **`columns, DSChip needs`** *`num`***`"`**
This can be caused by calling DSChipAppendRow with an insufficient number of columns.

`"column` *`string`* **`not in DSChip"`**
This can be caused by specifying an invalid second parameter to DSChipGetVarByName. Check the DSChip schema (using DSChipSchema) to make sure the column name is spelled correctly.

`"no matches between column list and columns in DSChip"`
columnNames clause (see Figure 15) has no columns in common with the columns actually stored in the DSChip.

`"result string too long"`
This can be caused by calling DSMinAsString or DSMaxAsString on a DSChip that has extremely long column names.

`"column` *`string`* **`in range list not in source DSChip"`**
This can be caused by calling DSChipExtractToChip or DSChipExtractToSet with an invalid column name in the rangeSpec. For example,

```
SELECT DSChipExtractToSet(chip,'badColumnName 3 4','')
      from xyvals;
```

Corrupt or Misinterpreted Binary Data

`"attempt to compress already compressed or empty DSChip"`
Internal error – contact BCS

`"attempt to decompress corrupt string data"`
Internal error – contact BCS

`"attempt to deserialize DSChip from bad data (wrong prolog)"`

Internal error – contact BCS

Bad ASCII Data

"bad DSChip header in DSChip text"
e.g., in the following error message:

```
ERROR:  DSChipInputText user error: bad DSChip header in
DSChip text
CONTEXT:  COPY measurementblocks, line 5, column
measurement_block:
"maxtuples=2,fiedtuples=2,numcolumns=4;datetime,date,10;la
titude,float,0.001;longitude,float,0.001;intColumn..."
```

the header portion of the DSChip text should have "filledtuples=2" instead of "fiedtuples=2". If you get this error when loading DSChip values from a text file (using COPY or \COPY), there is likely an error in the text file (on the 5th line, in this case).

"bad double value *string*"
When parsing a DSChip textual representation, "string" was found where a double or float value should have been. If you get this error when loading DSChip values from a text file (using COPY or \COPY), there is likely an error in the text file.

"bad int value *string*"
When parsing a DSChip textual representation, "string" was found where an integer value should have been. If you get this error when loading DSChip values from a text file (using COPY or \COPY), there is likely an error in the text file.

"bad max expr for *string* **was** *string*"
error in range clause (see figure 16)

"bad min expr for *string* **was** *string*"
error in range clause (see figure 16)

"bad or missing 'maxtuples' term in DSChip schema"
caused by bad literal value in call to DSChipNew

"bad precision text for column *columnName*, **was** *string*"
e.g., in the following error message:

```
ERROR:  DSChipInputText user error: bad precision text for
column latitude, was x.001
CONTEXT:  COPY measurementblocks, line 5, column
measurement_block:
"maxtuples=2,filledtuples=2,numcolumns=4;datetime,date,10;
latitude,float,x.001;longitude,float,0.001;int..."
```

the header portion of the DSChip text has an invalid floating point value for the precision ("x.001" instead of ".001"). If you get this error when loading DSChip values from a text file (using COPY or \COPY), there is likely an error in the text file (on the 5th line, in this case).

"DSChip already has a column of name *string*"
This can be caused by calling the C API function DSChipAddVar and passing it a column name that is already in the DSChip schema.

"empty DSChip schema"
This can be caused by passing an empty string to the C API function DSChipNew.

"empty DSChip text"
This can be caused by calling the SQL function DSChip() with an empty string. For example,

```
SELECT DSChip();
```

"empty column list"
This can be caused by passing a blank list (not an empty string) to the columnNames clause of a function (see Figure 16 on page 53). For example,

```
SELECT DSChipExtractToSet(chip,'x 26 26, y 50 52',' ')
FROM xyvals;
```

instead of

```
SELECT DSChipExtractToSet(chip,'x 26 26, y 50 52','') FROM
xyvals;
```

"missing column name in range list"
This can be caused by passing an empty Range clause as part of the rangeSpec clause of a function (see Figure 17 on page 53). For example,

```
SELECT DSChipExtractToSet(chip,', y 50 52','y') FROM
xyvals;
```

instead of

```
SELECT DSChipExtractToSet(chip,'y 50 52','y') FROM xyvals;
```

"missing min expr for *columnName*"
There was an error in the rangeSpec clause – see Figure 17 (page 53).

"schema did not contain any column definitions"
This can be caused by passing an incomplete DSChip schema specification to DSChipNew() – see Figure 12 (page 30). For example,

```
SELECT DSChipNew('maxtuples 2');
```

"truncated text when reading column name"
This can be caused by casting an incomplete DSChip textual representation to DSChip. For example,

```
SELECT
'maxtuples=2,filledtuples=2,numcolumns=3;x,integer,0;'
::DSChip;
```

See The DSChip Data Type on page 25 for examples of valid DSChip textual representations.

"truncated text when reading columns"
This can be caused by casting an incomplete DSChip textual representation to DSChip.

"truncated text when reading column type"
This can be caused by casting an incomplete DSChip textual representation to DSChip.

"type for column *columnName* **was invalid"**
This can be caused by casting an incomplete DSChip textual representation to DSChip.

"unexpected text following range, saw *string*, **check for missing comma?"**
This can be caused by specifying extra text at the end of a rangeSpec (see Figure 17 on page 53). For example,

```
SELECT DSChipExtractToSet(chip,'y 50 52 extrastuff','y')
FROM xyvals;
```

instead of

```
SELECT DSChipExtractToSet(chip,'y 50 52','y') FROM xyvals;
```

Bad DateTime Data

"time offset restricted to +-14 hours from GMT"
The timezone value must be between -14 and +14.

"unrepresentable time *string*"
DBXten is currently only able to represent datetime values that are in the range Jan 1 1902 to Dec 21 2037. This is due to the Unix internal datetime representation used on 32 bit machines.

"unexpected text trailing the date range = *string*"
The specified timestamp range had some invalid characters following the second timestamp value – e.g.:

```
SELECT DSRangeToCube('datetime 1970-01-01 01:00:00 1970-
01-01 01:00:01 badstuff, latitude 49.0 49.01');
ERROR:  DSRangeToCube user error: unexpected text trailing
the date range = badstuff
```

"missing date range"

The following example shows how this error can arise:

```
-- good SQL (timestamp range has two values)
SELECT measurement_block_id,
                DSChipNumMatches(measurement_block,
                        'datetime 2008-01-01 12:00:00
2008-01-01 14:00:00,latitude * *,longitude * *')
        FROM measurementBlocks ;
 measurement_block_id | dschipnummatches
----------------------+------------------
                  480 |                4
                  481 |                4
                  482 |                4
                  483 |                2
                  484 |                2
                  485 |                0
                  486 |                2
                  487 |                0
                  488 |                0
(9 rows)

-- bad SQL (timestamp range is missing)
SELECT measurement_block_id,
           DSChipNumMatches(measurement_block,
                  'datetime,latitude * *,longitude * *')
        FROM measurementBlocks ;
ERROR:  DSChipExtractAsChip user error: missing date range
```

"bad time value *string*"

This is a general catch-all message indicating that there is something wrong with the timestamp value:

```
SELECT DSRangeToCube('datetime 1970-01-01 01:00:00/1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSChipExtractAsChip user error: bad time value
2008-01-01
```

"bad year in datetime *string*"

The year portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 11970-01-01 01:00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSChipExtractAsChip user error: bad year in
datetime '11970-01-01 01:00:00'
```

"expected - after year in datetime *string* "

The year-month separator wasn't a hyphen as expected. For example,

```
SELECT DSRangeToCube('datetime 1970/01-01 01:00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: expected - after year in
datetime '1970/01-01 01:00:00'
```

"bad month in datetime *string* **"**
The year portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 1970-13-01 01:00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: bad month in datetime
'1970-13-01 01:00:00'
```

"expected - after month in datetime *string* **"**
The month-day separator wasn't a hyphen as expected. For example,

```
SELECT DSRangeToCube('datetime 1970-01/01 01:00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: expected - after month in
datetime '1970-01/01 01:00:00'
```

"bad day of month in datetime *string* **"**
The day portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 1970-02-32 01:00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: bad day of month in
datetime '1970-02-32 01:00:00'
```

"expected space between date and time in datetime *string* **"**
The day-hour separator wasn't a space as expected. For example,

```
SELECT DSRangeToCube('datetime 1970-01-01/01:00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: expected space between
date and time in datetime '1970-01-01/01:00:00'
```

"bad hour in datetime *string* **"**
The hour portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 1970-02-01 32:00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: bad hour in datetime
'1970-02-01 32:00:00'
```

"expected ':' after hour in datetime *string* **"**
The hour:minute separator wasn't a colon as expected. For example,

```
SELECT DSRangeToCube('datetime 1970-01-01 01/00:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: expected : after month in
datetime '1970-01-01 01/00:00'
```

"bad minute in datetime *string* **"**

The minute portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 1970-01-01 01:70:00 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: bad minute in datetime
'1970-01-01 01:70:00'
```

"bad second in datetime *string* **"**

The (optional) second portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 1970-02-01 01:00:80 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: bad second in datetime
'1970-02-01 01:00:80'
```

"bad fractional second in datetime *string* **"**

The (optional) fractional second portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 1970-02-01 01:00:01.1234567
1970-01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: bad second in datetime
'1970-02-01 01:00:01.1234567'
```

Note that the fractional second portion can be no more than 6 digits long.

"bad timezone in datetime *string* **"**

The (optional) timezone portion of a timestamp field was invalid. For example,

```
SELECT DSRangeToCube('datetime 1970-02-01 01:00:01+15
1970-01-01 01:00:00, latitude 49.0 49.01');
ERROR: DSRangeToCube user error: bad timezone in datetime
'1970-02-01 01:00:01.1234567'
```

Note that the timezone must be between -14 and +14.

"Unexpected text text following timeStamp *string* **"**

One cause of this error is specifying an invalid delimiter between the minutes portion of a timestamp and the optional seconds portion. For example,

```
SELECT DSRangeToCube('datetime 1970-02-01 01:00/01 1970-
01-01 01:00:00, latitude 49.0 49.01');
ERROR:  DSRangeToCube user error: Unexpected text '/01'
following timeStamp '1970-02-01 01:00/01'
```

"bad time zone direction *char* **in** *string***"**

For example, specifying '2008-01-01 12:30:30(12)' instead of ‘2008-01-01 12:30:30+12’

"bad time zone value *num* **in** *string***"**

For example, specifying '2008-01-01 12:30:30+36' instead of '2008-01-01 12:30:30+12'

"bad date in was ***string*****"**
This is a catch-all error for invalid timestamp values.

Appendix A: Complete List of BCS DBXten Extension User-Defined Routines (UDRs)

The following table lists, in alphabetical order, the user-callable SQL functions that are included in the BCS DBXten Extension.

Function	**Page**
FUNCTION DSAsCubeString(*existingChip* DSChip, *columnNames* char) RETURNS text	58
FUNCTION DSAsGeoCubeString (*existingChip* DSChip, *columnSpec* CHAR) RETURNS TEXT	35
FUNCTION DSChipAppendRow(*existingChip* DSChip, *valuesAsString* CHAR) RETURNS DSChip	31
FUNCTION DSChipExtractToBytes(*existingChip* DSChip, *rangeSpec* char, *columnNames* char) RETURNS bytea	64
FUNCTION DSChipExtractToChip(*existingChip* DSChip, *rangeSpec* char, *columnNames* char) RETURNS DSChip	62
FUNCTION DSChipExtractToRowChip(*existingChip* DSChip, *rangeSpec* char, *columnNames* char) RETURNS setof DSChip	64
FUNCTION DSChipNew(*schema* CHAR) RETURNS DSChip	29
FUNCTION DSChipNumMatches(*existingChip* DSChip, *rangeSpec* char) RETURNS integer	62
FUNCTION DSChipSchema(*existingChip* DSChip) RETURNS char	54
FUNCTION DSChipToRowChip(*existingChip* DSChip) RETURNS setof DSChip	60
FUNCTION DSChipToStrings(*existingChip* DSChip,	54

columnNames char) RETURNS setof char	
FUNCTION DSCompressInfo(*existingChip* DSChip) RETURNS TEXT	57
FUNCTION DSDistinct(*existingChip* DSChip, *columnNames* char) RETURNS setof char	54
FUNCTION DSDoubleToGMTTime(double precision) RETURNS char	28
FUNCTION DSDoubleToLocalTime(double precision) RETURNS char	28
FUNCTION DSGetDate(chip DSChip, columnPosition integer) RETURNS Timestamp	57
FUNCTION DSGetDouble(chip DSChip, columnPosition integer) RETURNS Double Precision	57
FUNCTION DSGetInteger(chip DSChip, columnPosition integer) RETURNS Integer	56
FUNCTION DSGetInt8(chip DSChip, columnPosition integer) RETURNS bigint	57
FUNCTION DSGetText(chip DSChip, columnPosition integer) RETURNS TEXT	56
FUNCTION DSGetVersion() RETURNS TEXT	24
FUNCTION DSGMTTimeToDouble(char) RETURNS double precision	28
FUNCTION DSHasVariables(*existingChip* DSChip, *columnNames* char) RETURNS boolean	55
FUNCTION DSLocalTimeToDouble(char) RETURNS double precision	28
FUNCTION DSMaxAsString(*existingChip* DSChip, *columnNames* char) RETURNS TEXT	56
FUNCTION DSMinAsString(*existingChip* DSChip, *columnNames* char) RETURNS TEXT	56
FUNCTION DSNumFilledRows(*existingChip* DSChip) RETURNS integer	55
FUNCTION DSNumVars(*existingChip* DSChip) RETURNS integer	55
FUNCTION DSRangeToCube(*rangeSpec* char) RETURNS text	58

Appendix B: The C API

List of C API Constants

These constants are defined in `includes/dschip_const.h` in the DBXten distribution.

Size Limits

```
dsNameLENGTH=80
dsMaxVARIABLES=100
```

Supported Data Types

```
DSVarTypeINT
DSVarTypeDOUBLE
DSVarTypeDATE
DSVarTypeSTRING
DSVarTypeINT8
```

List of C API Functions

This is a summary of the functions available in the DBXten C API. These functions are all declared in `includes/dschip_exports.h` in the DBXten distribution.

DSChip-specific Functions

CREATE A DSCHIP

```
DSChip *DSChipCreate(int maxTuples);
```

DISPOSE OF A PREVIOUSLY CREATED DSCHIP.

```
void DSChipFree(DSChip *chip);
```

CLEAR ANY ROWS STORED IN THE DSCHIP.

```
void DSChipClearRows(DSChip *chip);
```

ASK HOW MANY ROWS THE DSCHIP IS ACTUALLY HOLDING.

```
int DSChipGetNumRows(DSChip *chip);
```

ASK HOW MANY COLUMN VARIABLES THE DSCHIP IS HOLDING.

```
int DSChipGetNumVars(DSChip *chip);
```

ADDS A NEW COLUMN VARIABLE TO THE DSCHIP, AND RETURNS ITS INDEX.

```
int DSChipAddVar(DSChip *chip, char *varName, int varType,
double tolerance);

int DSChipAddVarWithUnit(DSChip *chip, char *varName,
char *unitName, int varType, double tolerance);
```

GETS A COLUMN VARIABLE BY ITS POSITION (0...NUMCOLUMNS-1).

```
DSChipVar * DSChipGetVarByPos(DSChip *chip, int position);
```

GETS A COLUMN VARIABLE BY ITS NAME.

```
DSChipVar * DSChipGetVarByName(DSChip *chip, char
*varName);
```

CHECK IF A COLUMN VARIABLE EXISTS IN A DSCHIP. RETURNS 0 IF NOT FOUND, 1 IF FOUND.

```
int DSChipCheckForVariable(DSChip *chip, char *varName);
```

RETURN THE NUMBER OF BYTES NEEDED FOR A SERIAL REPRESENTATION OF A DSCHIP.

```
int DSChipToByteLength(DSChip *chip);
```

SERIALIZE THE DSCHIP TO THE SUPPLIED ARRAY OF BYTES.

```
void DSChipToBytes(DSChip *chip, void *destBytes);
```

BUILD A DSCHIP FROM A SERIALIZED REPRESENTATION. THE LENGTH ARGUMENT IS CURRENTLY IGNORED.

```
DSChip *DSChipFromBytes(void *srcBytes, int length);
```

DSChip Column-specific Functions

GET THE NAME OF A COLUMN VARIABLE.

```
char *DSChipVarGetName(DSChipVar *chipVar);
```

GET THE UNITS OF A COLUMN VARIABLE.

```
char *DSChipVarGetUnits(DSChipVar *chipVar);
```

A zero-length string is returned if the column has no units defined.

SET THE UNITS OF A COLUMN VARIABLE.

```
void DSChipVarSetUnits(DSChipVar *chipVar, char *units);
```

ASSIGN A STRING VALUE TO A PARTICULAR ROW IN CHIPVAR.

```
void DSChipVarSetString(DSChipVar *chipVar, int row, char
      * value);
```

ASSIGN A DOUBLE VALUE TO A PARTICULAR ROW IN CHIPVAR.

```
void DSChipVarSetDouble(DSChipVar *chipVar, int row,
      double value);
```

ASSIGN A 32 BIT INTEGER VALUE TO A PARTICULAR ROW IN CHIPVAR.

```
void DSChipVarSetInt(DSChipVar *chipVar, int row, int
      value);
```

ASSIGN A 64 BIT INTEGER VALUE TO A PARTICULAR ROW IN CHIPVAR.

```
void DSChipVarSetInt8(DSChipVar *chipVar, int row, int64_t
      value);
```

GET THE *ROW*TH ROW OF CHIPVAR AS A STRING.

```
char * DSChipVarGetString(DSChipVar *chipVar, int row);
```

GET THE *ROW*TH ROW OF CHIPVAR AS A DOUBLE.

```
double DSChipVarGetDouble(DSChipVar *chipVar, int row);
```

GET THE *ROW*TH ROW OF CHIPVAR AS A 32 BIT INTEGER.

```
int DSChipVarGetInt(DSChipVar *chipVar, int row);
```

GET THE *ROW*TH ROW OF CHIPVAR AS A 64 BIT INTEGER.

```
int64_t DSChipVarGetInt8(DSChipVar *chipVar, int row);
```

GET THE TYPE OF A VARIABLE.

```
int DSChipVarGetType(DSChipVar *chipVar);
```

(Returns one of: `DSVarTypeINT` `DSVarTypeDOUBLE` `DSVarTypeDATE`.)

GET THE TOLERANCE/PRECISION OF A DSCHIP VARIABLE.

```
double DSChipVarGetTolerance(DSChipVar *chipVar);
```

(This quantity is meaningless for integer columns.)

Standard Support Functions.

A WRAPPER FOR MALLOC.

```
void * DSMalloc(size_t size);

void * DSCalloc(size_t size, size_t nelem);
```

A WRAPPER FOR FREE.

```
void DSFree(void *);
```

A WRAPPER FOR STRDUP.

```
char *DSStrdup(char *);
```

PRINTF STYLE ERROR REPORTING FOR USER ERRORS.

```
void DSUserError(char *fmt, ...);
```

This function does not return.

PRINTF STYLE ERROR REPORTING FOR CODING ERRORS.

```
void DSCodeError(char *fmt, ...);
```

This function does not return.

USED TO SUPPLY THE NAME OF THE APPLICATION IN WHICH THE ERROR OCCURRED.

```
char *DSErrSource(char *);
```

Only a shallow copy of the argument is made, so the user must guarantee the integrity of the memory pointed to for the life of the application.

CONVERT A TIME VALUE EXPRESSED IN SECONDS SINCE MIDNIGHT, JAN 1 1970 TO A STRING IN THE FORMAT YYYY-MM-DD HR:MN:SS IN THE LOCAL TIMEZONE.

```
void DSDoubleToLocalString(char *dest,
      double timeInSeconds,
      int fractionSize);
```

`fractionSize` is the number of digits after the decimal point in the seconds.

CONVERT A TIME REPRESENTED AS A STRING IN THE FORMAT YYYY-MM-DD HR:MN:SS TO THE NUMBER OF SECONDS SINCE MIDNIGHT, JAN 1 1970 IN THE LOCAL TIMEZONE.

```
double DSLocalStringToDouble(char *timeAsString);
```

CONVERT A TIME VALUE EXPRESSED IN SECONDS SINCE MIDNIGHT, JAN 1 1970 TO A STRING IN THE FORMAT YYYY-MM-DD HR:MN:SS IN GREENWICH MEAN TIME (GMT).

```
void DSDoubleToGMTString(char *dest,
      double timeInSeconds,
      int fractionSize);
```

`fractionSize` is the number of digits after the decimal point in the seconds.

CONVERT A TIME REPRESENTED AS A STRING IN THE FORMAT YYYY-MM-DD HR:MN:SS TO THE NUMBER OF SECONDS SINCE MIDNIGHT, JAN 1 IN GREENWICH MEAN TIME (GMT).

```
double DSGMTStringToDouble(char *timeAsString);
```

Appendix C: The Java API

List of Java API Functions

This is a summary of the functions available in the DBXten Java API. These functions are all defined in `com.barrodale.dschip.DSChip` class. A Javadoc version of this summary is provided with the DBXten distribution, in the `javadocs` directory.

DSChip Class Constants

DEFINE THE INDEX OF THE FIRST ROW IN THE DSCHIP

```
public final static int firstRowNum = 0;
```

DEFINE THE INDEX OF THE FIRST COLUMN IN THE DSCHIP

```
public final static int firstColumnNum = 0;
```

PSEUDO-ENUMERATED TYPE FOR 32 BIT INTEGER COLUMNS

```
public static final int varTypeINT = 0;
```

PSEUDO-ENUMERATED TYPE FOR 64 BIT INTEGER COLUMNS

```
public static final int varTypeINT8 = 4;
```

PSEUDO-ENUMERATED TYPE FOR DOUBLE COLUMNS

```
public static final int varTypeDOUBLE = 1;
```

PSEUDO-ENUMERATED TYPE FOR DATE COLUMNS

```
public static final int varTypeDATE = 2;
```

PSEUDO-ENUMERATED TYPE FOR STRING COLUMNS

```
public static final int varTypeSTRING = 3;
```

MAXIMUM LENGTH OF A COLUMN IN A DSCHIP.

```
public final static int dsNameLENGTH = 80;
```

MAXIMUM NUMBER OF COLUMNS IN A DSCHIP.

```
public final static int dsMaxVARIABLES = 100;
```

DSChip-Building Functions

CREATE A DSCHIP

```
public DSChip(int maxTuples)
```

Initially the DSChip has no columns and no filled rows. The input parameter is the maximum number of rows that can be stored in the DSChip.

DEFINE A NEW COLUMN

```
public int addColumn(String columnName, int columnType, double tolerance)
```

```
public int addColumn(String columnName, String unitName, int columnType, double tolerance)
```

Add a column (possibly with units) to the DSChip.

ADD A DATE COLUMN

```
public int addDateColumn(String columnName, double tolerance)
```

This is equivalent to calling `addColumn` with `columnType` set to `varTypeDATE`.

ADD AN INTEGER COLUMN

```
public int addIntColumn(String columnName)
```

```
public int addIntColumn(String columnName, String unitName)
```

```
public int addInt8Column(String columnName)
```

```
public int addInt8Column(String columnName, String unitName)
```

These are equivalent to calling `addColumn` with `columnType` set to `varTypeINT`.

ADD A DOUBLE COLUMN

```
public int addDoubleColumn(String columnName, double tolerance)
```

```
public int addDoubleColumn(String columnName, String unitName, double tolerance)
```

These are equivalent to calling `addColumn` with `columnType` set to `varTypeDOUBLE`.

ADD A STRING COLUMN

```
public int addStringColumn(String columnName)
```

This is equivalent to calling `addColumn` with `columnType` set to `varTypeSTRING`.

SET THE VALUE OF A DATE COLUMN

```
public void setDate(int rowNum, int column, Timestamp value)
```

SET THE VALUE OF AN INTEGER COLUMN

```
public void setInt(int rowNum, int column, int value)
public void setInt8(int rowNum, int column, long value)
```

SET THE VALUE OF A DOUBLE COLUMN

```
public void setDouble(int rowNum, int column, double value)
```

SET THE VALUE OF A STRING COLUMN

```
public void setString(int rowNum, int column, String value)
```

RETURN A COMPRESSED FORM OF THE DSCHIP JAVA OBJECT

```
public byte [] toBytes()
```

This function is called to produce a byte array that can be sent to the PostgreSQL server.

DSChip-Extraction Functions

CREATE A DSCHIP FROM A BYTE ARRAY

```
public DSChip(byte [] rawForm)
```

The byte array is typically supplied by the PostgreSQL server.

RETURN THE NUMBER OF COLUMNS

```
public int getNumColumns()
```

RETURN THE MAXIMUM NUMBER OF ROWS THE DSCHIP WILL HOLD

```
public int getMaxRows()
```

RETURN THE NUMBER OF ROWS ACTUALLY HOLDING DATA

```
public int getFilledRows()
```

RETURN THE NAME OF THE *COLUMN*TH COLUMN

```
public String getColumnName(int column)
```

RETURN THE TYPE OF THE *COLUMN*TH COLUMN

```
public int getColumnType(int column)
```

RETURN THE UNITS OF THE *COLUMN*TH COLUMN

```
public int getColumnUnitName(int column)
```

RETURN THE PRECISION/TOLERANCE OF THE *COLUMN*TH COLUMN

```
public double getColumnTolerance(int column)
```

RETURN THE VALUE OF THE ELEMENT AT THE *ROWNUM*TH ROW AND *COLUMN*TH COLUMN AS A DOUBLE VALUE

```
public double getDouble(int rowNum, int column)
```

RETURN THE VALUE OF THE ELEMENT AT THE *ROWNUM*TH ROW AND *COLUMN*TH COLUMN AS A DOUBLE VALUE

```
public Timestamp getDate(int rowNum, int column)
```

RETURN THE VALUE OF THE ELEMENT AT THE *ROWNUM*TH ROW AND *COLUMN*TH COLUMN AS AN INT VALUE

```
public int getInt(int rowNum, int column)
public long getInt8(int rowNum, int column)
```

RETURN THE VALUE OF THE ELEMENT AT THE *ROWNUM*TH ROW AND *COLUMN*TH COLUMN AS A STRING VALUE

```
public String getString(int rowNum, int column)
```

CLEAR ALL THE ROWS SO THE OBJECT CAN BE REUSED

```
public void clearRows():
```

Note that the schema is not changed.

"FIX" A TIMESTAMP

```
public java.sql.Timestamp fixTimestamp(java.sql.Timestamp
dateValue)
```

This method adjusts a timestamp that was created by supplying a local time string to a function that expected a GMT string.

FORMAT AS A GMT STRING A VALUE THAT IS IN TIME IN SECONDS SINCE MIDNIGHT JAN 1 1970.

```
public java.lang.String formatDoubleAsGMT(double value, double
tolerance)
```

FORMAT AS A LOCAL TIME ZONE STRING A VALUE THAT IS IN TIME IN SECONDS SINCE MIDNIGHT JAN 1 1970.

```
public java.lang.String formatDoubleAsLocal(double value, double
tolerance)
```

GET A DATETIME AS A STRING IN THE GMT TIMEZONE.

```
public java.lang.String getDateAsGMTString(int rowNum, int
columnNum)
```

GET A DATETIME AS A STRING IN THE LOCAL TIMEZONE.

```
public java.lang.String getDateAsLocalString(int rowNum, int
columnNum)
```

CONVERTS A GMT DATE STRING TO THE TIME IN SECONDS SINCE MIDNIGHT, JAN 1, 1970.

```
public double gmtDateStringToDouble(java.lang.String dateString)
```

CONVERTS A LOCAL DATE STRING TO THE TIME IN SECONDS SINCE MIDNIGHT, JAN 1, 1970.

```
public double localDateStringToDouble(java.lang.String
dateString)
```

SET A DATETIME COLUMN USING A LOCAL TIMEZONE STRING.

```
public void setLocalDateAsString(int rowNum, int column,
java.lang.String localDateString)
```

SET A DATETIME COLUMN USING A GMT STRING.

```
public void setGMTDateAsString(int rowNum, int column,
java.lang.String gmtDateString)
```

Appendix D: A Tutorial

A tutorial will be provided in a future release of the product.

Appendix E: CSV File Reader Utility

The CSV File Reader Utility (`csvChipLoader)` can be used to load a delimited text file of tuples into DSChip's. The utility can be run either on the same machine as that on which the PostgreSQL server resides or it can be run on a different machine (and communicate with the PostgreSQL server machine via TCP/IP).

Downloading the CSV File Reader Utility

An executable for the `csvChipLoader` program is supplied with DBXten, in `<DBXTENDIR>/bin/csvChipLoader`.

Alternatively, the source code and support files can be downloaded from http://www.barrodale.com/bcs/software/DBXtenClients/csvChipLoader.zip as a zip file. To unzip the file, issue the command:

```
unzip csvChipLoader.zip
```

This should produce a directory called `csvChipLoader` containing the following files:

- `src`: a directory with the source code for `csvChipLoader`.
- `csvChipLoader`: the actual program.
- `schema.sql`: an SQL script for defining a destination table called sampleChips.
- `sample.csv`: a sample .csv file used in the example below.
- `args`: a text file used in the example below.

Compiling the CSV File Reader Utility

Note: The program is supplied in binary form as well as source, so it is not necessary to compile the program unless you want to run it on a different platform than the supplied distribution.

To compile the program, enter the `csvChipLoader/src` directory and edit the `Makefile`, updating the definitions of `DBXTENDIR` and `POSTGRESQLDIR` if necessary. Then execute the `make` program. This will create an executable called `csvChipLoader` in the `csvChipLoader` directory.

Running the CSV File Reader Utility

The command line arguments for `csvChipLoader` are:

Command line argument	**Meaning**
-rowsPerChip *value*	The maximum number of tuples to put in a DSChip. All tuples except possibly the last one created will have this number of tuples. The default is 1000.
-monitor *count*	Output a progress message every *count* tuples.
-i *path*	The name of the file to be loaded. This name must be resolvable in the environment under which the CVS File Reader is running[23].
-indelimit *delimiter*	The delimiter to be used to separate fields in the input file. The word "tab" can be used as a special case to denote a tab character. The default is ','.
-h *host name*	The name of the machine on which the PostgreSQL server is running. The default is "localhost", which is only appropriate if the loader is running on the same machine as the PostgreSQL server.
-localTime	Specifies that timestamp values are in local time rather than GMT.
-port *port number*	The port number under which the PostgreSQL server is running. The default is 5432 (i.e., the default PostgreSQL port).

[23] For example, the file cannot be on a different machine from the one where the loading program is running (the *client machine*) unless the file has been network-mounted onto the client machine.

-u *username*	The PostgreSQL user name to use when connecting to the database. The default is $PGUSER.
-p *password*	The PostgreSQL password to use when connecting to the database.
-d *database*	The name of the PostgreSQL database to connect to. The default is $PGDATABASE.
-t *tablename*	The name of the PostgreSQL table containing the DSChip column to be inserted into. The default is "`mychips`".
-c *columnname*	The name of the DSChip column to be inserted into. The default is "`chip`".

column_name[=unit_name]: *column_type*[:*precision*]

The tuple column name, type, and precision. There will in general be several of these, one for each column in the tuple / field in the input file. Excess fields in the csv file are ignored.

The *column_type* value must be one of "int", "int8", "double", "datetime", and "string". As a convenience, the following can also be used for the column type: "integer", "long", "float", "time", and "date", and "text."

The *precision* is optional and can only be specified for types datetime, float, and double. See the discussion on page 30 for a description of the *precision* parameter.

There are two mechanisms for embedding spaces in *unit_name*.

1) you can quote the entire *unit_name*, or

2) you can use '@' instead of a space; any @'s will be

translated to ' ' at run time.

See the discussion on page 27 for a description of the *unit_name* parameter.

Example

```
% cat schema.sql

DROP TABLE sampleChips;

CREATE TABLE sampleChips (
  chip DSChip
);

% head sample.csv
0,0.237788,2008-1-1 0:0:0,1234567890124,alpha
1,0.291066,2008-1-1 0:0:1,2345678901234,beta
2,0.845814,2008-1-1 0:0:2,3456789012345,gamma
3,0.152208,2008-1-1 0:0:3,3456789012345,gamma
4,0.585537,2008-1-1 0:0:4,3456789012345,gamma
5,0.193475,2008-1-1 0:0:5,3456789012345,gamma
6,0.810623,2008-1-1 0:0:6,3456789012345,gamma
7,0.173531,2008-1-1 0:0:7,3456789012345,gamma
8,0.484983,2008-1-1 0:0:8,3456789012345,gamma
9,0.151863,2008-1-1 0:0:9,3456789012345,gamma

% wc sample.csv
 10000  20000 465004 sample.csv

#
# Make sure that you change the following -h argument to the
# appropriate hostname value!
#
% cat args
-h carbon
-port 5432
-d demo
-u demo
-t sampleChips
-c chip
-rowsPerChip 1000
-i sample.csv
-indelimit ,
-monitor 2
id:int pressure=atm:double:0.0001
obstime=time@in@seconds@since@1970@udt:date:1
globalid:int8
description:string

% psql demo -U demo < schema.sql
ERROR:  table "samplechips" does not exist
CREATE TABLE

% csvChipLoader `cat args`
```

```
% psql demo –U demo

demo=> SELECT count(*) FROM sampleChips;

count
-------
   10
(1 row)
```

Appendix F:
NetCDF File Reader Utility

Overview

The netCDF file reader utility (`ncToDBXten`) is a program that reads an arbitrary netCDF file and loads it into a table of DSChip's. The `ncToDBXten` program is supplied as C source code (as well as a binary executable) and its only build-dependencies are on the netCDF library.

The conversion of data from a netCDF file to rows in database tables is directed by a user supplied "run-control" file.

One of the design decisions behind the converter is that it reads the entire netCDF file before loading any data to the database. This has several consequences:

1. The amount of seeking done by the netCDF library is kept to a minimum, improving performance.
2. There is no contention for disk resources between the program's reading of data, and the database's use of the disk to store data.
3. Only netCDF files small enough to fit in memory can be loaded using this program.

The Run-Control File

The Overall Syntax

A Run control file has lines of the form

```
table_name = major_dim,...,minor_dim,first_grid_variable,...,
last_grid_variable
```

The *table_name* is the name of the table that the data will be loaded into. The table_name should be a sequence of alphanumeric characters without any embedded spaces or punctuation symbols.

The list of dimensions (major_dim ... minor_dim) must mirror the dimensions of the grid variables (first_grid_variable ... last_grid_variable), in both number and order. This means, of course, that the grid variables in a particular line must all have the same list of dimensions.

Dimension Syntax

Each dimension item in the *table_name* = ... line has the form

```
dim_name[:tile_size][;precision]
```

where `dim_name` is the name of the dimension in the netCDF file. The optional `tile_size` determines how many samples of this dimension are stored in each tile. Note that a semicolon (";"), not a colon (":"), precedes the `precision` value.

Grid Variable Syntax

Each grid variable item in the *table_name* = ... line has the form

```
variable_name[;precision ]
```

Comments and Whitespace

The # character indicates that the remainder of the line it appears on is a comment. Entirely blank lines are treated as white space (ignored).

Downloading the NetCDF File Reader Program

An executable for the `ncToDBXten` is supplied with DBXten, in `<DBXTENDIR>/bin/ncToDBXten`

Alternatively, the source code and support files can be downloaded from http://www.barrodale.com/bcs/software/DBXtenClients/ncToDBXten.zip as a zip file. To unzip the file, issue the command:

```
unzip ncToDBXten.zip
```

This should produce a directory called `ncToDBXten` containing the following files:

- `src`: a directory with the source code for the `ncToDBXten` program.
- `ncToDBXten`: the actual executable.

- `schema.sql`: an SQL script for defining a destination table called sampleChips.
- `test`.nc: a simple netCDF file with a 3D grid.
- `test`.rc: a run-control file to be used with test.nc.
- `test.cdl`: a text version of test.nc.

Compiling

Note: The program is supplied in binary form as well as source, so it is not necessary to compile the program unless you want to run it on a different platform than the supplied distribution or make modifications to it.

To compile the program, you must first install the netCDF library (C API), which is freely available from http://www.unidata.ucar.edu/downloads/netcdf/. The `ncToDBXten` program has been tested with version 3.6 of the library.

After you have installed the netCDF library, enter the `ncToDBXten/src` directory and edit the `Makefile`, updating the definitions of `DBXTENDIR` and `POSTGRESQLDIR` if necessary. Then execute the `make` program. This will create an executable called `ncToDBXten` in the `ncToDBXten` directory.

Running the NetCDF File Reader Utility

The command line arguments for `ncToDBXten` are:

Command line argument	Meaning
-i *path*	The name of the file to be loaded. This name must be resolvable in the environment under which the NetCDF File Reader is running[24].
-h *host name*	The name of the machine on which the PostgreSQL server is running. The default is "localhost", which is only appropriate if the loader is running on the same machine as the PostgreSQL server.
-port *port number*	The port number under which the PostgreSQL server is running. The default is 5432 (i.e., the default

[24] For example, the file cannot be on a different machine from the one where the loading program is running (the *client machine*) unless the file has been network-mounted onto the client machine.

	PostgreSQL port).
-u *username*	The PostgreSQL user name to use when connecting to the database. The default is $PGUSER.
-p *password*	The PostgreSQL password to use when connecting to the database.
-d *database*	The name of the PostgreSQL database to connect to. The default is $PGDATABASE.
-rc *configurationFile*	The path of the configuration file.

Example

The directory containing the source code for the `ncToDBXten` converter also contains files to demonstrate its use. As a simple first example, try:

```
% cd netcdfToRaw
% ncdump test.nc | head -30

netcdf test {
dimensions:
        lon = 180 ;
        lat = 170 ;
        time = UNLIMITED ; // (24 currently)
        bnds = 2 ;
variables:
        double lon(lon) ;
                lon:standard_name = "longitude" ;
                lon:units = "degrees_east" ;
        double lat(lat) ;
                lat:standard_name = "latitude" ;
                lat:units = "degrees_north" ;
        double time(time) ;
                time:standard_name = "time" ;
                time:units = "days since 2001-1-1" ;
                time:original_units = "seconds since 2001-1-1" ;
        double time_bnds(time, bnds) ;
        float tos(time, lat, lon) ;
                tos:standard_name = "sea_surface_temperature" ;
                tos:_FillValue = 1.e+20f ;
                tos:missing_value = 1.e+20f ;

// global attributes:
                :title = "Sea Temp" ;
                :realization = 1 ;
                :cmor_version = 0.96f ;
                :comment = "Test drive" ;
data:

% cat schema.sql

DROP TABLE sampleGrid;
```

```
CREATE TABLE sampleGrid(
  chip dschip
);

% psql demo -U demo  < schema.sql
ERROR:  table "samplegrid" does not exist
CREATE TABLE

% cat test.rc
#
# The DSChip column gets values with inernal columns of time,
lat, lon and tos.
#
sampleGrid=time:1;1,lat:5;0.5,lon:4;1;,tos;0.1

% ./ncToDBXten -d demo -u demo -i test.nc -rc test.rc

% psql demo -U demo

demo=> SELECT count(*) FROM sampleGrid;
count
-------
 36720
(1 row)

demo=> SELECT * FROM sampleGrid LIMIT 1;

                            chip
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------

maxtuples=20,filledtuples=20,numcolumns=4;time,float,1;lat,float
,0.5;lon,float,1;tos,float,0.1;15,-79.5,1,100.0;15,-79.5,4,101.0
;15,-79.5,5,103.0;15,-79.5,7,106.0;15,-78.5,1,100000002004087734
272.0;15,-78.5,4,100000002004087734272.0;15,-78.5,5,100000002004
087734272.0;15,-78.5,7,100000002004087734272.0;15,-77.5,1,100000
002004087734272.0;15,-77.5,4,100000002004087734272.0;15,-77.5,5,
100000002004087734272.0;15,-77.5,7,100000002004087734272.0;15,-7
6.5,1,100000002004087734272.0;15,-76.5,4,100000002004087734272.0
;15,-76.5,5,100000002004087734272.0;15,-76.5,7,10000000200408773
4272.0;15,-75.5,1,100000002004087734272.0;15,-75.5,4,10000000200
4087734272.0;15,-75.5,5,100000002004087734272.0;15,75.5,7,100000
002004087734272.0
(1 row)

demo=>
```

Appendix G:
CSV File Reordering Utility

Overview

As described earlier in Indexing DSChip's (see page 35), retrieval performance in DBXten is improved when tuples are pre-sorted / loaded into DSChip's in such a way that the number of DSChip's that contain a particular range of values is minimized. This reduces the number of DSChip's returned by the relatively fast "Region of Interest" filtering (see page 51), leaving less to be done by the relatively slow "Tuple Filtering" (see page 52). This pre-sorting can also enhance the compression potential, as adjacent rows, after sorting, can be closer to each other in value than if the sorting had not been done.

Using the CSV File Reordering Utility

The pre-sorting described in the previous paragraph can be performed by the CSV File Reordering Utility, which is distributed with DBXten (and stored in `<DBXTENDIR>/bin/reorderCsv`)

The utility is called as follows:

```
Usage:  reorderCsv options < input.csv > output.csv

where options are:
   -p keyColumn:size (key,size > 0)
  -rowsPerChip size  (default 1000, -chipRows is an alias)
```

The '`-p`' argument identifies a column from the csv file (the leftmost column is 1, next one to the right is 2, …) and a blocking factor to apply to that column. The "`-rowsPerChip`" argument indicates the number of CSV file lines that will go into each DSChip when the file is eventually loaded.

The reorderCsv utility uses the following process to perform its reordering:

1) Read the first $R \times BF_1 \times BF_2 \times \ldots \times BF_n$ rows, there R is the `-rowsPerChip` value and BF_i is the blocking factor for column i.
2) Let j be the first column with a blocking factor specified. Partition the rows read into BFj groups so that all the column j values in the first group are ≤ all the column j values in the second group, all the column j values in the second group are ≤ all the column j values in the third group, etc.
3) Repeat step 2, partitioning each of the partition groups using the next column and the blocking factor for the next column.
4) Step 3 is repeated for each column with a blocking factor.
5) Sort each of the resulting partition blocks by the columns specified (in the order specified).

Example

We will demonstrate the use of reorderCsv using the following simple CSV file (`test.csv`):

```
1,24
2,22
3,20
4,18
5,16
6,14
7,12
8,10
9,8
10,6
11,4
12,2
13,1
14,3
15,5
16,7
17,9
18,11
19,13
20,15
21,17
22,19
23,21
24,23
```

and the following command:

```
reorderCsv -p 1:2 -p 2:3 -rowsPerChip 4 < test.csv
```

The `reorderCsv` program first reads in all 24 lines, since $R \times BF_1 \times BF_2 = 4 \times 2 \times 3 = 24$.

The program then sorts the read-in data by the first column and then partitions the data into 2 (BF1) groups of 3×4 = 12 lines, so that everything in column 1 of the first group is less than or equal to everything in column 1 of the second group. This is already the case, so nothing is done at this stage.

The next step is to repeat the process with the second column, sorting and partitioning each of the 2 blocks produced in the last stage. But this time the program partitions the 2 blocks into 3 blocks each (since $BF_2 = 3$).

Finally, the program sorts each of the $BF_1 \times BF_2 = 2 \times 3 = 6$ blocks by column 1, then column 2.

Each of these stages is illustrated by a column in the following table:

Initial	Sort	Partition	Sort	Partition	Sort	**Final**
1,24	1,24	1,24	12,2	12,2	9,8	**9,8**
2,22	2,22	2,22	11,4	11,4	10,6	**10,6**
3,20	3,20	3,20	10,6	10,6	11,4	**11,4**
4,18	4,18	4,18	9,8	9,8	12,2	**12,2**
5,16	5,16	5,16	8,10	8,10	5,16	**5,16**
6,14	6,14	6,14	7,12	7,12	6,14	**6,14**
7,12	7,12	7,12	6,14	6,14	7,12	**7,12**
8,10	8,10	8,10	5,16	5,16	8,10	**8,10**
9,8	9,8	9,8	4,18	4,18	1,24	**1,24**
10,6	10,6	10,6	3,20	3,20	2,22	**2,22**
11,4	11,4	11,4	2,22	2,22	3,20	**3,20**
12,2	12,2	12,2	1,24	1,24	4,18	**4,18**
13,1	13,1	13,1	13,1	13,1	13,1	**13,1**
14,3	14,3	14,3	14,3	14,3	14,3	**14,3**
15,5	15,5	15,5	15,5	15,5	15,5	**15,5**
16,7	16,7	16,7	16,7	16,7	16,7	**16,7**
17,9	17,9	17,9	17,9	17,9	17,9	**17,9**
18,11	18,11	18,11	18,11	18,11	18,11	**18,11**
19,13	19,13	19,13	19,13	19,13	19,13	**19,13**
20,15	20,15	20,15	20,15	20,15	20,15	**20,15**
21,17	21,17	21,17	21,17	21,17	21,17	**21,17**
22,19	22,19	22,19	22,19	22,19	22,19	**22,19**
23,21	23,21	23,21	23,21	23,21	23,21	**23,21**
24,23	24,23	24,23	24,23	24,23	24,23	**24,23**

Appendix H: Tile Optimization Utility

Overview

Data tiling with DBXten has two purposes:

1) to improve the locality of data, thereby speeding up data extractions, and
2) to improve the locality and ordering, thereby improving compressibility.

The Tile Optimization Utility, `tileOptimizer`, is designed to assist you in picking tile sizes that balance these two purposes. When `tileOptimizer` is provided a dataset (in the form of a CSV file), it generates a list of possible tilings and determines how much space the dataset would take with each tiling. The list is sorted in order of ascending size. The intent is that you can look down the list until you find a tiling that is roughly consistent with your data extraction needs.

The `tileOptimizer` utility is distributed with DBXten (and stored in `<DBXTENDIR>bin/tileOptimizer`)

Usage

`tileOptimizer` accepts the following arguments:

`[-indelimit` *`delimiter_character`*`]`	Optional delimiter character, default is '|'. 'tab' can be used to indicate tab as the delimiter.
`-i` *`input_file`*	Input file name (mandatory).
`[-verbose]`	Do you want verbose output? If so, messages are written to `stderr`, indicating what tiling is currently being tried.
`[-csv]`	This argument causes the output to be in the form of a CSV file, suitable for importing into a spreadsheet and sorting according to your own criteria.
`(`*`dimension`*`|`*`field`*`)+`	Mandatory, see below.

"`(`*`dimension`*`|`*`field`*`)+`" indicates one or more occurrences of a *dimension* or *field* argument. The *dimension* and *field* arguments have the following forms, respectively:

```
dimension_name:length:datatype[:precision]
field_name:datatype[:precision]
```

These arguments must be listed in the order that columns appear in the input fle. The `length` term is an integer that denotes the number of samples in that direction. The `precision` term specifies the precision to which the column must be stored. The `datatype` term is one of "`integer`", "`int8`", "`double`", "`datetime`", or "`string`". The primary difference between the `dimension` and `field_name` arguments is that `tileOptimizer` will sort the input rows on the basis of the value of `dimension` columns but not `field` columns.

Limitations

`tileOptimizer` keeps a copy of the entire input file in memory, which limits the size of input file it can deal with. If your file is over a 100MB, consider partitioning it into smaller equally sized pieces, and invoke the tileOptimizer on one or more of the pieces.

`tileOptimizer` will only try tile sizes that divide evenly into the datasets dimension sizes. For example, a tile size of 33×50 would divide evenly into a

dataset of size 99×100, but not into a dataset of size 100×100. If a dataset's dimensions can not be factored, there will fewer tile sizes tried. For example, a 1511×1511 grid has only two possible tilings: 1511×1 and 1×1511.

`tileOptimizer` will also only try tiles that result in per-chip row counts between 500 and 15000.

Example usage

Example

Consider a CSV file representing the contents of an image 900 pixels wide, 1200 pixels high, with the following integer columns:

`column-index`, row-index, red-value, green-value, blue-value

as show below:

```
0,0,143,157,183
1,0,146,160,186
2,0,147,163,188
     …
897,1199,35,39,40
898,1199,37,39,38
899,1199,41,41,39
```

The following command would produce a list of 776 tile sizes. (Note that half the combinations derive from treating the second column, instead of the first, as the primary key.)

```
tileOptimizer –indelimit , -i clouds.csv x:900:integer \
y:1200:integer  r:integer  g:integer b:integer
```

Here is some sample output from such a file:

```
Size =3267300, Tiling:  y:1200 x:5
Size =3269128, Tiling:  y:400 x:18
Size =3274590, Tiling:  x:90 y:80
…
Size =3632528, Tiling:  x:10 y:1200
Size =3640375, Tiling:  x:12 y:1200
```

Using tileOptimizer in conjunction with reorderCsv and csvChipLoader

As described in Appendix G (see page 115), the arguments expected by `reorderCsv` are: the number of rows per chip and blocking factors. The number of rows per chip is simply the product of the tile dimensions. So for a tile of size "x:90 y:80", the rows per chip would be 7200. The blocking factor for each dimension is the size of dataset dimension divided by the size of the corresponding tile dimension. In the case of the 900×1200 pixel image used in

the example above, that would be 900/90 = 10, and 1200/80 = 15. The `reorderCsv` utility would be called as follows:

```
reorderCsv –rowsPerChip 7200 -p 1:10 -p 2:15 < clouds.csv > \
tiledClouds.csv
```

The csvChipLoader utility would be called as follows:

```
csvChipLoader –d dbname –t tablename -rowsPerChip 7200 \
-i tiledClouds.csv -indelimit  ,  x:integer  y:integer \
r:integer  g:integer  b:integer
```

If `tileOptimizer` listed the dimensions in the other order, y:80 x:90, the –p arguments to `reorderCsv` would need to be swapped, as below:

```
reorderCsv -rowsPerChip 7200 -p 2:15 -p 1:10 < clouds.csv > \
tiledClouds.csv
```