

BARRODALE COMPUTING SERVICES LTD.

---

# DBXten DataBlade for IBM Informix (Linux Version)



# Programmer's Guide

Version 1.10.0.2, February 4, 2011



# **DBXten DataBlade for IBM Informix (Linux Version) Programmer's Guide**

---

© Barrodale Computing Services Ltd.

<http://www.barrodale.com>

---



# Table of Contents

Chapter 1: Tuple Data and the BCS DBXten DataBlade – An	
Introduction .....	1
Storage and Representation of Tuple Data .....	1
“Strong” Entities and “Weak” Entities .....	2
Representation of Parent and Child Entities in a Database..	3
Some Queries Involving Parents and Children.....	4
Issues When <i>N</i> Becomes Large .....	5
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 DataBlade.....	19
Installing the Software.....	19
Setting up the License Key .....	20
Building Sample Programs and Testing the Installation .....	21
Determining the DBXten Software Version Number .....	22
Chapter 3: The BCS DBXten DataBlade – the DSChip and	
DSBox Data Types.....	25
The DSChip Data Type .....	25
Units Support.....	28
Data Types that are Supported Inside a DSChip .....	28
The DSBox Data Type .....	28
Chapter 4: Getting Data into DSChip’s .....	31
Using the DBXten SQL API to Insert Data .....	31
Creating an Empty DSChip .....	31
Adding Tuples to a DSChip .....	33

Indexing DSChip's.....	36
Other Functions.....	39
Using a Utility Loader to Insert Data .....	39
Loading into a VTI Table .....	40
Steps in Defining and Loading a Virtual Table .....	41
Create the Base Table.....	42
Define the VTI Virtual Table.....	42
Load the VTI Virtual Table .....	43
Performance Penalty in Using VTI for Loads .....	43
VTI Limitations.....	44
Using the DBXten C API to Insert Data.....	45
General Structure of Loading Programs.....	45
A Sample ESQL-C DSChip Loading Program.....	45
Using the DBXten Java API to Insert Data.....	48
A Sample Java DSChip Loading Program .....	48
Chapter 5: Retrieving Data from DSChip's .....	51
Options for Extracting Data .....	52
Other Processing Paths .....	54
Using the DBXten Native SQL API to Extract Data.....	55
Determining What Columns are in a DSChip .....	56
"Unwrapping" DSChip's.....	56
Listing Distinct Values for DSChip Column(s) .....	58
Determining Whether a DSChip has a Particular Column(s).....	61
Determining How Many Columns a DSChip has .....	61
Determining the Number of Tuples in a DSChip .....	62
Determining the Maximum Value for Columns in a DSChip.....	62
Determining the Minimum Value for Columns in a DSChip.....	63
Listing Compression Information for a DSChip.....	63
Converting a DSChip Range to a DSBox.....	64
Generating a Bounding DSBox from a DSChip.....	64
Doing a Region of Interest Query to Select DSChip's .....	65
Extracting Tuples from a DSChip (no tuple filtering).....	67
Counting Tuple Matches without Extracting .....	70
Tuple Filtering DSChip's into New DSChip's.....	71
Tuple Filtering DSChip's into a Set of Tuples.....	73
Using the DBXten C API to Extract Data.....	74
Performing Tuple Filtering on the Client.....	74
Performing Tuple Filtering on the Server.....	77

Using the DBXten Java API to Extract Data.....	78
Using the DBXten Virtual Table Interface to Extract Data .....	81
Steps in Defining a Virtual Table .....	83
Create and Load the Base Table.....	83
Create Indexes on the Base Table .....	83
Defining a VTI Virtual Table for Queries on non-DSChip Columns (e.g., C) .....	84
Defining a VTI Virtual Table for Queries on DSChip Extents (e.g. X/Y/Z/T).....	85
B-Tree Indexes on DSChip Dimension Columns .....	87
Using VTI - Current Limitations.....	89
Chapter 6: Updating Data in the Database .....	91
Chapter 7: Troubleshooting Guide .....	93
Connection Errors .....	93
C Client Library Errors.....	93
General Operational Errors .....	94
Corrupt or Misinterpreted Binary Data.....	95
Bad ASCII Data.....	95
Bad DateTime Data.....	98
DSQueryToString Specific Errors .....	101
Appendix A: Complete List of BCS DBXten DataBlade User- Defined Routines (UDRs).....	103
Appendix B: The C API.....	105
List of C API Constants .....	105
Size Limits.....	105
Supported Base Data Types.....	105
DBXten C Data Types .....	105
List of C API Functions.....	105
DSChip-specific Functions.....	106
DSChip Column-specific Functions .....	107
Standard Support Functions.....	108
Appendix C: The Java API.....	111
Appendix D: A Tutorial .....	113
Appendix E: CSV File Reader Utility.....	115
Downloading the CSV File Reader Utility .....	115
Compiling the CSV File Reader Utility .....	116
Running the CSV File Reader Utility .....	116
Appendix F: NetCDF File Reader Utility .....	119
Overview .....	119

The Run-Control File .....	119
The Overall Syntax .....	119
Dimension Syntax .....	120
Grid Variable Syntax .....	120
Comments and Whitespace .....	120
Downloading the NetCDF File Reader Program .....	120
Compiling .....	121
Running the NetCDF File Reader Utility .....	121
Appendix G: CSV File Reordering Utility .....	124
Overview .....	124
Using the CSV File Reordering Utility .....	124
Appendix H: Tile Optimization Utility .....	128
Overview .....	128
Usage .....	129
Limitations .....	129
Example usage .....	130
Using tileOptimizer in conjunction with reorderCsv and csvChipLoader .....	130

# List of Figures

Figure 1: Traditional Implementation of a Parent-Child Entity Relationship.....	4
Figure 2: Alternative Implementation of a Parent-Child Entity Relationship. ....	6
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.....	32
Figure 13: Two-dimensional DSChip's. ....	38
Figure 14: VTI Table Sample Timings.....	44
Figure 15: DSChip Extraction Processing paths.....	52
Figure 16: DSChip's A and B overlap the specified X and Y ranges. ....	53
Figure 17: Syntax of a <code>columnNames</code> parameter. ....	55
Figure 18: Syntax of a <code>rangeSpec</code> parameter. ....	55



# 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.
<i>italics</i>	<i>New terms, emphasized words, and variable values appear in italics.</i>
<i>italics</i>	
<i>italics</i>	
User input	Computer generated text (e.g., error messages) and user input appear in a non-proportional font.
<INFORMIXDIR>	The directory where the IBM Informix server was installed. By default, this is /opt/IBM/Informix.
<DBXTENDIR>	The directory where the DBXten DataBlade is installed. By default, this is <INFORMIXDIR>/extend/DBXten.n.n.n
(ehportsopa) 's	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<sup>1</sup>:

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.

---

<sup>1</sup> This manual follows the icon conventions used in IBM Informix manuals.

## What's New in This Version?

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

<b>Feature</b>	<b>Manual Sections Where Feature is Described.</b>
Tile Optimization utility	<a href="#">Appendix H</a> (page 128)
CSV File Reordering utility	<a href="#">Appendix G</a> (page 124)
Utility executables now supplied with DBXten	Appendices <a href="#">E</a> , <a href="#">F</a> , <a href="#">G</a> , <a href="#">H</a>

**BCS DBXTEN DATA BLADE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**



# Chapter 1: Tuple Data and the BCS DBXten DataBlade – 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<sup>2</sup> 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

<sup>2</sup> 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.

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 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:

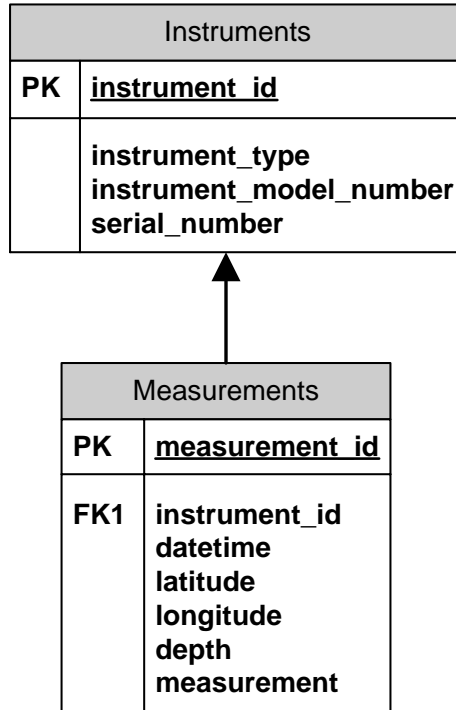


Figure 1: Traditional Implementation of a Parent-Child Entity Relationship<sup>3</sup>.

```
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 DATETIME YEAR TO SECOND,
                           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.

---

<sup>3</sup> 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.

- 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.

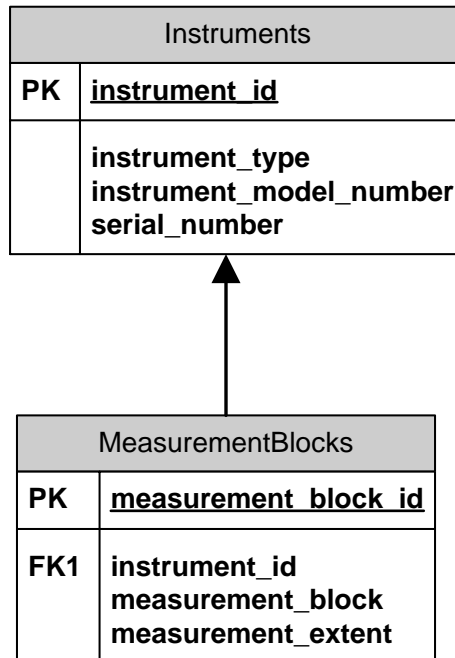


Figure 2: Alternative Implementation of a Parent-Child Entity Relationship.

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	
	<b>datetime</b> <b>latitude</b> <b>longitude</b> <b>depth</b> <b>measurement</b>

Figure 3: Looking Inside a Tuple Block.

A sequence of these measurement blocks may look like the following<sup>4</sup>:

	<b>Datetime</b>	<b>Latitude</b>	<b>Longitude</b>	<b>Depth</b>	<b>Measurement</b>
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.

<sup>4</sup> 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<sup>5</sup>:

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);
```

---

<sup>5</sup> 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](#)<sup>6</sup>. 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

---

<sup>6</sup> 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](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

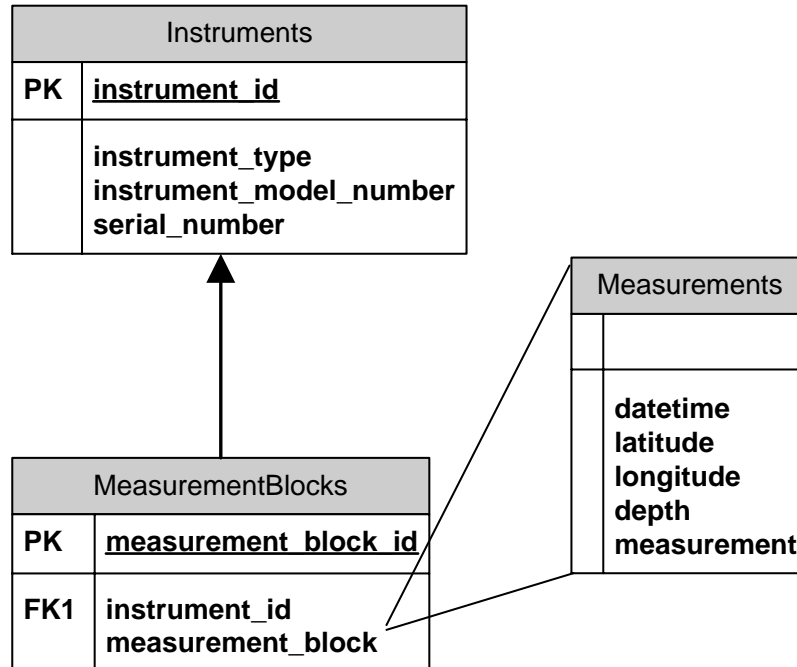


Figure 6: Instrument Measurements application schema.

## Airline Flight

### SCHEMA

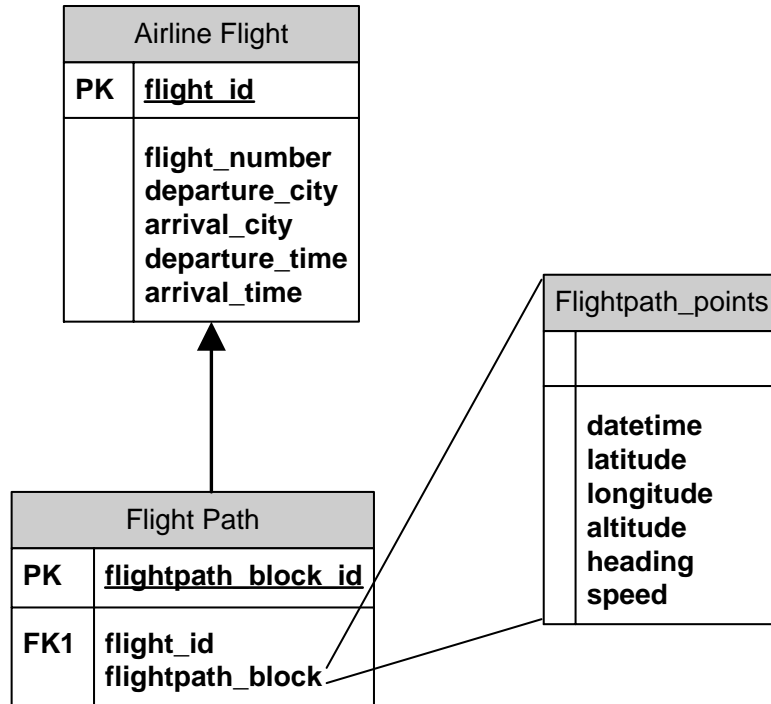


Figure 7: Airline Flight application schema.

## GIS Object

### SCHEMA

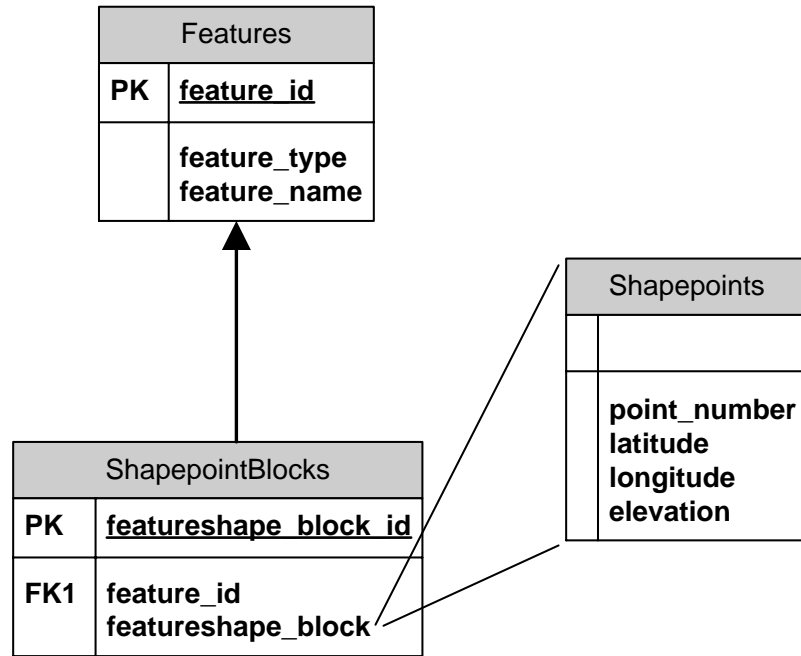


Figure 8: GIS Object application schema.

## Stock Market

### SCHEMA

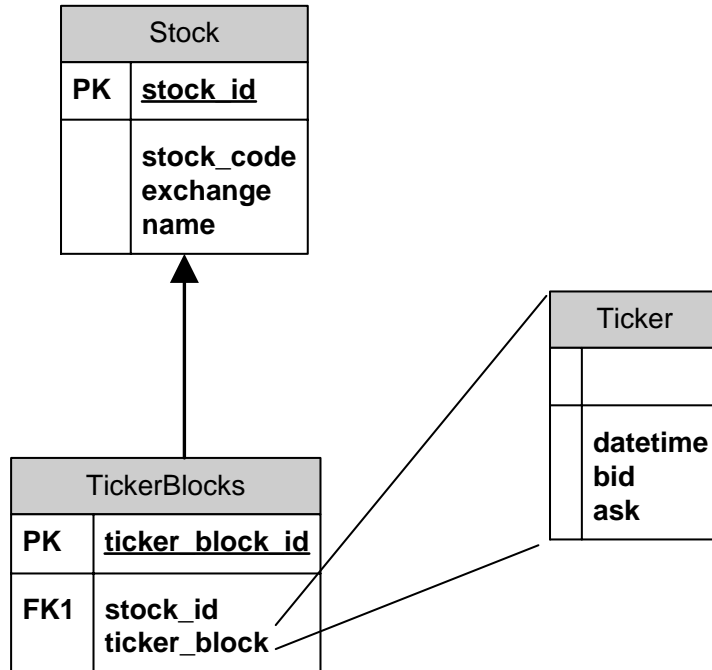


Figure 9: Stock Market application schema.

## Delivery Tracking

### SCHEMA

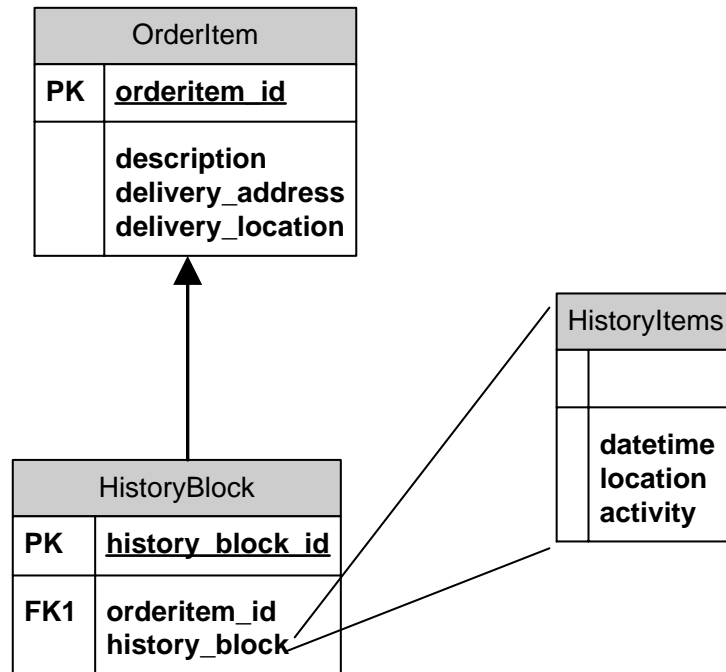


Figure 10: Delivery Tracking application schema.

## Inventory

### SCHEMA

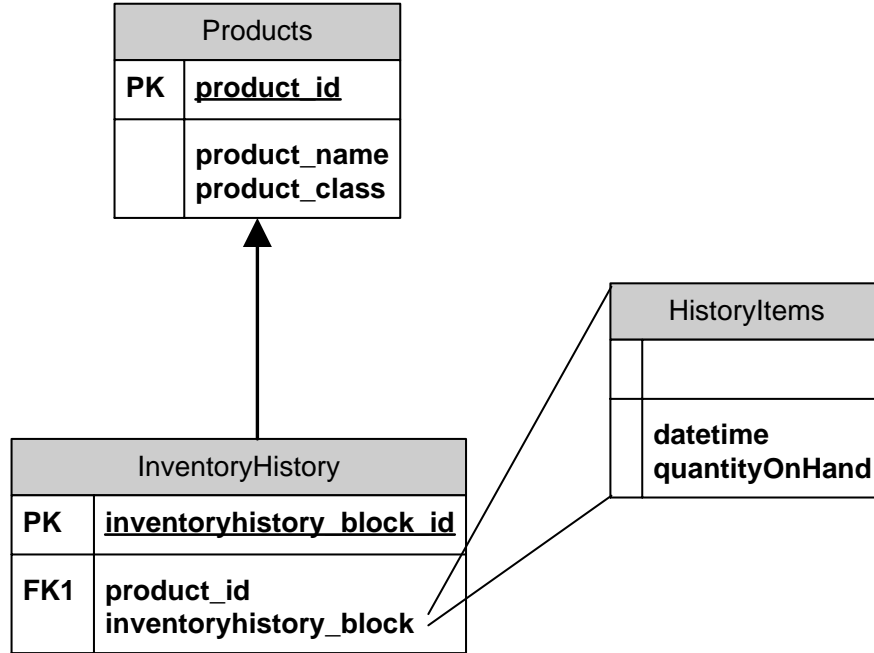


Figure 11: Inventory application schema.

## 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 DataBlade 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 DataBlade

This chapter describes how to install the BCS (Barrodale Computing Services) DBXten DataBlade software onto a Linux server machine and perform some simple operations to test the installation. This section assumes that

1. you have an IBM Informix installation already available,
2. a database into which the DBXten DataBlade is to be installed has been created, and
3. the environment variable `$INFORMIXDIR` is set to the Informix home directory, and
4. the environment variable `$ONCONFIG` is set to the name of the INFORMIX configuration file.

### Installing the Software

DBXten is packaged as a zip file, with a name of `DBXten_versionNumber.zip`. The following instructions should be executed either as user “informix” or as some other user who has been granted administrative (DBSA) privileges.

1. cd into directory `$INFORMIXDIR/extend`

```
$ cd $INFORMIXDIR/extend
```

2. Unzip the file.

```
$ unzip DBXten_versionNumber.zip
```

3. If the DataBlade is to be installed by another user, in particular a user who does not have DBSA privileges, then you must either:

- a. Modify `$INFORMIXDIR/etc/$ONCONFIG` to set the parameter `IFX_EXTEND_ROLE` to 0.

1. or

- b. Grant the “extend” role to the Informix user who will be registering the DBXten DataBlade into a database.

```
> GRANT extend TO SomeInformixUser;
```

4. Set up the license key as described in the [next section](#) (Setting up the License Key).
5. Register the DBXten dataBlade into the appropriate database (*dbname* in the following example) using `blademgr`:

```
% blademgr
SERVERNAME>list dbname
There are no modules registered in database dbname.
SERVERNAME>show modules
9 DataBlade modules installed on server SERVERNAME:
      wfs.1.00.UC1          binaryudt.1.0
      LLD.1.20.UC2          mqblade.2.0
      Node.2.0 c           ifxbuiltins.1.1
      DBXten.1.4.0         ifxrltree.2.00
      bts.1.10
A 'c' indicates DataBlade module has client files.
If a module does not show up, check the prepare log.
SERVERNAME>register ifxrltree.2.00 dbname
Register module ifxrltree.2.00 into database dbname?
[Y/n]Y
Registering DataBlade module... (may take a while).
DataBlade ifxrltree.2.00 was successfully registered in
database dbname.
SERVERNAME>register DBXten.1.4.0 dbname
Register module DBXten.1.4.0 into database dbname? [Y/n]Y
Registering DataBlade module... (may take a while).
DataBlade DBXten.1.4.0 was successfully registered in
database dbname.
SERVERNAME>
```

6. “Make” the examples and test the installation as described below in “Building Sample Programs and Testing the Installation”.

## Setting up the License Key

Unless you are using an evaluation version of the DBXten DataBlade, a license key is needed for each computer on which the IBM Informix server runs<sup>7</sup>. The

---

<sup>7</sup> Evaluation versions are time-limited and do not require a license key.

license key is provided to the BCS DBXten DataBlade by adding the following lines to the `.bashrc` file in the `informix` account<sup>8</sup>:

```
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 IBM Informix server. A simple way to do this is to log into the `informix` account and execute the following commands (assuming that the `informix` account uses the bash shell):

```
oninit > oninit.out 2>&1 &
```

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.

## Building Sample Programs and Testing the Installation

Note that the sample programs assume that the `userid` and `database` to be used are set with the environment variables `$DBXTEN_DEMO_USERID` and `$DBXTEN_DEMO_DATABASE`, respectively.

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

```
$ cd <DBXTENendir>/examples/sql
```

2. Create the `instruments`, `measurements`, and `measurementBlocks` tables used in the examples in this manual.

```
$ dbaccess -e $DBXTEN_DEMO_DATABASE \  
chapter2/sql/measurements.sql  
$ dbaccess -e $DBXTEN_DEMO_DATABASE \  
chapter2/sql/measurements_chapter1.sql
```

---

<sup>8</sup> If the `informix` 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 `tsh` is used, then “`setenv DBXTEN_LICENSE_KEY license_key_value`” will need to be placed in the `.cshrc` file in the `informix` account.

3. Create and populate the `xyvals` table.

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

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

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

5. Build the C executable programs. The first `make` command also runs the `loadTable` example and tests the results of inserting and fetching DSChip's from the database.

```
$ make  
$ make fetch1  
$ make fetch2
```

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

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

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

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

8. Build the class files (update Makefile as necessary).

```
$ make
```

9. Execute the loading class.

```
$ make runLoad
```

10. 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:

```
> EXECUTE FUNCTION DSGetVersion();
```

**BCS DBXTEN DATA BLADE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

(expression)

1.5.0.0

1 row(s) retrieved.

>



## Chapter 3: The BCS DBXten DataBlade – the DSChip and DSBox Data Types

This chapter discusses the representation of tuple blocks inside the BCS DBXten DataBlade. In particular, it defines the DSChip data type, which is the data type used by DBXten to store a block of tuples, and the DSBox datatype, which is used for creating multidimensional indexes on DSChip's.

### 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<sup>9</sup>:

---

<sup>9</sup> 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.

**BCS DBXTEN DATABLED FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

```
> SELECT measurement_block_id, instrument_id, measurement_block  
FROM measurements;
```

```
measurement_block+ 1  
instrument_id      1  
measurement_block  maxtuples=100, filledtuples=7, numcolumns=5; dat  
etime, datetime, 10; latitude, float, 0.001; longit  
ude, float, 0.001; depth, float, 0.01; measurement,  
float, 0.1; 2008-02-01 00:12:20, 46.343, -  
127.386, 14.34, 10.2; 2008-02-01  
00:12:30, 46.344, -127.385, 16.82, 11.9; 2008-02-  
01 00:12:30, 46.345, -127.383, 18.85, 10.7; 2008-  
02-01 00:12:30, 46.346, -127.382,  
21.22, 11.7; 2008-02-01 00:12:30, 46.347, -  
127.381, 23.66, 10.9; 2008-02-01  
00:12:30, 46.349, -127.3  
79, 26.05, 11.4; 2008-02-01 00:13:40, 46.392, -  
127.335, 104.82, 10.7
```

```
measurement_block+ 2  
instrument_id      1  
measurement_block  maxtuples=100, filledtuples=7, numcolumns=5; dat  
etime, datetime, 10; latitude, float, 0.001; longit  
ud  
e, float, 0.001; depth, float, 0.01; measurement, fl  
oat, 0.1; 2008-02-01 00:13:50, 46.394, -127.334  
, 106.83, 10.7; 2008-02-01 00:13:50, 46.395, -  
127.332, 108.90, 13.1; 2008-02-01  
00:13:50, 46.396, -127.331, 110.99, 10.7; 2008-  
02-01 00:13:50, 46.398, -127.  
330, 113.32, 11.4; 2008-02-01 00:13:50, 46.399, -  
127.328, 115.38, 10.2; 2008-02-01  
00:14:00, 46.400, -127.327, 117.65, 10.9; 2008-  
02-01 00:15:00, 46.439, -127.289, 188.40, 10.3
```

```
measurement_block+ 3  
instrument_id      2  
measurement_block  maxtuples=100, filledtuples=7, numcolumns=5; dat  
etime, datetime, 10; latitude, float, 0.001; longit  
ude, float, 0.001; depth, float, 0.01; measurement  
, float, 0.1; 2008-02-01 00:15:00, 46.441, -  
127.287, 190.88, 9.7; 2008-02-01  
00:15:00, 46.442, -127.286, 193.22, 11.9; 2008-  
02-01 00:15:10, 46.443, -127.285  
, 195.65, 11.5; 2008-02-01 00:15:10, 46.444, -  
127.283, 197.86, 10.5; 2008-02-01 00:15:1  
0, 46.446, -127.282, 200.02, 11.2; 2008-02-01  
00:15:10, 46.447, -127.281, 202.38, 10.7; 2008-  
02-01 00:16:20, 46.486, -127.241, 274.68, 11.3
```

```
3 row(s) retrieved.
```

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

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

## Units Support

DBXten will allow a string to be stored with each numeric<sup>10</sup> 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:

```
mxtuples=100, filledtuples=7, numcolumns=5; datetime, datetime, 10; latitude="degrees_north", float, 0.001; longitude="degrees_east", float, 0.001; depth="meters", float, 0.01; measurement="degC", float, 0.1; 2008-02-01 00:12:20, 46.343, -127.386, 14.34, 10.2; 2008-02-01 00:12:30, 46.344, -127.385, 16.82, 11.9; 2008-02-01 00:12:30, 46.345, -127.383, 18.85, 10.7; 2008-02-01 00:12:30, 46.346, -127.382, 21.22, 11.7; 2008-02-01 00:12:30, 46.347, -127.381, 23.66, 10.9; 2008-02-01 00:12:30, 46.349, -127.379, 26.05, 11.4; 2008-02-01 00:13:40, 46.392, -127.335, 104.82, 10.7
```

## 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<sup>11</sup> – 64 bit floating point
- 4) string – 8 bit character strings
- 5) date – dates with an optional time component

The *date* type supports basically the same syntax as used by the Informix datetime field. However, note that it is only able to support dates and times between Jan 1, 1902 and Dec 31, 2037.

## The DSBox Data Type

The DSBox data type is used to store the tuple extents for date and/or numeric columns, which, as explained [earlier](#) (see page 9), can be used to index DSChip's.

---

<sup>10</sup> Numeric columns are columns of type integer and float as described in the [“Data Types that are Supported Inside a DSChip”](#) section.

<sup>11</sup> “double” can be used as a synonym for float.

The DSBox data type has the following textual format:

```
'(min1, ..., minn), (max1, ..., maxn)'
```

where min<sub>i</sub> and max<sub>i</sub> refer to the i<sup>th</sup> date or numeric column to be included in the index.

Each min term is either a real number, in which case it is stored as a single precision floating point number, or the letter “D” followed by a real number, in which case it is stored as a double precision floating point number. The precision of the min term determines the precision with which the corresponding max term is stored. The presence or absence of the D in the max term is ignored. By default in DBXten, dates in DSBox’s are stored as double precision numbers and integers and floating point numbers are stored in single precision.

A DSBox can store the minimum and maximum values for the equivalent of up to six numeric columns. In other words, the following combinations are possible:

- 1) 6 integer or floating point dimensions, or
- 2) 1 date dimension and a total of 4 integer/floating point dimensions, or
- 3) 2 date dimensions and a total of 2 integer/floating point dimensions, or
- 4) 3 date dimensions and no other dimensions.

The [DSRangeToBox](#) and [DSAsBoxString](#) functions will automatically store a date as a double precision dimension and any other numeric as a single precision dimension, as show by the examples below.

```
> EXECUTE FUNCTION DSRangeToBox('datetime 2008-02-01 00:13:00  
2008-02-01 00:14:00, latitude 3 4, longitude 6 8');  
  
(expression) (D 1201853580.000000,3.000000,6.000000), (D  
1201853640.000000,4.000000,8.000000)  
  
1 row(s) retrieved.
```

**BCS DBXTEN DATABLADE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

```
> SELECT measurement_block_cube(measurement_block) FROM  
    measurements;
```

```
(expression) (D1201853535,46.3425,-  
127.386), (D1201853625,46.3925,-127.335)
```

```
(expression) (D1201853625,46.3935,-  
127.335), (D1201853705,46.4395,-127.288)
```

```
(expression) (D1201853695,46.4405,-  
127.287), (D1201853785,46.4865,-127.241)
```

## 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:

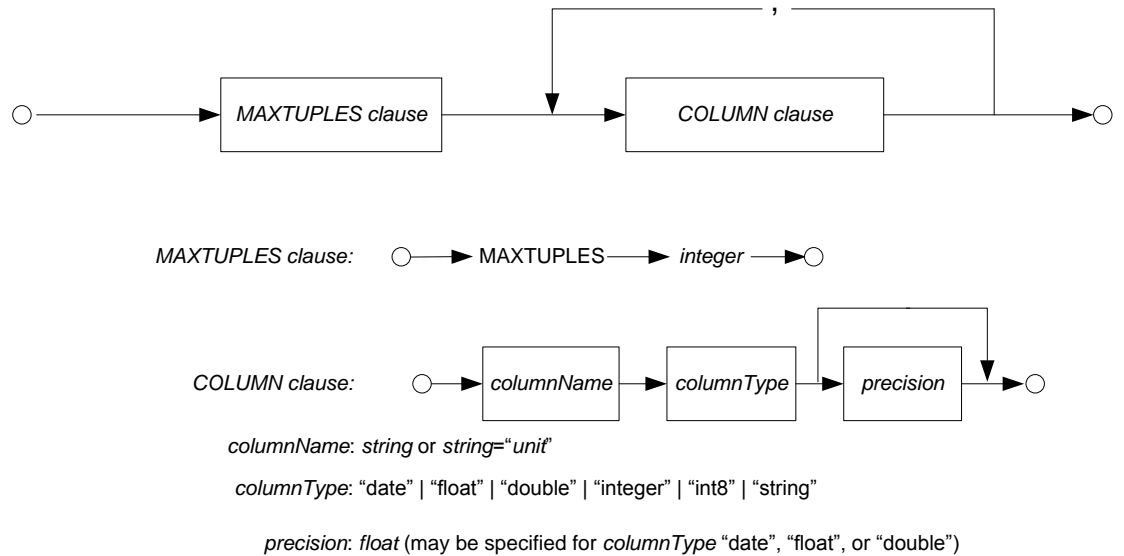


Figure 12: Syntax of a DSChip schema.



The precision value, which can optionally be specified for dates and floating point numbers (float and 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**

```
> EXECUTE FUNCTION DSChipNew('maxtuples 200,datetime date
1,latitude float 0.001,longitude float 0.001,intColumn
integer');

(expression)  maxtuples=200, filledtuples=0, numcolumns=4;datetime,
              datetime,1;latitude,float,0.001;longitude,float,0.0
              01; intColumn,integer,0

1 row(s) retrieved.

> CREATE TABLE mytable(chipcol DSChip);
Table created.

> INSERT INTO mytable VALUES(DSChipNew('maxtuples 200,datetime
date 1,latitude float 0.001,longitude float 0.001,intColumn
integer'));
1 row(s) inserted.

> SELECT * FROM mytable;
```

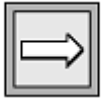
```
chipcol          maxtuples=200, filledtuples=0, numcolumns=4; datetime,  
                datetime, 1; latitude, float, 0.001; longitude, float, 0.0  
                01; intColumn, integer, 0
```

1 row(s) retrieved.

## **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.)

```
> CREATE TABLE instruments (instrument_id integer primary
                             key,
                             instrument_type integer,
                             instrument_model_number
                             varchar(30),
                             serial_number varchar(30));
```

Table created.

```
> INSERT INTO instruments VALUES
    (1,101,'ABC','SerialNum1');
1 row(s) inserted.
```

```
> INSERT INTO instruments VALUES
    (2,102,'DEF','SerialNum2');
1 row(s) inserted.
```

```
> INSERT INTO instruments VALUES
    (3,103,'GHI','SerialNum3');
1 row(s) inserted.
```

```
> INSERT INTO instruments VALUES
    (4,104,'JKL','SerialNum4');
1 row(s) inserted.
```

Create the child table (measurementBlocks).

```
> CREATE TABLE measurementBlocks (measurement_block_id
                                   serial primary key,
                                   instrument_id integer,
                                   measurement_block DSChip);
```

Table created.

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.

```
> 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') from systables where
    tabid = 1;
1 row(s) inserted.
```

Insert tuples into the DSChip.

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

> SELECT * FROM measurementBlocks;

measurement_block+ 1
instrument_id       1
measurement_block   maxtuples=100, filledtuples=2, numcolumns
                    =4; datetime, datetime, 10; latitude, float
                    , 0.001; longitude, float, 0.001; intColumn
                    n, integer, 0; 2008-01-01 12:30:30, 49.700, -
                    127.500, 45;

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

> SELECT * FROM measurementBlocks;

measurement_block+ 1
instrument_id       1
measurement_block   maxtuples=100, filledtuples=2, numcolumns
                    =4; datetime, datetime, 10; latitude, float
                    , 0.001; longitude, float, 0.001; intColumn
                    n, integer, 0; 2008-01-01 12:30:30, 49.700
                    , -127.500, 45; 2008-01-01 12:30:40, 49.5
                    12, -127.724, 24
```

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).

```
> INSERT INTO measurementBlocks(
        measurement_block_id,
        instrument_id,
        measurement_block)
VALUES (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')));

1 row(s) inserted.
```

## **Indexing DSChip's**

Indexes can be created on a DSChip by first casting the DSChip to a DSBox ( $n$ -dimensional box) data type and then indexing the DSBox with an R-tree index.

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

```
FUNCTION DSAsBoxString (existingChip DSChip,
        columnSpec CHAR) RETURNS lvarchar
```

This function returns the text form of an  $n$ -dimensional DSBox built on the  $n$  columns listed in the *columnSpec* parameter. The *columnSpec* parameter has the structure shown in Figure 12 (page 32).

R-tree indexes can be built using SQL similar to the following:

```
> CREATE FUNCTION measurementBlockBox(chipIn DSChip)
    RETURNING DSBox WITH (not variant)
> RETURN DSAsBoxString(chipIn,
    'datetime,latitude,longitude')::DSBox;
> END FUNCTION;
```

Routine created.

```
> CREATE INDEX measurementBlock_idx ON
    measurementBlocks(measurementBlockBox(
        measurement_block) dsbox_ops)
    USING rtree;
```

Index created.

In the SQL above, a functional index is used. It's not possible to use `DSAsBoxString` directly in this functional index, since it takes a literal string as its second argument, and in Informix it is not possible to create a functional index involving a literal string. So a second function – `measurementBlockBox` – is created to hide this literal string, and the functional index is built instead on that second function.

Note that the choice of columns to include in the `columns` parameter for `DSAsBoxString` depends on the sorts of queries that are envisioned. The goal of using the R-tree 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`.

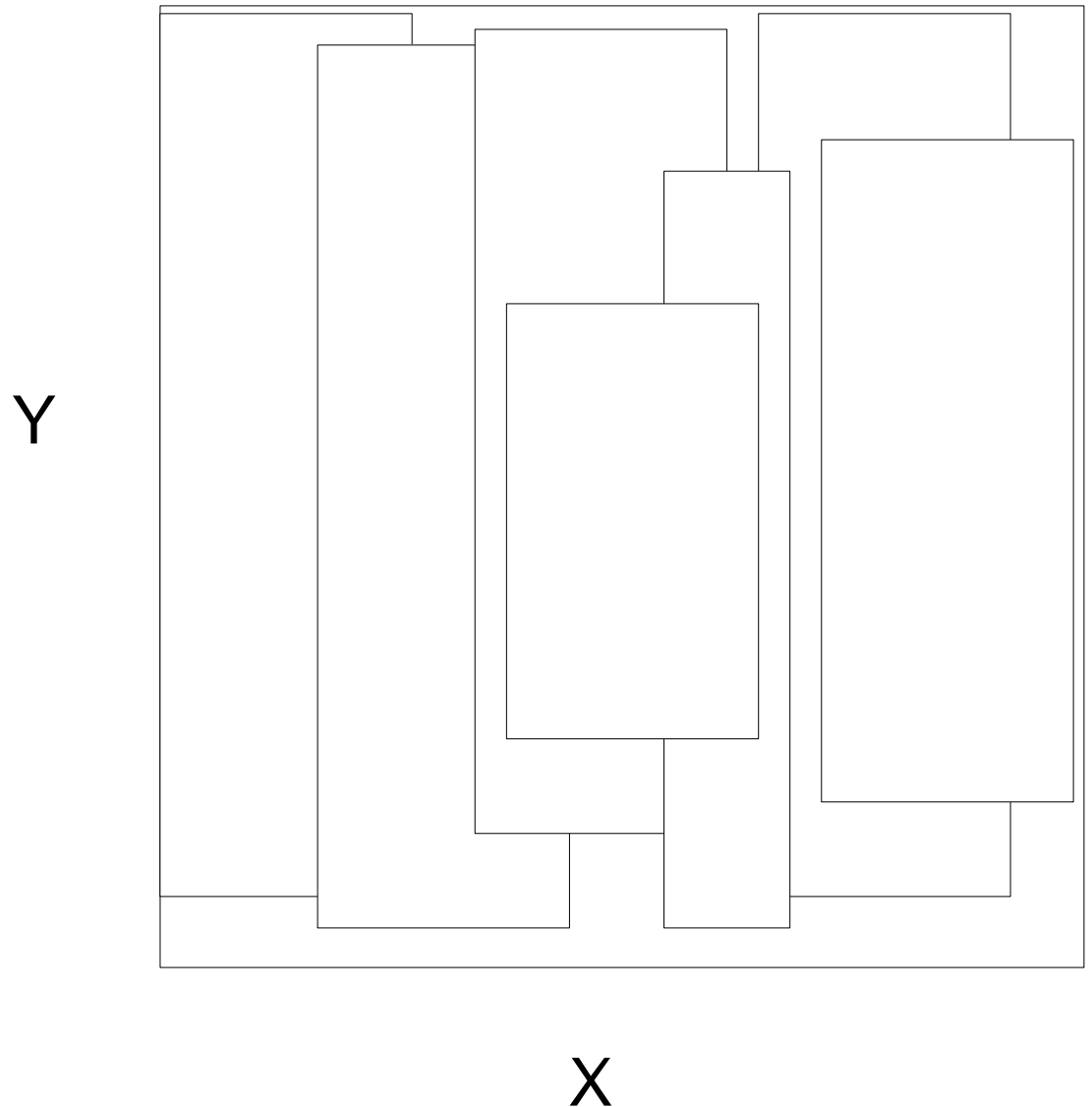


Figure 13: Two-dimensional DSChip's.

In this example, “Y” is *not* a good candidate for a R-tree 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<sup>12</sup>.

---

<sup>12</sup> Note that the CSV File Reordering Utility described in [Appendix G](#) can be used re-cluster records, potentially making Y a better candidate for indexing.

## **Other Functions**

FUNCTION DSChipRecv(*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 section “A Sample Java DSChip Loading Program” (page 48).

FUNCTION DSChipIn(*valueAsString* cstring) RETURNS DSChip

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

FUNCTION DSChipIn(*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<sup>13</sup> to separate the columns. This utility is described in [Appendix E](#) on page 115.

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 119.

---

<sup>13</sup> The name *csv*Loader implies that the delimiter is a comma, but this is just the default; it can be any single character.

## Loading into a VTI Table

The section [Using the DBXten Virtual Table Interface to Extract Data](#) later in this manual (page 81) explains how DBXten can be used in conjunction with the Informix Virtual Table Interface. As explained in that section, the Virtual Table Interface (VTI) allows a table with a DSChip column to appear as if the DSChip column (and its internal “columns”) had been replaced with conventional columns. Hence a previously-written application that loads data into a non-DBXten database table can continue to be used, without modification or even recompilation, after the database table has been restructured using DBXten.

As an illustration of VTI, consider the following table called “mytable”.

mytable:

A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>
A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>
...			
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	D <sub>m</sub>

This table has three conventional (e.g., integer, float, varchar, etc.) columns – A, B, C – and one DSChip column D. Suppose that D has five internal columns – X, Y, Z, T, and V, where X, Y, Z, and T are space and time coordinates and V is the value of some property at the point in space and time (X,Y,Z,T).

mytable:

A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>	
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>	
A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>	
...				
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	D <sub>m</sub>	

Inside of D<sub>1</sub>:

X <sub>11</sub>	Y <sub>11</sub>	Z <sub>11</sub>	T <sub>11</sub>	V <sub>11</sub>
X <sub>12</sub>	Y <sub>12</sub>	Z <sub>12</sub>	T <sub>12</sub>	V <sub>12</sub>
X <sub>13</sub>	Y <sub>13</sub>	Z <sub>13</sub>	T <sub>13</sub>	V <sub>13</sub>
...				
X <sub>1n</sub>	Y <sub>1n</sub>	Z <sub>1n</sub>	T <sub>1n</sub>	V <sub>1n</sub>

VTI provides a view<sup>14</sup> of this table, under a different name – we'll call it `mytable_vti`. This table appears to the user as a table of eight conventional columns – A, B, C, X, Y, Z, T, and V:

`mytable_vti`:

A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	X <sub>11</sub>	Y <sub>11</sub>	Z <sub>11</sub>	T <sub>11</sub>	V <sub>11</sub>
A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	X <sub>12</sub>	Y <sub>12</sub>	Z <sub>12</sub>	T <sub>12</sub>	V <sub>12</sub>
...							
A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	X <sub>1n</sub>	Y <sub>1n</sub>	Z <sub>1n</sub>	T <sub>1n</sub>	V <sub>1n</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	X <sub>21</sub>	Y <sub>21</sub>	Z <sub>21</sub>	T <sub>21</sub>	V <sub>21</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	X <sub>22</sub>	Y <sub>22</sub>	Z <sub>22</sub>	T <sub>22</sub>	V <sub>22</sub>
...							
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	X <sub>2n</sub>	Y <sub>2n</sub>	Z <sub>2n</sub>	T <sub>2n</sub>	V <sub>2n</sub>
...							
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	X <sub>m1</sub>	Y <sub>m1</sub>	Z <sub>m1</sub>	T <sub>m1</sub>	V <sub>m1</sub>
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	X <sub>m2</sub>	Y <sub>m2</sub>	Z <sub>m2</sub>	T <sub>m2</sub>	V <sub>m2</sub>
...							
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	X <sub>mn</sub>	Y <sub>mn</sub>	Z <sub>mn</sub>	T <sub>mn</sub>	V <sub>mn</sub>

Loads performed into this new *virtual* table `mytable_vti` (via SQL “load from”, “insert into”, `dbload`, and `HPL`) are automatically rewritten, under the covers of VTI, to load data instead into the conventional and DSChip columns of the *base* table `mytable`.

### **Steps in Defining and Loading a Virtual Table**

This section lists the steps involved in creating a VTI virtual table into which loads can be performed.

---

<sup>14</sup> By “view” here we mean “view” in the general sense. This is not a database view; rather, it is a completely new (but virtual) table, having an access method that pulls data from `mytable`.

### **Create the Base Table**

```
CREATE TABLE mytable (A integer, B float, C date, D DSChip);
```

### **Define the VTI Virtual Table**

Either of the SQL statements in the following examples can be used to create a virtual table based on mytable.

#### **Example**

```
CREATE TABLE tempschemasource(schematext lvarchar);  
INSERT INTO tempschemasource values(  
'(maxtuples 1000,T date,X double .1,Y double .1,Z double .1,  
V double .1)');
```

```
CREATE TABLE mytable_vti_1(  
    A integer,  
    B float,  
    C date,  
    X float,  
    Y float,  
    Z float,  
    T datetime year to second,  
    V float)  
USING DSChipAccess(  
    basetable='mytable',  
    dschipcolumn='D',  
    schemasource='tempschemasource'  
);
```

#### **Example**

```
CREATE TABLE mytable_vti_2(  
    A integer,  
    B float,  
    C date,  
    X float,  
    Y float,  
    Z float,  
    T datetime year to second,  
    V float)  
USING DSChipAccess(  
    basetable='mytable',  
    dschipcolumn='D',  
    schematext='(maxtuples 1000, T date,X double .1,Y double  
.1,Z double .1,V double .1)'  
);
```

Note:

- 1) "basetable" gives the name of the base table upon which this virtual table has been defined. This is a required parameter.
- 2) "dschipcolumn" identifies the DSChip column that is to be accessed by this VTI table. If there is just one DSChip column in the base table then this parameter can be omitted.

- 3) “schematext” indicates the schema of the DSChip. The format of this parameter is the same as the format produced by the [DSChipSchema](#) function (surrounded by ‘ ( and ) ’). Either “schematext” or “schemasource” (described next) must be specified.
- 4) “schemasource” identifies a table containing a single row with a single column (lvarchar), holding the text that would otherwise appear in the “schematext” parameter. It is sometimes necessary to use “schemasource” instead of “schematext”, since VTI limits the total length of the parameter string to 255 characters.
- 5) Any column names used in the virtual table definition must exist either as base table columns (in this case A, B, or C) or DBXten DSChip internal columns (in this case, X,Y,Z,T, or V), but not both.
- 6) Other parameters governing how the VTI table will be queried can be specified. See the various examples in [Using the DBXten Virtual Table Interface to Extract Data](#) (starting on page 84).

### **Load the VTI Virtual Table**

Finally, load the VTI virtual table as you would any other conventional Informix table, e.g., with SQL, ODBC, JDBC, or the High Performance Loader.

### **Performance Penalty in Using VTI for Loads**

Note that there is a performance penalty in using VTI to load a DBXten table. Figure 14 below shows timings (h:mm:ss.frac) for loads into indexed tables of various sizes. For each *Number of Rows* value N, four loads were performed:

- 1) into an N row conventional table, using the dbload utility,
- 2) into an N row conventional table, using the High Performance Loader,
- 3) into an N tuple, N/1000 row DBXten table, using the CSV loader utility described in [Appendix E](#), and
- 4) into an N tuple, N/1000 row DBXten table using the High Performance Loader loading into a VTI table.

Note that the significance of the penalty decreases with number of rows. For large numbers of rows, loading into an indexed table using VTI is still much faster than loading into an equivalent conventional table.

# of rows	Timings for Table Type and Load Method			
	Conventional Table		DBXten Table	
	DBLOAD	HPL	nonVTI	VTI/HPL
2 million	0:04:12.00	0:01:46.00	0:01:44.00	0:02:11.00
3 million	0:06:09.00	0:02:54.00	0:02:32.00	0:03:15.00
4 million	0:08:12.00	0:03:31.00	0:03:31.00	0:04:31.00
5 million	0:10:38.00	0:07:29.00	0:04:21.00	0:05:34.00
6 million	0:13:09.00	0:09:25.00	0:05:21.00	0:06:47.00
10 million	0:26:12.00	0:28:10.00	0:08:56.00	0:11:01.00
25 million	1:20:48.00	1:07:49.00	0:22:47.00	0:28:10.00
50 million	3:08:18.00	2:09:10.00	0:45:44.00	0:55:02.00

Figure 14: VTI Table Sample Timings

### **VTI Limitations**

See [Using VTI - Current Limitations](#) (page 89).

## 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 48).

### 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 ESQ-C DSChip Loading Program

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

```
make loadTable
```

in that directory.

Before running this program, run the following SQL:

```
> create table BCS_DBXten_chips(chip DSChip);
```

and set the environment variable `$DBXTEN_DEMO_DATABASE` to the name of the database that you wish to use.

The first lines in the ESQ C file include the definitions for DBXten and ESQ statements for pulling in the appropriate Informix files.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <dschip_exports.h>
#include <dschipInformixClient.h>
#include <dschip_exports.h>
```



```
EXEC SQL include sqltypes;  
EXEC SQL include exp_chk.ec;
```

The section below defines the metadata for the DSChip's that we will be loading. Each DSChip will have 4 columns, and there will be a maximum of 1000 tuples per DSChip. Two functions are defined for generating the sample data to be loaded into the DSChip's.

```
/* actual data */  
#define MAX_ROWS_PER_CHIP (1000)  
#define NUM_CHIP_COLUMNS (4)  
  
static double nextDouble(int i) {  
    double t = fabs( sin(i*3));  
    return t - floor(t);  
}  
  
static int nextInt(int i) {  
    double t = nextDouble(i);  
    return (int)floor(t*37);  
}
```

The next section builds a DSChip. The basic process for doing this is to first initialize the DSChip, then define each column, and finally add data for each column and tuple.

```
DSChip * BuildChip()  
{  
    DSChip *chip;  
  
    DSChipVar *columns[NUM_CHIP_COLUMNS];  
    int i;  
    char textBuffer[80];  
  
    chip = DSChipCreate(MAX_ROWS_PER_CHIP);  
    DSChipAddVar(chip, "intVal", DSVarTypeINT, 0);  
    DSChipAddVar(chip, "doubleVal", DSVarTypeDOUBLE, 0.1);  
    DSChipAddVar(chip, "dateVal", DSVarTypeDATE, 0.1);  
    DSChipAddVar(chip, "stringVal", DSVarTypeSTRING, 0);  
  
    for( i = 0; i < NUM_CHIP_COLUMNS; i++ ) {  
        columns[i] = DSChipGetVarByPos(chip, i);  
    }  
  
    for( i = 0; i < MAX_ROWS_PER_CHIP; i++ ) {  
        DSChipVarSetInt(columns[0], i, i&7);  
        DSChipVarSetDouble(columns[1], i, nextDouble(i));  
        DSChipVarSetDouble(columns[2], i,  
            DSTimeStringToDouble("2008-04-1 12:43:49"));  
        sprintf(textBuffer, "hello%d", nextInt(i)%5);  
        DSChipVarSetString(columns[3], i, textBuffer);  
    }  
    return chip;  
}
```

}

The StoreChipInTable procedure simply takes as input a DSChip binary object and inserts it into a DSChip column of a new row in the BCS\_DBXten\_chips table.

```
void StoreChipInTable( DSChip *chip)
{
    EXEC SQL BEGIN DECLARE SECTION;
        var binary "DSChip" obj=NULL;
        char *database;
    EXEC SQL END DECLARE SECTION;

    database = getenv("DBXTEN_DEMO_DATABASE");

    if (! database) {
        DSCodeError("DBXTEN_DEMO_DATABASE environment variable is
not set");
    }

    obj = DSChipPack(chip);
    EXEC SQL whenever sqlerror CALL processSQLException;
    EXEC SQL connect to :database;
    EXEC SQL insert into BCS_DBXten_chips(chip) values( :obj) ;
    if( ifx_var_dealloc(&obj)) {
        DSCodeError("failed to free obj");
    }
    EXEC SQL disconnect current;
}
```

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

- 1) constructs a DSChip,
- 2) opens a database connection, inserts the DSChip into a table, and closes the database connection.

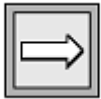
```
int main(int argc, char *argv) {
    DSChip *chip;

    chip = BuildChip();
    StoreChipInTable(chip);
    return 0;
}
```

## 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 consists of two classes – BuildChip and Insert – that can be found in the `examples/java/` directory in the DBXten distribution. The program can be run by issuing the command “`make runLoad`” in that directory.

### A Sample Java DSChip Loading Program



Before running this program, set the environment variable `$DBXTEN_DEMO_DATABASE` to the name of the database that you wish to use. In addition, you must replace the “`you_must_set_this`” strings in the first few lines of the Makefile

```
#  
# Password so that jdbc can form a connection.  
#  
PASSWORD = you_must_set_this  
IDS_SERVER_NAME = you_must_set_this  
IDS_SERVER_PORT = you_must_set_this
```

so that JDBC can connect to the database.

The first lines in the Java classes define the namespaces needed for using DBXten and running these examples:

Insert.java:

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

BuildChip.java:

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

#### BuildChip

The BuildChip class has two private methods for generating integer and double precision data to be inserted into the sample DSChip's. The rest of the class is comprised of a single public method – `build()` – which builds and returns a DSChip binary object. The object is built by first initializing it (by specifying the maximum number of tuples it can hold), defining its columns, and then loading values into each column and each tuple.

```
public class BuildChip {
    private static double nextDouble(int i) {
        double t = Math.abs(Math.sin(i*3));
        return t - Math.floor(t);
    }

    private static int nextInt(int i) {
        double t= nextDouble(i);
        return (int)Math.floor(t*37);
    }

    public static DSChip build() {
        final int numRows = 1000;
        DSChip chip = new DSChip(numRows);
        int intCol = chip.addIntColumn("anIntColumn");
        int doubleCol = chip.addDoubleColumn(
            "aDoubleColumn", 0.1);
        int dateCol = chip.addDateColumn("aDateColumn", 0.1);
        int stringCol = chip.addStringColumn("aStringColumn");

        for(int i = 0; i < numRows; i++ ) {
            chip.setInt(i, intCol, i%7);
            chip.setDouble(i, doubleCol, nextDouble(i));
            chip.setDate(i, dateCol,
                Timestamp.valueOf("2008-04-01 12:42:49"));
            chip.setString(i, stringCol, "hello" + nextInt(i)%5);
        }
        return chip;
    }
}
```

## **Insert**

The Insert class simply creates a binary DSChip “BuildChip” instance, then opens a connection to the database, inserts a row into the table, and closes the connection to the database.

```
public class Insert {

    public static void main(String args[]) {
        DSChip chip = BuildChip.build();

        try {
            Connection conn = Connect.getConnection(args);
            PreparedStatement ps = conn.prepareStatement(
                "insert into BCS_DBXten_chips" +
                " values(DSChipRecv(?))");
            ps.setBytes(1, chip.toBytes());
            chip.clearRows();
            ps.execute();
            conn.close();
        } catch( java.sql.SQLException e2 ) {
            throw new RuntimeException(e2.toString());
        }
    }
}
```





## Chapter 5: Retrieving Data from DSChip's

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

- 1) by using the DBXten Native SQL API,
- 2) by using SQL and the DBXten Virtual Table Interface,
- 3) by writing and running a C program written using the DBXten C API,  
or
- 4) 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 the DBXten Native SQL, C, or Java API's. The picture for the DBXten Virtual Table Interface is much simpler – see [Using the DBXten Virtual Table Interface to Extract Data](#).

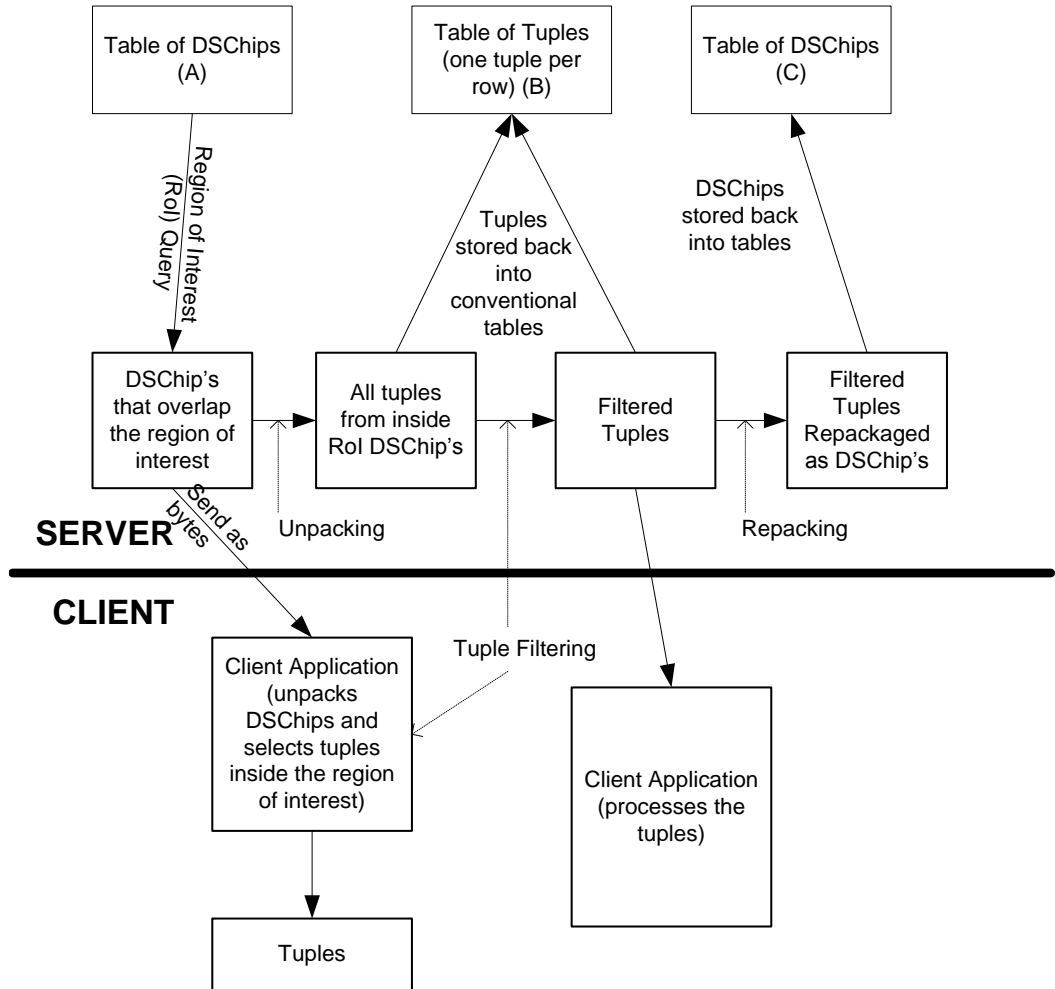


Figure 15: DSChip Extraction Processing paths.

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  $x = x_{val}$ , y is between  $y_{min}$  and  $y_{max}$ , 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 -4 and +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.

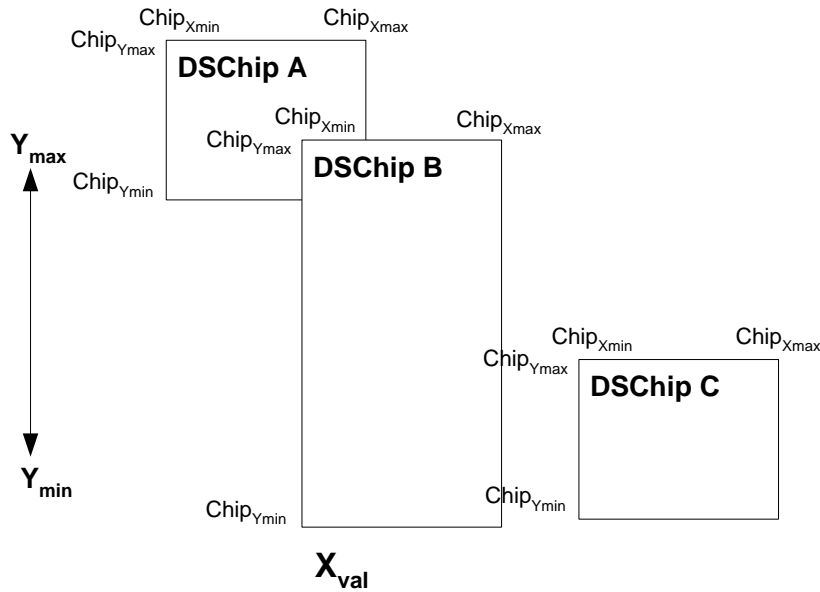


Figure 16: DSChip's A and B overlap the specified X and Y ranges.

**“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 15 (page 52), 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 15) 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 15 (page 52) 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 Native 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:

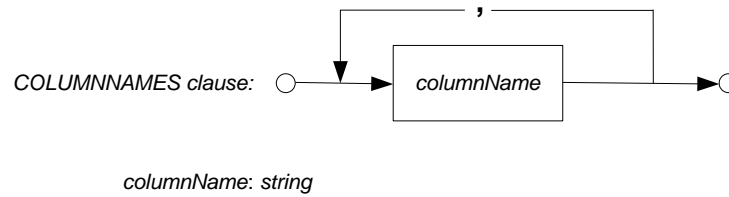


Figure 17: Syntax of a `columnNames` parameter.

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

The `rangeSpec` parameter has this syntax:

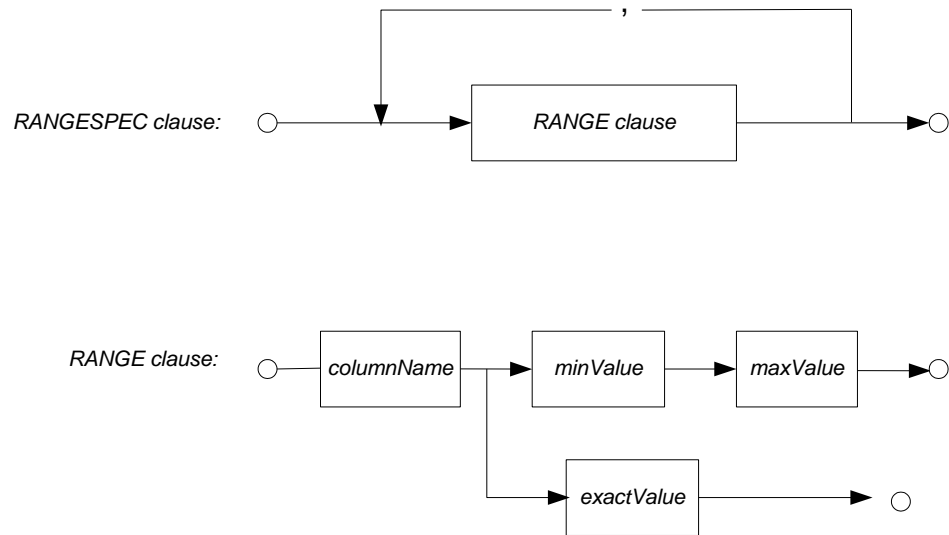


Figure 18: Syntax of a `rangeSpec` parameter.

## 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<sup>15</sup>), so we use a “LIMIT 1” clause to limit the number of rows returned to one.

### Example

```
SELECT FIRST 1 DSChipSchema(chip) FROM xyvals;
```

```
(expression)  maxtuples 100, x integer, y integer, z float 0.1,  
              val float 0.01
```

1 row(s) retrieved.

### Example

```
SELECT FIRST 1 DSChipSchema(measurement_block) FROM  
              measurementBlocks;
```

```
(expression)  maxtuples 2, datetime datetime 10, latitude float  
              0.001, longitude float 0.001, intColumn integer
```

1 row(s) retrieved.

## “Unwrapping” DSChip’s

The following function can be used to turn DSChip’s into tuples.

```
FUNCTION DSQueryToStrings(sqlSelectStatement lvvarchar) RETURNS  
setof(lvvarchar)
```

What is returned by this function is the same as what would be returned by the embedded *sqlSelectStatement* alone, except:

- 1) instead of individual column values being returned, a “*pseudo-row-type*” string which is cast-able to a row type is returned. This string looks like a row type, but in actuality it is simply the literal “ROW (”, followed by a comma-separated list of values, followed by a final “)”,
- 2) each tuple of any DSChip’s returned is reflected in a row of the output,

---

<sup>15</sup> 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?”

- 3) each column of any DSChip's returned by the select statement is reflected by a column in the pseudo-row-type.

### Example

These examples use the “measurements” table, the contents of which were shown [earlier](#) (page 26). That table consists of three rows, the first two referring to instrument\_id = 1 and the third referring to instrument\_id = 2. Each of the rows contains a DSChip with 7 tuples.

```
> execute function DSQueryToStrings('select
    measurement_block_id::integer,
    measurement_block
    from measurements
    where instrument_id = 1;');

(expression) ROW(1,2008-02-01 00:12:20,46.343,-127.386,14.34,10.2)
(expression) ROW(1,2008-02-01 00:12:30,46.344,-127.385,16.82,11.9)
(expression) ROW(1,2008-02-01 00:12:30,46.345,-127.383,18.85,10.7)
(expression) ROW(1,2008-02-01 00:12:30,46.346,-127.382,21.22,11.7)
(expression) ROW(1,2008-02-01 00:12:30,46.347,-127.381,23.66,10.9)
(expression) ROW(1,2008-02-01 00:12:30,46.349,-127.379,26.05,11.4)
(expression) ROW(1,2008-02-01 00:13:40,46.392,-127.335,104.82,10.7)
(expression) ROW(2,2008-02-01 00:13:50,46.394,-127.334,106.83,10.7)
(expression) ROW(2,2008-02-01 00:13:50,46.395,-127.332,108.90,13.1)
(expression) ROW(2,2008-02-01 00:13:50,46.396,-127.331,110.99,10.7)
(expression) ROW(2,2008-02-01 00:13:50,46.398,-127.330,113.32,11.4)
(expression) ROW(2,2008-02-01 00:13:50,46.399,-127.328,115.38,10.2)
(expression) ROW(2,2008-02-01 00:14:00,46.400,-127.327,117.65,10.9)
(expression) ROW(2,2008-02-01 00:15:00,46.439,-127.289,188.40,10.3)

14 row(s) retrieved.
```

In the SQL statement above we've appended the `measurement_block_id` to each tuple contained in `measurement_block`, and we've restricted our output to just those DSChip-contained tuples from the two rows that have an `instrument_id` value of 1.

Note that each row of the output consists of a single *character string* column, even though it looks like a composite data (row) type column. The following strategy can be used to cast the output into a group of distinct columns.

First, create a row type that has elements corresponding to the elements that would be returned by the `SELECT` statement that is passed to `DSQueryToStrings`, treating each tuple column as a separate element:

```
> create row type measurements_t(blockid integer,  
                                dt datetime year to second,  
                                latitude float,  
                                longitude float,  
                                depth float,  
                                measurement float);  
  
Row type created.
```

Then the following `SELECT` statement will return the elements of the select statement as separate columns. Note that some of the columns can be left out:

```
> select (a::measurements_t).blockid,  
        (a::measurements_t).dt,  
        (a::measurements_t).depth,  
        (a::measurements_t).measurement  
from  
TABLE(function DSQueryToStrings('select  
    measurement_block_id::integer,  
    measurement_block  
from measurements  
where instrument_id = 1')) as vtab(a);
```

blockid	dt		depth	measurement
1	2008-02-01 00:12:20	14.340000000000	10.200000000000	
1	2008-02-01 00:12:30	16.820000000000	11.900000000000	
1	2008-02-01 00:12:30	18.850000000000	10.700000000000	
1	2008-02-01 00:12:30	21.220000000000	11.700000000000	
1	2008-02-01 00:12:30	23.660000000000	10.900000000000	
1	2008-02-01 00:12:30	26.050000000000	11.400000000000	
1	2008-02-01 00:13:40	104.820000000000	10.700000000000	
2	2008-02-01 00:13:50	106.830000000000	10.700000000000	
2	2008-02-01 00:13:50	108.900000000000	13.100000000000	
2	2008-02-01 00:13:50	110.990000000000	10.700000000000	
2	2008-02-01 00:13:50	113.320000000000	11.400000000000	
2	2008-02-01 00:13:50	115.380000000000	10.200000000000	
2	2008-02-01 00:14:00	117.650000000000	10.900000000000	
2	2008-02-01 00:15:00	188.400000000000	10.300000000000	

14 row(s) retrieved.

As mentioned above this query restricted the output to just some of the columns in the `DSChip` tuple (leaving out latitude and longitude). In [a later section](#) another method for doing this will be described.

### **Listing Distinct Values for DSChip Column(s)**

The following function eliminates all but a selected set of columns from a `DSChip` and then removes duplicate tuples from the result.

```
FUNCTION DSDistinct(existingChip DSChip, columnNames char)  
RETURNS DSChip
```

**Example**

Consider the block of tuples described [earlier](#).

	<b>Datetime</b>	<b>Latitude</b>	<b>Longitude</b>	<b>Depth</b>	<b>Measurement</b>
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

Suppose that these tuples, minus the ... tuples, were stored in a DSChip and that we applied DSDistinct to this DSChip, specifying a columnNames parameter value that consisted of just the Measurement column. Then DSDistinct would first eliminate the first four columns:

	<b>Measurement</b>
Block 1:	10.2
	11.9
	10.7
	11.7
	10.9
	11.4
	10.7

and remove the duplicate 10.7 tuple, leaving:

	<b>Measurement</b>
Block 1:	10.2
	11.9
	10.7
	11.7
	10.9
	11.4

The following example provides the SQL for doing this.

**Example**

```
> execute function DSQueryToStrings("select  
    DSDistinct(measurement_block,'measurement') from  
    measurements where measurement_block_id = 1;");
```

```
(expression)    ROW(10.2)  
(expression)    ROW(10.7)  
(expression)    ROW(10.9)  
(expression)    ROW(11.4)  
(expression)    ROW(11.7)  
(expression)    ROW(11.9)
```

6 row(s) retrieved.

Of course the same strategy can be applied to multiple DSChip's simply by removing or changing the WHERE clause:

```
> EXECUTE FUNCTION DSQueryToStrings("SELECT  
    DSDistinct(measurement_block,'measurement') FROM  
    measurements WHERE instrument_id = 1;");
```

```
(expression)    ROW(10.2)  
(expression)    ROW(10.7)  
(expression)    ROW(10.9)  
(expression)    ROW(11.4)  
(expression)    ROW(11.7)  
(expression)    ROW(11.9)  
(expression)    ROW(10.2)  
(expression)    ROW(10.3)  
(expression)    ROW(10.7)  
(expression)    ROW(10.9)  
(expression)    ROW(11.4)  
(expression)    ROW(13.1)
```

12 row(s) retrieved.

Note, however, that DSDistinct only removes the duplicates within DSChip's. In the example above, the values 10.2, 10.7, 10.9, and 11.4 all appear twice since they occur in both DSChips. In order to remove duplicates appearing in different DSChip's you can do the following:

```
> SELECT DISTINCT * from TABLE(DSQueryToStrings("SELECT  
DSDistinct(measurement_block,'measurement') FROM measurements  
WHERE instrument_id = 1;"));
```

```
unnamed_col_1 ROW(10.2)  
unnamed_col_1 ROW(10.3)  
unnamed_col_1 ROW(10.7)  
unnamed_col_1 ROW(10.9)  
unnamed_col_1 ROW(11.4)  
unnamed_col_1 ROW(11.7)  
unnamed_col_1 ROW(11.9)  
unnamed_col_1 ROW(13.1)
```

8 row(s) retrieved.

### **Determining Whether a DSChip has a Particular Column(s)**

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

This function returns a boolean value indicating whether selected DSChip's contain *all* of the column names specified.

#### **Example**

```
> SELECT measurement_block_id,  
       DSHasVariables(measurement_block,'latitude,longitude')  
FROM measurements;
```

```
measurement_block+ (expression)
```

1	t
2	t
3	t

3 row(s) retrieved.

```
> SELECT measurement_block_id,  
       DSHasVariables(measurement_block,  
                       'latitude,longitude,notAColumn')  
FROM measurements;
```

```
measurement_block+ (expression)
```

1	f
2	f
3	f

3 row(s) retrieved.

### **Determining How Many Columns a DSChip has**

```
FUNCTION DSNumVars(existingChip DSChip) RETURNS integer
```

**Example**

```
> SELECT measurement_block_id,  
       DSNumVars(measurement_block)  
FROM measurements;
```

```
measurement_block+ (expression)
```

1	5
2	5
3	5

3 row(s) retrieved.

**Determining the Number of Tuples in a DSChip**

FUNCTION DSNumFilledRows(*existingChip* DSChip) RETURNS integer

**Example**

```
> SELECT measurement_block_id,  
       DSNumFilledRows(measurement_block)  
FROM measurements;
```

```
measurement_block+ (expression)
```

1	7
2	7
3	7

3 row(s) retrieved.

**Determining the Maximum Value for Columns in a DSChip**

FUNCTION DSMaxAsString(*existingChip* DSChip, *columnNames* char)  
RETURNS lvarchar

**Example**

```
> SELECT measurement_block_id,  
       DSMaxAsString(measurement_block, 'latitude longitude')  
FROM measurements;
```

```
measurement_block+ 1  
(expression)      46.392,-127.335  
  
measurement_block+ 2  
(expression)      46.439,-127.289  
  
measurement_block+ 3  
(expression)      46.486,-127.241
```

3 row(s) retrieved.

Note that this function returns subsets of actual tuples, finding the tuple(s) with the largest value for the first column, then of those tuples finding the tuple(s) with the largest value for the second column, and so on. So the values shown in the second, third, fourth, etc. columns might not be the actual maximums for those columns (although in this case it is).

**Determining the Minimum Value for Columns in a DSChip**

FUNCTION DSMinAsString(*existingChip* DSChip, *columnNames* char)  
RETURNS lvarchar

**Example**

```
> SELECT measurement_block_id,  
       DSMinAsString(measurement_block, 'latitude')  
FROM measurements;
```

```
measurement_block+ 1  
(expression)      46.343  
  
measurement_block+ 2  
(expression)      46.394  
  
measurement_block+ 3  
(expression)      46.441
```

3 row(s) retrieved.

**Listing Compression Information for a DSChip**

FUNCTION DSCompressInfo(*existingChip* DSChip) RETURNS lvarchar

**Example**

```
> SELECT measurement_block_id,  
       DSCompressInfo(measurement_block)  
FROM measurements;
```

```
measurement_block+ 1  
(expression)      Total 280 | datetime delta-fixed 29 |  
                  latitude delta-fixed 32 | longitude delta-  
                  fixed 32 | depth signed-delta-variable 35 |  
                  measurement fixed 25
```

```
measurement_block+ 2  
(expression)      Total 287 | datetime delta-fixed 29 |  
                  latitude delta-fixed 32 | longitude delta-  
                  fixed 32 | depth signed-delta-variable 35 |  
                  measurement delta-fixed 32
```

```
measurement_block+ 3  
(expression)      Total 280 | datetime delta-fixed 29 |  
                  latitude delta-fixed 32 | longitude delta-  
                  fixed 32 | depth signed-delta-variable 35 |  
                  measurement fixed 25
```

3 row(s) retrieved.

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* does not include meta-data such as precisions and the column name.

**Converting a DSChip Range to a DSBox**

FUNCTION DSRangeToBox(*rangeSpec* char) RETURNS lvarchar

**Example**

```
SELECT DSRangeToBox('datetime 1970-01-01 00:00:00 1970-01-01  
00:00:01, latitude 49.0 49.01')  
FROM systables WHERE tabid = 1;
```

```
(constant) (D          28800.000000,49.000000), (D  
           28801.000000,49.010000)
```

1 row(s) retrieved.

**Generating a Bounding DSBox from a DSChip**

FUNCTION DSAsBoxString(*existingChip* DSChip, *columnNames* char)  
RETURNS lvarchar

**Example**

```
> SELECT measurement_block_id,  
           DSAsBoxString(measurement_block,  
                         'latitude,longitude')  
FROM measurements;  
  
measurement_block+ 1  
(expression)      (46.342500,-127.386500), (46.392500,-127.334500)  
  
measurement_block+ 2  
(expression)      (46.393500,-127.334500), (46.439500,-127.288500)  
  
measurement_block+ 3  
(expression)      (46.440500,-127.287500), (46.486500,-127.240500)  
  
3 row(s) retrieved.
```

**Doing a Region of Interest Query to Select DSChip's**

As discussed [earlier](#) (page 53), 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 DSBox-valued functional index built on the DSChip columns of interest (possibly in addition to other columns that are not of interest), and
- 2) some DSBox, or set of DSBox's, to be used in comparison, and
- 3) the Informix "overlap" 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 the following SQL:

```
> CREATE FUNCTION chip_cube(chipIn DSChip)  
   RETURNING DSBox WITH (NOT VARIANT)  
>   RETURN DSAsBoxString(chipIn, 'x,y')::DSBox;  
> END FUNCTION;
```

Routine created.

```
> CREATE INDEX xyvalsIndex ON  
   xyvals(chip_cube(chip) dsbox_ops)  
   USING rtree;
```

Index created.

**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:

```
> SELECT chip FROM xyvals
   WHERE overlap(chip_cube(chip),
                 DSRangeToBox('x 26 39,y 31 53')::DSBox);
```

**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 overlap(chip_cube(chip),
                 DSRangeToBox('x * *,y 31 53')::DSBox);
```

**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 overlap(DSAsBoxString(chip,'y'),
                 DSRangeToBox('y 31 53')::DSBox);
```

**Example**

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

```
> SELECT chip FROM xyvals, references
   WHERE overlap(DSAsBoxString(chip,'y'),
                 referenceBoxes.cube) and
                 referenceBoxes.otherAttribute = 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:

```
> CREATE FUNCTION measurement_block_cube(chipIn DSChip)
    RETURNING DSBox WITH (not variant)
> RETURN DSAsBoxString(chipIn,
    'datetime,latitude,longitude')::DSBox;
> END FUNCTION;
```

Routine created.

```
> CREATE INDEX measurementblock_idx on measurementblocks
    (measurement_block_cube(measurement_block) dsbox_ops)
    USING rtree;
```

Index created.

```
> SELECT measurement_block_id, measurement_block
    FROM measurementBlocks
    WHERE overlap(measurement_block_cube(measurement_block),
        DSRangeToBox('datetime 2008-01-01 12:00:00
2008-01-01 14:00:00,latitude * *,longitude * *')::DSBox);
```

```
measurement_block+ 2
measurement_block  maxtuples=2, filledtuples=2, numcolumns=4; d
                    atetime,datetime,10;latitude,float,0.001;
                    longitude,float,0.001;intColumn,integer,0
                    ;2008-01-01 12:30:30,49.700,-127.500,45;2
                    008-01-01 12:30:40,49.512,-127.724,24
```

1 row(s) retrieved.

## **Extracting Tuples from a DSChip (no tuple filtering)**

In an [earlier section](#) (page 56) we described the function:

```
FUNCTION DSQueryToStrings(sqlSelectStatement lvarchar) RETURNS
setof(lvarchar)
```

which can be used to “unwrap” DSChip’s.

All of the DSChip-returning examples described in the previous sections can all be modified to return tuples instead of DSChip’s simply by wrapping the SELECT statements inside EXECUTE FUNCTION DSQueryToStrings(...).

**Example**

(In the following example, line feeds have been added for readability).

```
> EXECUTE FUNCTION DSQueryToStrings("SELECT chip FROM xyvals  
WHERE overlap(chip_cube(chip),  
DSRangeToBox('x 25,y 50 51')::DSBox)");
```

```
(expression) ROW(20,50,0.0,0.89)  
(expression) ROW(20,51,0.0,0.18)  
(expression) ROW(20,52,0.0,0.79)  
(expression) ROW(20,53,0.0,0.20)  
.  
.  
.  
(expression) ROW(29,57,12.0,0.38)  
(expression) ROW(29,58,12.0,0.55)  
(expression) ROW(29,59,12.0,0.45)
```

500 row(s) retrieved.

The query above does return what *appears* to be a set of tuples, but they are really just character strings. However, as described [earlier](#) (page 57), the SQL can be rewritten to return individual columns:

**Example**

**BCS DBXTEN DATABLADE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

```
> CREATE ROW TYPE xyz_t(x integer,  
                        y integer,  
                        z double precision,  
                        val double precision);
```

Row type created.

```
> SELECT FIRST 10 (a::xyz_t).x,  
                (a::xyz_t).y,  
                (a::xyz_t).z,  
                (a::xyz_t).val  
FROM TABLE(FUNCTION DSQueryToStrings("SELECT chip FROM  
xyvals WHERE overlap(chip_cube(chip),  
DSRangeToBox('x 25,y 50 51')::DSBox)")) as vtab(a);
```

x	y	z	val
20	50	0.00	0.89
20	51	0.00	0.18
20	52	0.00	0.79
20	53	0.00	0.2
20	54	0.00	0.05
20	55	0.00	0.31
20	56	0.00	0.17
20	57	0.00	0.56
20	58	0.00	0.95
20	59	0.00	1.000000000000

10 row(s) retrieved.

It is also possible to do the following:

**Example**

```
> CREATE TABLE xyzvals (tuple xyz_t);  
Table created.
```

```
> INSERT INTO xyzvals  
    EXECUTE FUNCTION DSQueryToStrings("SELECT chip FROM xyzvals  
    WHERE overlap(chip_cube(chip), DSRangeToBox('x 25,y  
    50 51')::DSBox)");
```

500 row(s) inserted.

```
> SELECT FIRST 10 tuple.x, tuple.y, tuple.z, tuple.val  
    FROM xyzvals;
```

x	y	z	val
20	50	0.00	0.89
20	51	0.00	0.18
20	52	0.00	0.79
20	53	0.00	0.2
20	54	0.00	0.05
20	55	0.00	0.31
20	56	0.00	0.17
20	57	0.00	0.56
20	58	0.00	0.95
20	59	0.00	1.000000000000

10 row(s) retrieved.

**Counting Tuple Matches without Extracting**

The following function can be used to count the number of tuples from selected DSChip's that satisfy a specified rangeSpec:

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

### Example

```
> SELECT measurement_block_id,  
       DSChipNumMatches(measurement_block,  
                        'datetime 2008-02-01 00:13:00 2008-02-01  
                        00:14:00,latitude * *,longitude * *')  
FROM measurements  
WHERE overlap(measurement_block_cube(measurement_block),  
             DSRangeToBox('datetime 2008-02-01 00:13:00  
                          2008-02-01 00:14:00,  
                          latitude * *,  
                          longitude * *')::DSBox);  
  
measurement_block+ (expression)  
  
           1           1  
           2           6  
           3           0
```

3 row(s) retrieved.

By looking at the tuple values shown [earlier](#) (page 8), we can see that there is indeed one tuple in the first block that satisfies the datetime range specification, six in the second block, and none in the third block.

### 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 DSChipExtract(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 two DSChip's that satisfy the WHERE clause, so the query produces two new DSChip's. The *columnNames* parameter is used to specify that the new DSChip's will have just two columns (datetime and measurement) instead of five. (Linefeeds have been added to improve readability).

**BCS DBXTEN DATABLEDE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

```
> SELECT DSChipExtract(measurement_block,  
                        'datetime 2008-02-01 00:13:00  
                          2008-02-01 00:14:00,  
                          latitude * *,  
                          longitude * *',  
                        'datetime,measurement')  
      FROM measurements  
WHERE overlap(measurement_block_cube(measurement_block),  
             DSRangeToBox('datetime 2008-02-01 00:13:00  
                           2008-02-01 00:14:00,  
                           latitude * *,  
                           longitude * *')::DSBox);
```

```
measurement_block+ 1  
(expression)       maxtuples=100, filledtuples=1, numcolumns=2; dat  
etime, datetime, 10; measurement, float, 0.1; 2008-  
02-01 00:13: 40, 10.7
```

```
measurement_block+ 2  
(expression)       maxtuples=100, filledtuples=6, numcolumns=2; dat  
etime, datetime, 10; measurement, float, 0.1; 2008-  
02-01 00:13: 50, 10.7; 2008-02-01  
00:13:50, 13.1; 2008-02-01 00:13:5 0, 10.7; 2008-  
02-01 00:13:50, 11.4; 2008-02-01 00:13:50  
, 10.2; 2008-02-01 00:14:00, 10.9
```

2 row(s) retrieved.

**Example**

In the second example there are again two DSChip's that satisfy the `WHERE` clause, so the query again produces two new DSChip's. The `columnNames` parameter requests that the new DSChip's have the same five columns as the input, but the `rangeSpec` parameter specifies a tighter range than the `WHERE` clause and hence results in no tuples being put into the new DSChip's.

```
> SELECT DSChipExtract(measurement_block,  
                        'datetime 2008-02-01 00:13:00  
                          2008-02-01 00:13:05,  
                          latitude * *,  
                          longitude * *',  
                        'datetime,latitude,longitude,  
                          depth,measurement')  
      FROM measurements  
      WHERE overlap(measurement_block_cube(measurement_block),  
                   DSRangeToBox('datetime 2008-02-01 00:13:00  
                                2008-02-01 00:14:00,  
                                latitude * *,  
                                longitude * *')::DSBox);
```

```
measurement_block+ 1  
(expression)       maxtuples=100, filledtuples=0, numcolumns=5; dat  
etime,datetime,10;latitude,float,0.001;longit  
ude,float,0.001;depth,float,0.01;measurement,  
float,0.1  
  
measurement_block+ 2  
(expression)       maxtuples=100, filledtuples=0, numcolumns=5; dat  
etime,datetime,10;latitude,float,0.001;longit  
ude,float,0.001;depth,float,0.01;measurement,  
float,0.1
```

2 row(s) retrieved.

## **Tuple Filtering DSChip's into a Set of Tuples**

The DSChipExtract function can be used in conjunction with function [DSQueryToStrings](#) to return tuples instead of 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 DSChipExtract(existingChip DSChip, rangeSpec char,  
                      columnNames char) RETURNS DSChip;
```

### **Example**

```
> EXECUTE FUNCTION DSQueryToStrings(  
    "SELECT measurement_block_id::integer,  
        DSChipExtract(measurement_block,  
            'datetime 2008-02-01 00:13:00 2008-02-01 00:14:00,  
            latitude * *',  
            longitude * *',  
            'datetime,measurement')  
    FROM measurements  
    WHERE  
        overlap(  
            measurement_block_cube(measurement_block),  
            DSRangeToBox('datetime 2008-02-01 00:13:00  
                2008-02-01 00:14:00,  
                latitude * *,longitude * *')::DSBox)"  
);  
  
(expression) ROW(1,2008-02-01 00:13:40,10.7)  
(expression) ROW(2,2008-02-01 00:13:50,10.7)  
(expression) ROW(2,2008-02-01 00:13:50,13.1)  
(expression) ROW(2,2008-02-01 00:13:50,10.7)  
(expression) ROW(2,2008-02-01 00:13:50,11.4)  
(expression) ROW(2,2008-02-01 00:13:50,10.2)  
(expression) ROW(2,2008-02-01 00:14:00,10.9)
```

7 row(s) retrieved.

## **Using the DBXten C API to Extract Data**

This section will guide you through two sample ESQL - 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.ec`) 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 Informix 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)

The first lines in the ESQL C file include the definitions for DBXten and ESQL statements for pulling in the appropriate Informix files.

```
#include <stdio.h>
#include <stdlib.h>
#include <dschip_exports.h>
#include <dschipInformixClient.h>

EXEC SQL include sqltypes;
EXEC SQL include exp_chk.ec;
```

The next section establishes a connection to the Informix server in function `CreateConnection`, and closes it in function `CloseConnection`.

```
void CreateConnection()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char *database;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL whenever sqlerror CALL processESQLerror;

    database = getenv("DBXTEN_DEMO_DATABASE");

    if (! database) {
        DSCodeError("DBXTEN_DEMO_DATABASE environment variable is
not set");
    }
    EXEC SQL connect to :database;
}

void CloseConnection()
{
    EXEC SQL disconnect current;
}
```

The following function generates the textual representation of the DSBox 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. It opens a cursor to process all the DSChip's that satisfy a [Region of Interest](#) condition. For each DSChip returned, the DSChip is unpacked and sent to the ProcessASingleChip function where [tuple filtering](#) and printing is performed.

```

void FetchChips( char *cubeExpression, double xRange[],
    double yRange[])
{
    EXEC SQL BEGIN DECLARE SECTION;
        var binary "dschip" chipVar=NULL;
        char *database;
        char *cubeExpressionP;
    EXEC SQL END DECLARE SECTION;
    DSChip *chip;

    EXEC SQL whenever sqlerror CALL processSQLException;

    cubeExpressionP = cubeExpression;
    database = getenv("DBXTEN_DEMO_DATABASE");

    if (! database) {
        DSCodeError("DBXTEN_DEMO_DATABASE environment variable is
not set");
    }

    EXEC SQL DECLARE chipCursor cursor for
        SELECT chip into :chipVar FROM xyvals WHERE
            overlap(DSAsBoxString(chip, 'x,y')::DSBox,
                DSRangeToBox(:cubeExpressionP)::DSBox);
    EXEC SQL open chipCursor;

    for (;;)
    {

```

```
EXEC SQL fetch chipCursor;
if (strncmp(SQLSTATE, "00", 2) != 0)
    break;
chip = DSChipUnPack(chipVar);
ifx_var_dealloc(&chipVar);
ProcessASingleChip(chip, xRange, yRange);
}
if (strncmp(SQLSTATE, "02", 2) != 0)
    printf("SQLSTATE after fetch is %s\n", SQLSTATE);
EXEC SQL close chipCursor;
EXEC SQL free chipCursor;
}
```

Finally, here is the main program, which specifies the Region of Interest (X between 26 and 30; Y between 50 and 51).

```
static double xRange[] = { 26, 30};
static double yRange[] = { 50, 51 };

int main(int argc, char *argv) {
    char queryText[256];

    CreateConnection();
    GenerateKeyText(queryText, xRange, yRange);
    FetchChips(queryText, xRange, yRange);
    CloseConnection();
    return 0;
}
```

## **Performing Tuple Filtering on the Server**

The second program (fetch2.ec) 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 Informix database server.

There are only two difference between this program and the [fetch1.ec](#) program described in the previous section. In `FetchChips`, the SQL `SELECT` statement is changed from

```
SELECT chip into :chipVar FROM xyvals WHERE ...
```

to

```
SELECT DSChipExtract(chip, :cubeExpressionP, 'x,y,val')
    into :chipVar FROM xyvals WHERE ...
```

The change causes tuple filtering to be done on the server.

In `ProcessASingleChip`, the following lines (doing tuple filtering on the client) are removed.

```
/* 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));
    }
}
```

## Using the DBXten Java API to Extract Data

This class is the Java version of the [fetch1.c](#) program listed earlier.

```
/*
 * Fetch1.java
 */

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

public class Fetch1 {
```

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 `DSBox`-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 dschipSend(" +
            column_name + ") FROM " + table_name +
            " WHERE overlap(DSAsBoxString(chip,'x,y')::DSBox," +
            " DSRangeToBox(?)::DSBox)");
    }
```

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;
    }
    System.out.println("New DSChip");
    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 `DSRangeToBox`, which in turn generates the `DSBox` 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 Fetch1(Connection connection) {
    try {
        double [] xRange = { 26, 30 };
        double [] yRange = { 50, 51 };
        PreparedStatement ps =
            prepareToFetch(connection, "xyvals",
                            "chip");
        String key = generateKey(xRange, yRange);
        performQuery(ps, key, xRange, yRange);
        ps.close();
    } catch( java.sql.SQLException e1 ) {
        System.out.println(e1.toString());
    }
}

public static void main(String args[]) {
    Connection conn;
    try {
        conn = Connect.getConnection(args);
    } catch( java.sql.SQLException e2 ) {
        throw new RuntimeException(e2.toString());
    }
    new Fetch1(conn);
    try {
        conn.close();
    } catch( java.sql.SQLException e2 ) {
        throw new RuntimeException(e2.toString());
    }
}
}
```

## Using the DBXten Virtual Table Interface to Extract Data

The IBM Informix [Virtual Table Interface](#) (VTI) feature makes it possible for applications to see the columns inside a DSChip as if they were traditional columns in a traditional Informix table, i.e., a table with traditional columns instead of a DSChip column. Hence a previously-written application that accesses data from a non-DBXten database table can continue to be used, without modification or even recompilation, after the database table has been restructured using DBXten.

As an illustration of VTI, consider the following table called “mytable”.

mytable:

A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>
A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>
...			
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	D <sub>m</sub>

This table has three conventional (e.g., integer, float, varchar, etc.) columns – A, B, C – and one DSChip column D. Suppose that D has five internal columns – X, Y, Z, T, and V, where X, Y, Z, and T are space and time coordinates and V is the value of some property at the point in space and time (X,Y,Z,T).

mytable:

A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>
A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>
...			
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	D <sub>m</sub>

Inside of D<sub>1</sub>:

X <sub>11</sub>	Y <sub>11</sub>	Z <sub>11</sub>	T <sub>11</sub>	V <sub>11</sub>
X <sub>12</sub>	Y <sub>12</sub>	Z <sub>12</sub>	T <sub>12</sub>	V <sub>12</sub>
X <sub>13</sub>	Y <sub>13</sub>	Z <sub>13</sub>	T <sub>13</sub>	V <sub>13</sub>
...				
X <sub>1n</sub>	Y <sub>1n</sub>	Z <sub>1n</sub>	T <sub>1n</sub>	V <sub>1n</sub>

VTI provides a view<sup>16</sup> of this table, under a different name – we'll call it `mytable_vti`. This table appears to the user as a table of eight conventional columns – A, B, C, X, Y, Z, T, and V:

`mytable_vti:`

A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	X <sub>11</sub>	Y <sub>11</sub>	Z <sub>11</sub>	T <sub>11</sub>	V <sub>11</sub>
A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	X <sub>12</sub>	Y <sub>12</sub>	Z <sub>12</sub>	T <sub>12</sub>	V <sub>12</sub>
...							
A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	X <sub>1n</sub>	Y <sub>1n</sub>	Z <sub>1n</sub>	T <sub>1n</sub>	V <sub>1n</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	X <sub>21</sub>	Y <sub>21</sub>	Z <sub>21</sub>	T <sub>21</sub>	V <sub>21</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	X <sub>22</sub>	Y <sub>22</sub>	Z <sub>22</sub>	T <sub>22</sub>	V <sub>22</sub>
...							
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	X <sub>2n</sub>	Y <sub>2n</sub>	Z <sub>2n</sub>	T <sub>2n</sub>	V <sub>2n</sub>
...							
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	X <sub>m1</sub>	Y <sub>m1</sub>	Z <sub>m1</sub>	T <sub>m1</sub>	V <sub>m1</sub>
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	X <sub>m2</sub>	Y <sub>m2</sub>	Z <sub>m2</sub>	T <sub>m2</sub>	V <sub>m2</sub>
...							
A <sub>m</sub>	B <sub>m</sub>	C <sub>m</sub>	X <sub>mn</sub>	Y <sub>mn</sub>	Z <sub>mn</sub>	T <sub>mn</sub>	V <sub>mn</sub>

Queries issued against this new *virtual* table `mytable_vti` are automatically rewritten, under the covers of VTI, to access the data that is actually in the conventional and DSChip columns of the *base* table `mytable`.

In defining a VTI virtual table, you must tell VTI which indexed columns are to be used when answering queries against that virtual table. Hence, there is typically one VTI virtual table for each index that might get used.

---

<sup>16</sup> By “view” here we mean “view” in the general sense. This is not a database view; rather, it is a completely new (but virtual) table, having an access method that pulls data from `mytable`.

**Example**

If, for example, we have defined the following indexes on mytable

- 1) B-tree index on C (a conventional column)
- 2) R-tree index on the extents of X, Y, Z, and T in each DSChip

then we would define two VTI virtual tables, one for cases where we were querying based on C value and one for cases where we were querying based on X/Y/Z/T bounds:

```
SELECT * FROM mytable_vti_1 WHERE C = 'abc';
```

```
SELECT * FROM mytable_vti_2 WHERE T <= ... AND T >= ... AND X <= ...  
AND X > ... AND ...;
```

### **Steps in Defining a Virtual Table**

This section lists the steps involved in creating a VTI virtual table.

#### **Create and Load the Base Table**

These steps will likely already have been done, but they are included here for reference:

```
CREATE TABLE mytable (A integer, B float, C date, D DSChip);
```

...

< steps to load the table >

...

```
SELECT FIRST 1 DSChipSchema(D) FROM mytable;
```

```
(expression)  maxtuples 200, x float 0.01, y float 0.01, z  
              float 0.01, t datetime 1, v float 0.001
```

```
1 row(s) retrieved.
```

#### **Create Indexes on the Base Table**

In this example, the table has two indexes, a B-tree index on C and an R-tree index on the X, Y, Z, and T extents of each D DSChip.

```
CREATE INDEX mytable_c_idx ON mytable(c) USING btree;

CREATE FUNCTION pos_box(chipIn DSCHIP)
    RETURNING DSBox WITH (not variant)
RETURN DSAsBoxString(chipIn, 'x,y,z,t')::DSBox;
END FUNCTION;

CREATE INDEX mytable_pos_idx ON mytable(pos_box(D) DSBox_ops)
    USING rtree;
```

### **Defining a VTI Virtual Table for Queries on non-DSChip Columns (e.g., C)**

The SQL statement in the following example can be used to create a virtual table based on mytable; it specifies that C is an indexed column.

#### **Example**

```
CREATE TABLE mytable_vti_1(
    A integer,
    B float,
    C date,
    X float,
    Y float,
    Z float,
    T datetime year to second,
    V float)
USING DSChipAccess(
    basetable='mytable',
    keylist='(C)',
    dschipcolumn='D',
    schematext='(maxtuples 1000, T date,X double .1,Y double
.1,Z double .1,V double .1)',
    usescalarkeys='1'
);
```

Note:

- 1) “basetable=” gives the name of the base table upon which this virtual table has been defined.
- 2) “dschipcolumn” identifies the DSChip column that is to be accessed by this VTI table. If there is just one DSChip column in the base table then this parameter can be omitted.
- 3) “keylist=” identifies the columns that appear, in order, in the (existing) base table index that is to be used when this virtual table is queried. If a composite index was to be used, then there would be multiple entries in this field.

```
CREATE INDEX mytable_c_idx ON mytable(c,a) USING btree;
...
CREATE TABLE mytable_vti_1(
...
USING DSChipAccess(
```

```
...  
    keylist=' (C,A) '  
    ...  
);
```

- 4) “usescalarkeys='1'” indicates that the index to be used is a B-tree index.
- 5) “schematext” indicates the schema of the DSChip. The format of this parameter is the same as the format produced by the [DSChipSchema](#) function (surrounded by ‘ ( and ) ’). Either “schematext” or “schemasource” (described next) must be specified.
- 6) “schemasource” identifies a table containing a single row with a single column (lvarchar), holding the text that would otherwise appear in the “schematext” parameter. It is sometimes necessary to use “schemasource” instead of “schematext”, since VTI limits the total length of the parameter string to 255 characters. See [Loading into a VTI Table](#) for an example of using this parameter.
- 7) Any column names used in the virtual table definition must exist either as base table columns (in this case A, B, or C) or DBXten DSChip internal columns (in this case, X,Y,Z,T, or V), but not both.

#### **Defining a VTI Virtual Table for Queries on DSChip Extents (e.g. X/Y/Z/T)**

The SQL statement in the following example can be used to create a virtual table based on mytable; it specifies that a functional R-tree index has been built on D (the DSChip column), that the dimensions of this multidimensional index are X, Y, and Z (in that order), and that the name of the R-tree index function is ‘pos\_box’.

**Example**

```
CREATE TABLE mytable_vti_1(  
    A integer,  
    B float,  
    C date,  
    X float,  
    Y float,  
    Z float,  
    T datetime year to second,  
    V float)  
USING DSChipAccess(  
    basetable='mytable',  
    dschipcolumn='D',  
    keylist='pos_box(d):(x,y,z)',  
    schematext='(maxtuples 1000, T date,X double .1,Y double  
.1,Z double .1,V double .1)',  
    usescalarkeys='0'  
);
```

**Note:**

- 1) “basetable=” gives the name of the base table upon which this virtual table has been defined.
- 2) “dschipcolumn” identifies the DSChip column that is to be accessed by this VTI table. If there is just one DSChip column in the base table then this parameter can be omitted.
- 3) “keylist=” identifies the functional index to be used when this virtual table is queried. The portion after the ‘:’ indicates the names of the columns that correspond to the dimensions of the DSBox that gets returned by the functional index. VTI uses this information to convert clauses like

“WHERE x < ... AND x > ... AND y < ... AND y > ...”

to the appropriate DSBox overlaps test.

- 4) The “usescalarkeys='0'” option indicates that the index to be used is *not* a B-tree index, i.e., it is an R-tree index. Alternatively, for R-tree indexes the “usescalarkeys=” term can just be omitted.
- 5) “schematext” indicates the schema of the DSChip. The format of this parameter is the same as the format produced by the [DSChipSchema](#) function (surrounded by ‘( and )’). Either “schematext” or “schemasource” (described next) must be specified.
- 6) “schemasource” identifies a table containing a single row with a single column (lvarchar), holding the text that would otherwise

appear in the “schematext” parameter. It is sometimes necessary to use “schemasource” instead of “schematext”, since VTI limits the total length of the parameter string to 255 characters. See [Loading into a VTI Table](#) for an example of using this parameter.

- 7) Any column names used in the virtual table definition must exist either as base table columns (in this case A, B, or C) or DBXten DSChip internal columns (in this case, X,Y,Z,T, or V), but not both.

### **B-Tree Indexes on DSChip Dimension Columns**

If queries are to be based on just a single dimension of a DSChip then using a B-Tree on the minimum or maximum value of that dimension in each DSChip is an option. The following example builds a VTI virtual table geared for cases where queries are of the form:

“SELECT ... WHERE X < ... AND X > ...” or

“SELECT ... WHERE Y < ... AND Y > ...”

Note that this example requires that the base table have additional columns that store the minimum and maximum values for the corresponding DSChip dimension. These would likely be kept in synch with the DSChip via insert triggers<sup>17</sup>.

---

<sup>17</sup> For example, B\_min would be set to `DMinAsString(D, 'B')`

**Example**

```
CREATE TABLE mytable2 (A integer,
                        B float,
                        C date,
                        X_min float, -- new column
                        X_max float, -- new column
                        Y_min float, -- new column
                        Y_max float, -- new column
                        D DSChip);

CREATE INDEX mytable2_Xmin_idx on mytable2(X_min) using btree;
CREATE INDEX mytable2_Xmax_idx on mytable2(X_max) using btree;
CREATE INDEX mytable2_Ymin_idx on mytable2(Y_min) using btree;
CREATE INDEX mytable2_Ymax_idx on mytable2(Y_max) using btree;

CREATE TABLE mytable2_vti(
    A integer,
    B float,
    C date,
    X float,
    Y float,
    Z float,
    T datetime year to second,
    V float)
USING DSChipAccess(
    basetable='mytable',
    dschipcolumn='D',
    keylist='(X,Y)',
    schematext='(maxtuples 1000, T date,X double .1,Y double
.1,Z double .1,V double .1)',
    usescalarkeys='1'
);
```

**Note:**

- 1) “basetable=” gives the name of the base table upon which this virtual table has been defined.
- 2) “keylist=” identifies the individually indexed DSChip dimensions. This is set equal to the name of one or more of the DSChip internal columns. It is assumed that there are two B-Tree indexes on each of these dimensions: one on the minimum value of the dimension in the DSChip, the other on the maximum value of the dimension.
- 3) The “usescalarkeys='1'” option indicates that the index to be used is a B-tree index.
- 4) “schematext” indicates the schema of the DSChip. The format of this parameter is the same as the format produced by the [DSChipSchema](#) function (surrounded by ` ( and ) `). Either “schematext” or “schemasource” (described next) must be specified.

- 5) “schemasource” identifies a table containing a single row with a single column (lvarchar), holding the text that would otherwise appear in the “schematext” parameter. It is sometimes necessary to use “schemasource” instead of “schematext”, since VTI limits the total length of the parameter string to 255 characters. See [Loading into a VTI Table](#) for an example of using this parameter.
- 6) Any column names used in the virtual table definition must exist either as base table columns (in this case A, B, or C) or DBXten DSChip internal columns (in this case, X,Y,Z,T, or V), but not both.

### **Using VTI - Current Limitations**

The DBXten – VTI query interface has the following current limitations:

- 1) Columns of VTI virtual tables are restricted to the following data types:
  - a. char
  - b. datetime
  - c. float
  - d. 32 bit integer
  - e. 64 bit integer
  - f. serial
  - g. serial8
  - h. smallint
  - i. smallfloat
  - j. varchar

So, for example, if the base table has a column X of type “money” then any VTI virtual tables based on that table must omit column X.

- 2) Each column name in a VTI virtual table is the same as the name of a conventional column or a DSChip internal column (i.e., the association between base table columns and virtual table columns is by name). Hence, it is not possible to define a VTI table on a base table where a DSChip has an internal column name that is the same as the name of a conventional column in the table.
- 3) While base tables can have multiple DSChip columns, each VTI table can reference just one of these columns (but you can have multiple VTI tables defined on a single base table).



## 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 33).

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

### Example

```
> DELETE FROM measurementBlocks
WHERE overlap(DSAsBoxString(measurement_block,
                             'datetime,latitude,longitude')::DSBox,
              DSRangeToBox('datetime 2008-01-01 12:00:00 2008-
01-01 14:00:00,latitude * *,longitude * *')::DSBox);

2 row(s) deleted.
```

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: Troubleshooting Guide

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

### 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.

**"attempt to open internal query connection failed"**

Something is wrong with the server.

### 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 `DSChipVarSetString` 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 `DSChipVarGetString` 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 `DSChipVarSetDouble` or `DSChipVarSetInt` 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 `DSChipVarGetDouble` or `DSChipVarGetInt` 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 [DSChipExtract](#) 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 5<sup>th</sup> 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 5<sup>th</sup> 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 [columnName](#) clause of a function (see [DSChipExtract](#) on page 71). For example,

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

instead of

```
SELECT DSChipExtract(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 [DSChipExtract](#) on page 71). 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 [DSChipExtract](#) on page 71.

"schema did not contain any column definitions"

This can be caused by passing an incomplete DSChip schema specification to DSChipNew() – see [DSChipExtract](#) on page 71. 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.

"truncated text when reading column name"

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.

"bad tolerance value for column *column\_name*, was *numeric\_value*"  
Negative tolerance values are not allowed.

"bad numcolumns must be > 0"

The number of columns specified by text wasn't a positive integer.

"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 [DSChipExtract](#) on page 71). For example,

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

instead of

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

## Bad DateTime Data

**"time offset must be less than 24 hours, either direction"**

The timezone value must be between -24 and +24.

**"unrepresentable time string"**

DBXten is currently only able to represent datetime values that are in the range Jan 1 1902 to Dec 31 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 DSRangeToBox('datetime 1970-01-01 01:00:00 1970-01-  
01 01:00:01 badstuff, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('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 DSRangeToBox('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 DSRangeToBox('datetime 1970/01-01 01:00:00 1970-01-  
01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-13-01 01:00:00 1970-01-  
01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-01/01 01:00:00 1970-01-  
01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-02-32 01:00:00 1970-01-  
01 01:00:00, latitude 49.0 49.01');
```

ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-01-01/01:00:00 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-02-01 32:00:00 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-01-01 01/00:00 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-01-01 01:70:00 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-02-01 01:00:80 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-02-01 01:00:01.1234567 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-02-01 01:00:01+15 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox 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 DSRangeToBox('datetime 1970-02-01 01:00/01 1970-01-01 01:00:00, latitude 49.0 49.01');  
ERROR: DSRangeToBox user error: Unexpected text '/01'  
following timeStamp '1970-02-01 01:00/01'
```

**"bad date in was string"**

This is a catch-all error for invalid timestamp values.

## **DSQueryToString Specific Errors**

**"bad query:<query\_text>"**

Query passed to DSQueryToString had an error in it. Try executing the query directly in dbaccess to see what the error was.

**"attempt to read row of query failed"**

Try executing the query directly in dbaccess to see what the error was.

**"query column column\_name (column\_position) had type type-name, which is not supported. Try casting to a supported type."**

Only types boolean, smallint, integer, double precision, float, smallfloat, char, text, varchar, lvarchar, and DSChip are supported.

**"saw null chip column in query"**

The query is not allowed to produce rows containing null chips.

**"no chip column in query"**

The result row from the query did not contain a chip.

## **Others**

**"attempt to get column column\_number as integer"**

The DSGetInteger function was called on a non-integer column.

**"attempt to get column column\_number as date"**

The DSGetDate function was called on a non-date column.

"changing number of columns in chips (was *chip\_column\_count*, now *row\_column\_count*)"

Attempt to add a row to a DSChip that had a different number of columns than the chip.

"nonconstant column types in column *column\_position*"

Attempt to add a row to a DSChip that had the wrong type for one its columns.

"attempt to compress already compressed or empty DSChip"

An attempt was made to convert a DSChip to its serialized format, but the DSChip had 0 rows. If this error message is seen with a non-empty DSChip, contact BCS.

## **Blob Space Issues**

The follow denote a problem with your blob space. Ask your database administrator to examine the blob space.

"unable to increment reference count on blob, blob space may be corrupt"

"mi\_lo\_spec\_init failed. Blob space problem with column?"

"mi\_lo\_colinfo\_by\_ids failed."

"unable to create large object of size *size\_in\_bytes*, check your smart blob space"

"unable to write data into newly created blob"



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

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

Function	Page
FUNCTION DSAsBoxString( <i>existingChip</i> DSChip, <i>columnNames</i> char) RETURNS lvarchar	64
FUNCTION DSChipAppendRow( <i>existingChip</i> DSChip, <i>valuesAsString</i> CHAR) RETURNS DSChip	33
FUNCTION DSChipExtract( <i>existingChip</i> DSChip, <i>rangeSpec</i> char, <i>columnNames</i> char) RETURNS DSChip	71
FUNCTION DSChipNew( <i>schema</i> CHAR) RETURNS DSChip	31
FUNCTION DSChipNumMatches( <i>existingChip</i> DSChip, <i>rangeSpec</i> char) RETURNS integer	70
FUNCTION DSChipSchema( <i>existingChip</i> DSChip) RETURNS char	56
FUNCTION DSCompressInfo( <i>existingChip</i> DSChip) RETURNS lvarchar	63
FUNCTION DSDistinct( <i>existingChip</i> DSChip, <i>columnNames</i> char) RETURNS DSChip	58
FUNCTION DSHasVariables( <i>existingChip</i> DSChip, <i>columnNames</i> char) RETURNS boolean	61
FUNCTION DSMaxAsString( <i>existingChip</i> DSChip, <i>columnNames</i> char) RETURNS lvarchar	62
FUNCTION DSMinAsString( <i>existingChip</i> DSChip, <i>columnNames</i> char) RETURNS lvarchar	63

**BCS DBXTEN DATABLEDE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

FUNCTION DSNumFilledRows( <i>existingChip</i> DSChip) RETURNS integer	62
FUNCTION DSNumVars( <i>existingChip</i> DSChip) RETURNS integer	61
FUNCTION DSQueryToStrings( <i>sqlSelectStatement</i> lvvarchar) RETURNS setof(lvvarchar)	56
FUNCTION DSRangeToBox( <i>rangeSpec</i> char) RETURNS lvvarchar	64



## 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 Base Data Types

These represent the data types that can be used for the various columns of DSChip's.

<code>DSVarTypeINT</code>	a 32 bit integer
<code>DSVarTypeDOUBLE</code>	a 64 bit floating point number
<code>DSVarTypeDATE</code>	a timestamp, capable of storing a date-time value between Jan 1, 1902 and Dec 31, 2037.
<code>DSVarTypeSTRING</code>	a variable length character string
<code>DSVarTypeINT8</code>	a 64 bit integer

#### DBXten C Data Types

There are two DBXten-specific C data types used in C programs:

- `DSChip`: used to store the internal representation of a single `DSChip` instance within the database.
- `DSChipVar`: used to store the internal representation of a single column within a single tuple of a `DSChip`.

### 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 maxRows);
```

### **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 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*<sup>TH</sup> ROW OF CHIPVAR AS A STRING.**

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

**GET THE *ROW*<sup>TH</sup> ROW OF CHIPVAR AS A DOUBLE.**

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

**GET THE *ROW*<sup>TH</sup> ROW OF CHIPVAR AS A 32 BIT INTEGER.**

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

**GET THE *ROW*<sup>TH</sup> 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.)

**DECOMPRESS THE COLUMNS IN A DSCHIP.**

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

### **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 *);
```

**OUTPUT TEXT TO DEBUG FILE.**

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

This function outputs text to the /tmp/DBXten\_debug.log file on the server.

**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  
1970 TO A STRING IN THE FORMAT YYYY-MM-DD HR:MN:SS  
IN THE LOCAL TIMEZONE.**

```
void DSDoubleToTimeString(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 1970.**

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





## Appendix C: The Java API

The Java API is described in the Javadocs stored in  
<DBXTENDIR>/javadocs/index.html.

An online version can be found at:

<http://www.barrodale.com/software/DBXtenClients/javadocs/>





## 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 IBM Informix server resides or it can be run on a different machine (and communicate with the IBM Informix server machine via TCP/IP).

### Downloading the CSV File Reader Utility

An executable for the `csvChipLoader` program (called `csvChipLoaderIfx` for the Informix version) is supplied with DBXten, in `<DBXTENDIR>/bin/csvChipLoaderIfx`

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

```
unzip csvChipLoaderIfx.zip
```

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

- `csvChipLoaderIfx`: the actual program.
- `csvLoaderIfx.c`: C code produced by the ESQL pre-processor.
- `csvLoaderIfx.ec`: the C-ESQL source code.
- `exp_chk.ec`: C-ESQL source code for error processing.
- `Makefile`: Makefile for making the program
- `schema.sql`: an SQL script for defining a destination table called `sampleChips`.

- `sample.csv`: a sample .csv 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, edit the `Makefile`, updating the definitions of `DBXTENDIR` and `INFORMIXDIR` if necessary. Then execute the `make` program. This will create an executable called `csvChipLoaderIfx`.

## Running the CSV File Reader Utility

The command line arguments for `csvChipLoaderIfx` are:

<u>Command line argument</u>	<u>Meaning</u>
<code>-rowsPerChip <i>value</i></code>	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.
<code>-monitor <i>count</i></code>	Output a progress message every <i>count</i> DSChip's. The default is to produce no output.
<code>-i <i>path</i></code>	The name of the file to be loaded. This name must be resolvable in the environment under which the CVS File Reader is running <sup>18</sup> .
<code>-indelimit <i>delimiter</i></code>	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 ','.
<code>-d <i>database</i></code>	The name of the IBM Informix database to connect to. There is no default.
<code>-t <i>tablename</i></code>	The name of the IBM Informix table containing the DSChip column to be inserted into. There is no default.
<code>-c <i>columnname</i></code>	The name of the DSChip column to be inserted into. The default is "chip".

---

<sup>18</sup> 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.

*column\_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”, “double”, “datetime”, and “string”. As a convenience, the following can also be used for the column type: “integer”, “float”, “time”, and “date”.

The precision is optional and can only be specified for types datetime, float, and double. See the [discussion](#) on page 32 for a description of the precision 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
1,0.291066,2008-1-1 0:0:1
2,0.845814,2008-1-1 0:0:2
3,0.152208,2008-1-1 0:0:3
4,0.585537,2008-1-1 0:0:4
5,0.193475,2008-1-1 0:0:5
6,0.810623,2008-1-1 0:0:6
7,0.173531,2008-1-1 0:0:7
8,0.484983,2008-1-1 0:0:8
9,0.151863,2008-1-1 0:0:9

% wc sample.csv
10000 20000 305420 sample.csv

% dbaccess -e demodb schema.sql
Database selected.

DROP TABLE sampleChips;
206: The specified table (samplechips) is not in the database.

111: ISAM error: no record found.
Error in line 1
```

**BCS DBXTEN DATABLED FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

Near character position 22

```
CREATE TABLE sampleChips (  
    chip DSChip  
);
```

Table created.

Database closed.

```
% csvChipLoaderIfx -t samplechips -d demodb -i sample.csv -  
monitor 1 id:integer value:double when:datetime
```

lines read= 10000, chips written = 10

```
% dbaccess demodb -
```

```
> SELECT count(*) FROM sampleChips;
```

```
    (count(*))
```

```
        10
```

1 row(s) retrieved.

```
> SELECT FIRST 1 * FROM sampleChips;
```

```
chi  maxtuples=1000, filledtuples=1000, numcolumns=3; id, integer  
p    , 0; value, float, 0; when, datetime, 0; 0, 0.2377880000000000, 200  
8-01-0100:00:0.000000; 1, 0.2910660000000000, 2008-01-01 00:  
00:1.000000; 2, 0.8458140000000000, 2008-01-01 00:00: 2.0000  
00; 3, 0.1522080000000000, 2008-01-01 00:00:3.000000; 4, 0.585  
537000000000, 2008-01-0100:00:4.000000; 5, 0.19347 50000000  
00, 2008-01-01 00:00:5.000000; 6, 0.8106230000000000, 2008-0  
1-01 00:00:6.000000; 7, 0.1735310000000000, 2008-01-01 00:0  
:7.000000; 8, 0.4849830000000000, 2008-01-01 00:00:8.000000;  
9, 0.1518630000000000, 2008-01-01 00:00:9.000000; 10, 0.3669  
...  
...  
8720780000000000, 2008-01-01 00:16:33.000000; 994, 0.082055  
0000000000, 2008-01-01 00:16:34.000000; 995, 0.23 6118000000  
000, 2008-01-01 00:16:35.000000; 996, 0.2425370000000000, 200  
8-01-01 00:16:36.000000; 997, 0.2856960000000000, 2008-01-01  
00:16:37.000000; 998, 0.6057540000000000, 2008-01-01 00:16:3  
8.000000; 999, 0.9949840000000000, 2008-01-01 00:16:39.00000  
0
```

1 row(s) retrieved.



## Appendix F: NetCDF File Reader Utility

### Overview

The netCDF file reader utility (`ncToDBXtenIfx`) is a program that reads an arbitrary netCDF file and loads it into a table of DSChip's. The `ncToDBXtenIfx` 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` program (called `ncToDBXtenIfx` for the Informix version) is supplied with DBXten, in  
`<DBXTENDIR>/bin/ncToDBXtenIfx`

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

```
unzip ncToDBXtenIfx.zip
```

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

- \*.c and \*.ec: several C and C-ESQL source files

- `ncToDBXtenIfx`: the actual executable.
- `testschema.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.cd1.head`: a text version of the first 100 lines 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 `ncToDBXtenIfx` program has been tested with version 3.6 of the library.

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

## Running the NetCDF File Reader Utility

The command line arguments for `ncToDBXten` are:

<u>Command line argument</u>	<u>Meaning</u>
<code>-i path</code>	The name of the file to be loaded. This name must be resolvable in the environment under which the NetCDF File Reader is running <sup>19</sup> .
<code>-d database</code>	The name of the IBM Informix database to connect to. There is no default.
<code>-rc configurationFile</code>	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:

---

<sup>19</sup> 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.

**BCS DBXTEN DATABLADE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

```
% 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 testschema.sql

DROP TABLE sampleGrid;

CREATE TABLE sampleGrid(chip dschip, id bigserial);

% dbaccess -e demodb testschema.sql

Database selected.

DROP TABLE sampleGrid;
206: The specified table (samplegrid) is not in the database.

111: ISAM error: no record found.
Error in line 1
Near character position 21

CREATE TABLE sampleGrid(chip dschip, id bigserial);
Table created.

Database closed.

% cat test.rc
```

**BCS DBXTEN DATABLEADE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

```
#
# The chip column gets values with internal columns of time,
# lat, lon and tos.
#
sampleGrid=time:1;1,lat:5;0.5,lon:4;1;,tos;0.1

% ./ncToDBXten -d demodb -i test.nc -rc test.rc

% dbaccess demo -

> SELECT count(*) from sampleGrid;

      (count(*))

      36720

1 row(s) retrieved.

demo=> SELECT FIRST 1 * FROM sampleGrid;

chi  maxtuples=20,filledtuples=20,numcolumns=4;time,float,1;1
p    at,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,-7
    8.5,1,100000002004087734272.0;15,-78.5,4,100000002004087
    734272.0;15,-78.5,5,100000002004087734272.0;15,-78.5,7,1
    00000002004087734272.0;15,-77.5,1,10000000200408 7734272
    .0;15,-77.5,4,100000002004087734272.0;15,-77.5,5,1000000
    02004087734272.0;15,-77.5,7,100000002004087734272.0;15,-
    76.5,1,100000002004087734272.0;15,-76.5,4,10000000200408
    7734272.0;15,-76.5,5,100000002004087734272.0;15,-76.5,7,
    100000002004087734272.0;15,-75.5,1,100000002004087734272
    .0;15,-75.5,4,100000002004087734272.0;15,- 75.5,5,100000
    002004087734272.0;15,-75.5,7,100000002004087734272.0

id    1

1 row(s) retrieved.
```



## Appendix G: CSV File Reordering Utility

### Overview

As described earlier in [Indexing DSChip's](#) (see page 36), 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 53), leaving less to be done by the relatively slow "[Tuple Filtering](#)" (see page 54). 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  $R \times BF_1 \times BF_2 \times \dots \times BF_n$  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  $BF_j$  groups so that all the column  $j$  values in the first group are  $\leq$  all the column  $j$  values in the second group, all the column  $j$  values in the second group are  $\leq$  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 (BF<sub>1</sub>) groups of  $3 \times 4 = 12$  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<sub>2</sub> = 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:

**BCS DBXTEN DATABLEADE FOR IBM INFORMIX (LINUX  
VERSION) PROGRAMMER'S GUIDE**

Initial	Sort	Partition	Sort	Partition	Sort	Final
1,24	1,24	1,24	12,2	12,2	9,8	<b>9,8</b>
2,22	2,22	2,22	11,4	11,4	10,6	<b>10,6</b>
3,20	3,20	3,20	10,6	10,6	11,4	<b>11,4</b>
4,18	4,18	4,18	9,8	9,8	12,2	<b>12,2</b>
5,16	5,16	5,16	8,10	8,10	5,16	<b>5,16</b>
6,14	6,14	6,14	7,12	7,12	6,14	<b>6,14</b>
7,12	7,12	7,12	6,14	6,14	7,12	<b>7,12</b>
8,10	8,10	8,10	5,16	5,16	8,10	<b>8,10</b>
9,8	9,8	9,8	4,18	4,18	1,24	<b>1,24</b>
10,6	10,6	10,6	3,20	3,20	2,22	<b>2,22</b>
11,4	11,4	11,4	2,22	2,22	3,20	<b>3,20</b>
12,2	12,2	12,2	1,24	1,24	4,18	<b>4,18</b>
13,1	13,1	13,1	13,1	13,1	13,1	<b>13,1</b>
14,3	14,3	14,3	14,3	14,3	14,3	<b>14,3</b>
15,5	15,5	15,5	15,5	15,5	15,5	<b>15,5</b>
16,7	16,7	16,7	16,7	16,7	16,7	<b>16,7</b>
17,9	17,9	17,9	17,9	17,9	17,9	<b>17,9</b>
18,11	18,11	18,11	18,11	18,11	18,11	<b>18,11</b>
19,13	19,13	19,13	19,13	19,13	19,13	<b>19,13</b>
20,15	20,15	20,15	20,15	20,15	20,15	<b>20,15</b>
21,17	21,17	21,17	21,17	21,17	21,17	<b>21,17</b>
22,19	22,19	22,19	22,19	22,19	22,19	<b>22,19</b>
23,21	23,21	23,21	23,21	23,21	23,21	<b>23,21</b>
24,23	24,23	24,23	24,23	24,23	24,23	<b>24,23</b>



## 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:

<code>[-indelim <i>delimiter_character</i>]</code>	Optional delimiter character, default is ' '. 'tab' can be used to indicate tab as the delimiter.
<code>-i <i>input_file</i></code>	Input file name (mandatory).
<code>[-verbose]</code>	Do you want verbose output? If so, messages are written to <code>stderr</code> , indicating what tiling is currently being tried.
<code>[-csv]</code>	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.
<code>(<i>dimension field</i>)+</code>	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 file. 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 124), 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
```