

# PostgreSQL Performance is Enhanced with DBXten

## Table of Contents

Executive Summary .....	1
Introduction.....	2
Computing Environment.....	3
Server .....	3
PostgreSQL.....	3
The Data.....	3
Native PostgreSQL Table .....	4
Native PostgreSQL B-Tree Indexes.....	5
DBXten Table .....	5
DBXten Index .....	5
Loading Trials .....	6
Native PostgreSQL Loading Trials.....	6
DBXten Loading Trials.....	6
Loading Time and Space Comparison.....	7
Query Trials .....	7
Native PostgreSQL Queries .....	8
DBXten Queries.....	9
Query Time Comparison.....	9
Concurrent Query Testing.....	12
Appendix A – Abridged NetCDF File Header (ncdump).....	14
Appendix B – Source of pgFetchData Program .....	16

## Executive Summary

Having benchmarked our [DBXten DataBlade](#) on a powerful and expensive IBM System x computer (documented in a previous 2010 report<sup>1</sup>), we wanted to explore what performance advantages could be offered by this BCS database plug-in on a commonly available and inexpensive office PC. Our objective was to assess the loading and retrieval performance of PostgreSQL, *with and without* DBXten, on a simple workstation.

These tests on the PC showed that:

1. Using the DBXten DataBlade required almost 60 times less storage space than “native PostgreSQL” (i.e., PostgreSQL without DBXten).
2. The DBXten DataBlade reduced load times of indexed tables to less than 10% of the load times taken by native PostgreSQL.
3. The automatic data compression in DBXten actually improved multi-dimensional query performance *by more than an order of magnitude*.

Since databases are usually multi-user intensive platforms, we also examined DBXten’s performance in concurrent query testing ranging from 2 to 25 simulated users. We found that as we gradually increased the number of users up to 25, the performance gains over native PostgreSQL were always *more than an order of magnitude!*

These are truly impressive results. Keep in mind that the only change between the comparative tests was addition of the DBXten DataBlade. *No* additional customization was necessary to achieve these results.

The DBXten DataBlade provides compact storage, fast loads, and quick queries for large datasets on a wide range of computing platforms, from inexpensive desktop PC’s up to and including large and costly multi-server environments.

---

<sup>1</sup> [http://www.barrodale.com/bcs/whitePapers/Impact\\_of\\_DBXten\\_on\\_IBM\\_Informix\\_Performance.pdf](http://www.barrodale.com/bcs/whitePapers/Impact_of_DBXten_on_IBM_Informix_Performance.pdf)

## Introduction

In this report we compare the load and query performance of PostgreSQL with and without the Barrodale Computing Services (BCS) [DBXten DataBlade](#)<sup>2</sup>. Computations with a scientific dataset occupying up to 440 million database rows were run on a BCS workstation (an AMD Athlon 64 dual core). This benchmarking was conducted after we had completed similar tests<sup>3</sup> with IBM Informix on the same BCS workstation.

Of primary interest to us was how the performance of DBXten would scale as the number of rows in the database increased. For each of eight increasing table sizes ranging from 55 million to 440 million rows, tables were loaded and the same set of five queries was executed, firstly by a single user and then by (simulated) multiple concurrent users. The PostgreSQL load and query performances, with and without DBXten, were then compared. In short, DBXten scaled very well on the office PC at BCS. Loading with DBXten required much less space (by a factor of 59 — largely because of a factor of almost 300 in index space reduction) and load times were faster by a factor of about 12. Individual query times with DBXten were faster by factors ranging from 3.2 to as much as 100, with an average factor of 13.2. The DBXten cumulative query times for each of the eight tables were more than an order of magnitude faster than for native PostgreSQL. Finally, performance trials with up to twenty five (simulated) concurrent users were run on the 220 million row table. These trials demonstrated that enhancing PostgreSQL with DBXten provided answers to queries that were always more than order of magnitude faster (sometimes more than 50 times faster) when two or more concurrent users were querying on the BCS workstation.

The body of this report contains technical specifications of the comparison testing and the computational results obtained, and the two Appendices provide a sample data file header information and the querying source code for when the DBXten C API was used. For readers of this report who wish to focus mainly on the outcomes of our testing, we direct attention to the Tables and Figures on pages 7–12. Finally, a PDF version of the Programmer's Guide (Linux Version) for the DBXten database extension for PostgreSQL can be downloaded<sup>4</sup> at no charge.

---

<sup>2</sup> Update: In December 2011 BCS was awarded US Patent 8077059 for DBXten.

<sup>3</sup> [http://www.barrodale.com/bcs/whitePapers/Using\\_IBM\\_Informix\\_on\\_a\\_workstation\\_with\\_DBXten.pdf](http://www.barrodale.com/bcs/whitePapers/Using_IBM_Informix_on_a_workstation_with_DBXten.pdf)

<sup>4</sup> [http://www.barrodale.com/bcs/docs/DBXten\\_Programmer\\_Guide\\_PostgreSQL\\_Version.pdf](http://www.barrodale.com/bcs/docs/DBXten_Programmer_Guide_PostgreSQL_Version.pdf)

## Computing Environment

### Server

**CPUs:** 1 AMD Athlon 64 X2 3600+ Dual-Core, 2.0 GHz, 2x256KB L2Cache, 64 bit

**RAM:** 2GB DDR2 667MHz

**OS:** Linux kernel 2.6.18 (Centos 5)

**Disk:** 1x160GB Seagate Barracuda 7200 rpm 3GB/s SATA internal hard drive with cooked filesystem

### PostgreSQL

PostgreSQL version 8.4.2 was used throughout.

## The Data

The data was generated using a high resolution model developed by the Ocean Circulation and Climate Advanced Modelling Project, using the OCCAM data selector at <http://www.noc.soton.ac.uk/JRD/OCCAM/EMODS/select.php>. The parameters shown in the following image were used to direct the generation of data:

**National Oceanography Centre, Southampton**  
UNIVERSITY OF SOUTHAMPTON AND NATURAL ENVIRONMENT RESEARCH COUNCIL

**OCCAM**  
OCEAN CIRCULATION and CLIMATE ADVANCED MODELLING PROJECT

EMODS DATASERVER    ABOUT THE DATASERVER    FAQ    CONTACT US

**Email Address:**

**Model domain:**

**Model resolution:**

**Model run:**

**Model dataset:**

**2D Model variables:**

**3D Model variables:**

**Longitude:** Start:   End:   ° E

**Latitude:** Start:   End:   ° N

**Model levels:** Start:   End:

**Model months:** Start:  1996  End:  2004  Increment (months):

**Output format:**

**Output type:**

**Estimated size:-**

W3C CSS  W3C HTML 4.01

Figure 1: OCCAM Data Selector, with values filled in

This request<sup>5</sup> resulted in the generation of 105 netCDF files, one for each month between April 1996 and December 2004. The header information from one of these netCDF files is provided in [Appendix A](#). Each of the netCDF files was then converted to a comma-separated (CSV) text file, with each line containing values from the six variables:

- TIMESTEP
- DEPTH
- LATITUDE\_T
- LONGITUDE\_T
- POTENTIAL\_TEMPERATURE\_\_MEAN\_
- SALINITY\_\_MEAN\_

### ***Native PostgreSQL Table***

The native PostgreSQL tables had the following schema:

<b>Column</b>	<b>Type</b>
timeval	integer
depth	float
latitude	float
longitude	float
temperature	float
salinity	float

**Table 1: Native PostgreSQL Table Schema**

The tables were defined using the following SQL:

```
CREATE TABLE "barrodale".occam_conventional_ind_&N
(
  timeval integer,
  depth float,
  latitude float,
  longitude float,
  temperature float,
  salinity float
);
```

where &N is used to distinguish the number of rows in the various cases.

---

<sup>5</sup> The request was actually submitted as three individual requests, covering different periods of time.

## ***Native PostgreSQL B-Tree Indexes***

For each PostgreSQL table four B-Tree indexes were built, one on each of the spatio-temporal columns timeval, latitude, longitude, and depth. The following SQL was used to define indexes on the tables:

```
CREATE INDEX occam_conventional_ind_${N}_depth_idx
  ON occam_conventional_ind_${N}(depth);

CREATE INDEX occam_conventional_ind_${N}_timeval_idx
  ON occam_conventional_ind_${N}(timeval);

CREATE INDEX occam_conventional_ind_${N}_latitude_idx
  ON occam_conventional_ind_${N}(latitude);

CREATE INDEX occam_conventional_ind_${N}_longitude_idx
  ON occam_conventional_ind_${N}(longitude);
```

## ***DBXten Table***

For the DBXten trials the table had a single column, a so-called “DSChip” type column with the following schema definition:

<b>Column</b>	<b>Type</b>	<b>Precision</b>
timeval	integer	
depth	float	.01
latitude	float	.001
longitude	float	.001
temperature	float	.01
salinity	float	.000001

**Table 2: DBXten DSChip Schema**

One of the features of DBXten is that it is able to exploit limited precision in a particular dataset. The values shown above were chosen to correspond with the actual precision of the data in the input dataset (as converted from the original netCDF files).

The following SQL was used to define the table:

```
CREATE TABLE occam_chips_${N}_0_TM(occamchip dschip);
```

## ***DBXten Index***

The SQL overleaf was used to define a GiST index on the DSChip’s:

```
CREATE INDEX occam_chips_${N}_0_TM_idx ON
  occam_chips_${N}_0_TM
  USING gist((DSAsCubeString(occamchip,
    'timeval,latitude,longitude,depth'::bpchar)::cube));
```

## Loading Trials

### *Native PostgreSQL Loading Trials*

The following table defines the loading trials that were performed on native PostgreSQL tables:

Case Name	Rows (N)	netCDF Files	Indexes	Pre-indexed?
55M_B_R	55,109,736	4	4 B-Tree	yes
83M_B_R	82,664,604	6	4 B-Tree	yes
110M_B_R	110,219,472	8	4 B-Tree	yes
138M_B_R	137,774,340	10	4 B-Tree	yes
165M_B_R	165,329,208	12	4 B-Tree	yes
193M_B_R	192,884,076	14	4 B-Tree	yes
220M_B_R	220,438,944	16	4 B-Tree	yes
440M_B_R	440,877,888	32	4 B-Tree	yes

**Table 3: Loading Trials – Native PostgreSQL**

Each of these loads was performed using `psql` “\copy” commands applied to each [converted](#) netCDF file in turn. Upon all of the netCDF files for the particular case, the following SQL command was run using `psql`:

```
vacuum full analyze occam_chips_${N}_0_TM;
```

### *DBXten Loading Trials*

The following table defines the loading trials that were performed on the DBXten tables; the case names reflect the name of the comparable native PostgreSQL loading trial:

Case Name	Rows (N)	netCDF Files	Indexes	Pre-indexed?
55M_B_R	55,109,736	4	1 R-Tree	yes
83M_B_R	82,664,604	6	1 R-Tree	yes
110M_B_R	110,219,472	8	1 R-Tree	yes
138M_B_R	137,774,340	10	1 R-Tree	yes
165M_B_R	165,329,208	12	1 R-Tree	yes
193M_B_R	192,884,076	14	1 R-Tree	yes
220M_B_R	220,438,944	16	1 R-Tree	yes
440M_B_R	440,877,888	32	1 R-Tree	yes

**Table 4: Loading Trials – DBXten**

The DBXten tables were loaded with a CSV file loading client program, which is distributed with the DBXten product.

Note that each of the DBXten trials in the tables above have corresponding native PostgreSQL loading trials to which load times and space consumption can be compared. These comparisons were performed, and the results are presented in the next two sections.

### ***Loading Time and Space Comparison***

The following table presents the load times and table and index space consumption for each of the trials:

Case	Total Space (2K pages)			Index Space (2K pages)			Load Times (hh:mm:ss)		
	Native	DBXten	Ratio	Native	DBXten	Ratio	Native	DBXten	Ratio
55M_B_R	5,125,300	86,444	0.0169	3,064,604	10,860	0.003544	1:15:48	0:06:31	0.08594
83M_B_R	7,688,116	129,224	0.0168	4,597,080	16,284	0.003542	2:00:33	0:10:25	0.08643
110M_B_R	10,251,444	172,180	0.0168	6,130,064	21,712	0.003542	2:46:22	0:13:25	0.08066
138M_B_R	12,812,284	215,204	0.0168	7,660,564	27,140	0.003543	3:27:11	0:17:35	0.08484
165M_B_R	15,379,052	258,420	0.0168	9,196,992	32,568	0.003541	4:19:38	0:21:44	0.08369
193M_B_R	17,941,992	301,428	0.0168	10,729,588	37,992	0.003541	5:17:24	0:24:51	0.07827
220M_B_R	20,503,904	344,648	0.0168	12,261,156	43,420	0.003541	5:49:04	0:27:42	0.07936
440M_B_R	41,013,180	688,616	0.0168	24,527,700	86,840	0.003540	12:10:19	0:55:03	0.07539

**Table 5: Load Time and Space Comparison**

Total space ratios were very consistent ranging from .0168 to .0169. Load time ratios were also consistent, ranging from .075 to .086. The load time advantage of using DBXten generally increased with table size.

### **Query Trials**

Query performance, with and without DBXten, was assessed by running a variety of queries against each of the tables loaded as described in the previous sections.

For each of the cases listed in Table 5, five queries were issued in succession against the native PostgreSQL tables and then against the DBXten tables. In between each query invocation the PostgreSQL engine was stopped and restarted with

```
service postgresql restart
```

and file caches were cleared by issuing the following Linux commands:

```
sync
sync
echo 1 > /proc/sys/vm/drop_caches
```

Four of the queries – Q1 through Q4 – were basic spatial-temporal queries, requesting all the records that fell within a particular spatial-temporal “4D cube”. These 4D cubes varied in both size and shape. The fifth query – Q5 – requested all the records that fell

inside a particular cube and that also satisfied conditions on the two other columns. The dimensions of the cubes for each of the queries are shown in the following table:

Query	Time Bounds	Depth Bounds	Latitude Bounds	Longitude Bounds
Q1	1,297,200 to 1,297,300 (small)	266.8 to 266.9 (small)	-35 to -25 (medium)	65 to 75 (medium)
Q2	1,297,200 to 1,297,300 (small)	266.8 to 266.9 (small)	-56.25 to -56.15 (small)	64.25 to 64.3 (small)
Q3	1,000,000 to 10,000,000 (large)	100,000 to 400,000 (large)	-56.25 to -56.15 (small)	64.25 to 64.3 (small)
Q4	1,000,000 to 10,000,000 (large)	200 to 3,000 (medium)	-55 to -50 (medium)	65 to 70 (medium)
Q5	1,279,700 to 1,279,700 (small)	3,000 to 4,000,000 (large)	-35 to -15 (large)	70 to 80 (large)

**Table 6: Query Cube Sizes and Shapes**

For Q5, salinity was further restricted to values  $< -.00055$  and temperature was further restricted to values  $> 27.5$ .

The number of rows returned by each of the queries, for each of the cases, is shown in the following tables:

Case	Q1	Q2	Q3	Q4	Q5
55M_B_R	14,400	1	76	72,000	2
83M_B_R	14,400	1	114	108,000	2
110M_B_R	14,400	1	152	144,000	2
138M_B_R	14,400	1	190	180,000	2
165M_B_R	14,400	1	228	216,000	2
193M_B_R	14,400	1	266	252,000	2
220M_B_R	14,400	1	304	288,000	2
440M_B_R	14,400	1	608	576,000	2
1450M_B_R	14,400	1	1,995	1,890,000	2

**Table 7: Number of Rows Returned by Each Query**

### ***Native PostgreSQL Queries***

For the native PostgreSQL cases the following SQL was run using `psql` (with output piped to a file):

```

SELECT
latitude, longitude, timeval, depth, temperature, salinity
  FROM  occam_conventional_hpl_ind_comp_${N}
 WHERE latitude BETWEEN $MINLAT AND $MAXLAT AND
        longitude BETWEEN $MINLONG AND $MAXLONG AND
        timeval BETWEEN $MINDATE AND $MAXDATE AND
        depth BETWEEN $MINDEPTH AND $MAXDEPTH AND
        temperature > $MINTEMP AND salinity < $MAXSAL ;

```

As stated earlier, the following SQL command was issued in between loading the table and starting the queries:

```
vacuum full analyze occam_chips_${N}_0_TM;
```

Executing this command should allow the SQL query optimizer to select the best access path for executing the queries.

### ***DBXten Queries***

The following command was used to perform the DBXten queries:

```

pgFetchData -table occam_chips_${N}_0_tM -chip occamchip \
  -boxcolumn 'occam_cube(occamchip)' \
  -u userid -d database
  -columns timeval,depth,latitude,longitude,temperature,salinity \
  -key "timeval,$MINDATE,$MAXDATE" \
  -key "Latitude,$MINLAT,$MAXLAT" \
  -key "Longitude,$MINLONG,$MAXLONG" \
  -key "Depth,$MINDEPTH,$MAXDEPTH" \
  -key "Temperature,$MINTEMP,100.0" \
  -key "Salinity,-1.0,$MAXSAL" > file

```

The source code for the `pgFetchData` program, which is written using the DBXten C API, is provided in [Appendix B](#).

### ***Query Time Comparison***

The five tables on the next page contain the timing results for queries Q1 through Q5 in the various cases tested<sup>6</sup>.

---

<sup>6</sup> The queries are defined in Table 6 on page 8.

Q1			
Case	Native	DBXten Basic	Basic Ratio
55M_B_R	00:09.5	00:01.6	0.165
83M_B_R	01:53.9	00:01.5	0.013
110M_B_R	01:48.0	00:01.5	0.014
138M_B_R	01:54.2	00:01.6	0.014
165M_B_R	01:47.7	00:01.5	0.014
193M_B_R	02:05.0	00:01.5	0.012
220M_B_R	01:46.6	00:01.5	0.014
440M_B_R	01:46.0	00:01.8	0.017
Q2			
Case	Native	DBXten Basic	Basic Ratio
55M_B_R	00:11.3	00:01.2	0.110
83M_B_R	00:16.3	00:01.3	0.077
110M_B_R	00:22.2	00:01.2	0.055
138M_B_R	00:28.2	00:01.3	0.045
165M_B_R	00:35.0	00:01.2	0.036
193M_B_R	00:41.9	00:01.3	0.030
220M_B_R	00:48.4	00:01.3	0.027
440M_B_R	02:10.1	00:01.2	0.010
Q3			
Case	Native	DBXten Basic	Basic Ratio
55M_B_R	00:10.7	00:03.4	0.315
83M_B_R	00:16.5	00:03.9	0.236
110M_B_R	00:22.1	00:04.7	0.213
138M_B_R	00:28.0	00:05.6	0.201
165M_B_R	00:34.6	00:06.4	0.184
193M_B_R	00:41.8	00:07.4	0.176
220M_B_R	00:48.2	00:08.7	0.181
440M_B_R	02:10.4	00:15.8	0.121
Q4			
Case	Native	DBXten Basic	Basic Ratio
55M_B_R	00:27.2	00:02.8	0.102
83M_B_R	00:40.0	00:03.4	0.085
110M_B_R	00:53.2	00:04.3	0.081
138M_B_R	01:07.1	00:05.0	0.075
165M_B_R	01:22.2	00:05.9	0.072
193M_B_R	01:37.4	00:06.6	0.068
220M_B_R	01:50.3	00:07.5	0.068
440M_B_R	03:45.7	00:13.9	0.062
Q5			
Case	Native	DBXten Basic	Basic Ratio
55M_B_R	01:48.4	00:02.0	0.018
83M_B_R	01:48.3	00:01.7	0.016
110M_B_R	01:48.6	00:03.1	0.029
138M_B_R	01:48.4	00:01.8	0.016
165M_B_R	01:47.6	00:01.7	0.016
193M_B_R	01:48.4	00:01.7	0.016
220M_B_R	01:49.2	00:01.8	0.016
440M_B_R	01:48.1	00:01.8	0.017

Table 8: Query Timings

The table on the previous page indicates that DBXten performed queries much faster than when using just native PostgreSQL.

The timings in Table 8 are summarized in the following table and figure as cumulative query times for all five queries, when using each of the two options on each of the table sizes:

Million rows	Native	DBXten	Ratio
55	02:47.1	00:10.9	0.065
83	04:55.1	00:11.8	0.040
110	05:14.1	00:14.8	0.047
138	05:45.9	00:15.3	0.044
165	06:07.1	00:16.7	0.045
193	06:54.5	00:18.5	0.045
220	07:02.7	00:20.8	0.049
440	11:40.4	00:34.6	0.049

Table 9: Cumulative Query Times

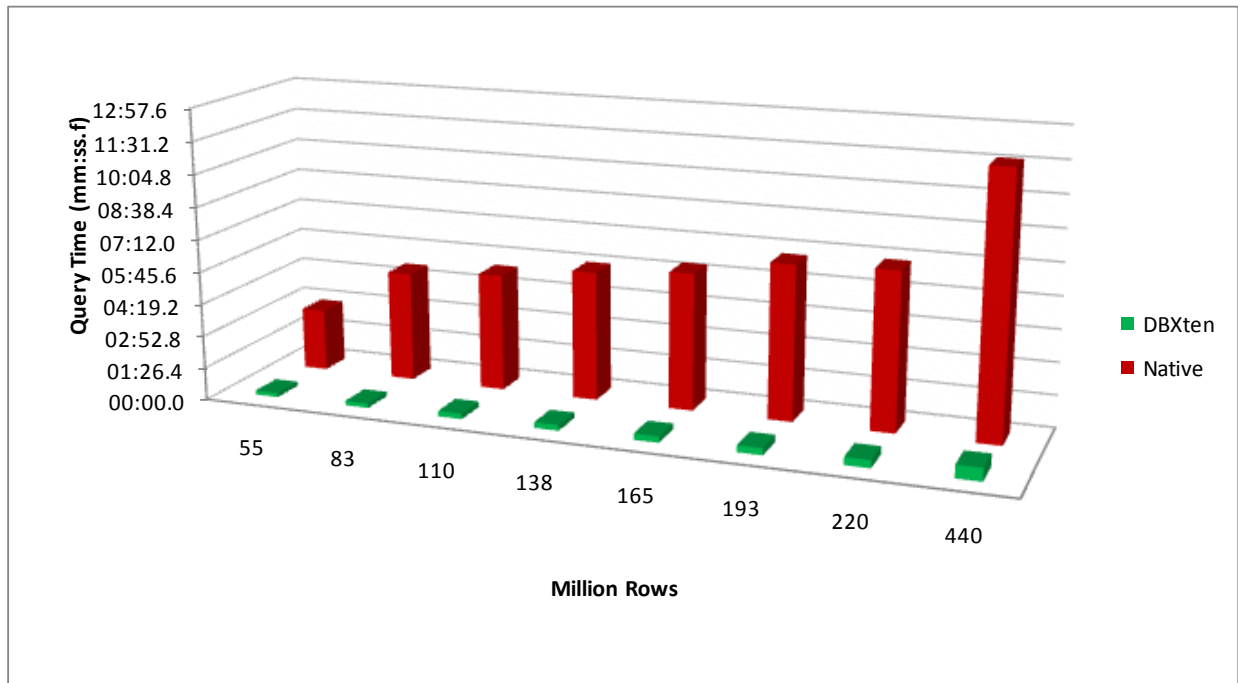


Figure 2: Cumulative Query Time as a Function of Table Size and Platform

The table and graph above dramatically illustrate that the performance advantage of DBXten over native PostgreSQL improves steadily as the data volume increases.

## Concurrent Query Testing

We conducted a final test to assess how well DBXten performs queries relative to native PostgreSQL under various load conditions. Each of queries Q1 through Q5 was run on the 220 million row table, and the numbers of concurrent users tested were N = 1, 2, 3, 5, 10, 15, 20, 25. The following table and graph present the results of this experiment:

N	Native-Avg	DBXten-Avg	Native-Max	DBXten-Max	Ratio-Avg
1	0:06:41.1	0:00:23.5	0:06:41.1	0:00:23.5	0.058
2	0:10:57.3	0:00:52.0	0:14:31.5	0:00:52.1	0.079
3	0:21:20.4	0:00:44.7	0:28:13.8	0:00:45.3	0.035
5	0:21:11.6	0:00:46.1	0:24:37.2	0:00:46.5	0.036
10	1:02:05.4	0:01:05.5	1:09:02.8	0:01:06.5	0.018
15	1:27:11.5	0:01:38.6	1:36:47.1	0:01:43.3	0.019
20	1:44:00.0	0:03:09.0	2:02:16.4	0:03:12.0	0.030
25	3:02:20.2	0:04:31.5	3:31:50.1	0:04:34.1	0.025

Table 10: Cumulative (Q1-Q5) Query Average and Maximum Times (h:mm:ss.f)

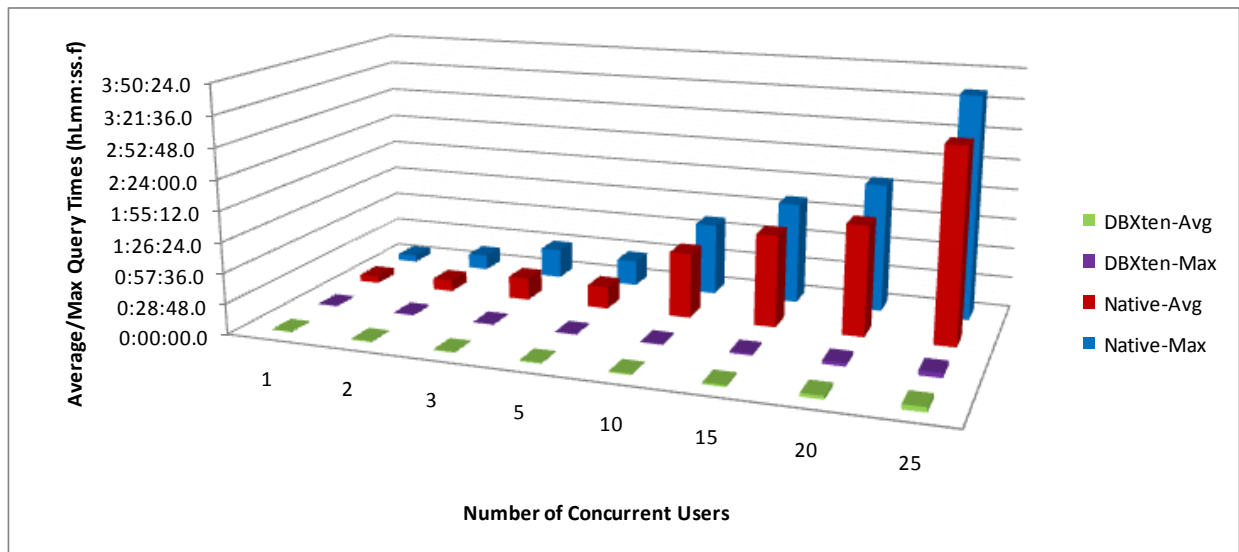


Figure 3: Cumulative (Q1-Q5) Query Average and Maximum Times (h:mm:ss.f) as a Function of Concurrency Level (N) and Platform

These concurrency trials showed that PostgreSQL *with* DBXten provided answers to queries more than an order of magnitude faster than native PostgreSQL when multiple users were involved, and that sometimes this improvement was by a factor of more than 50. Some additional remarks about these final query timings are probably in order. Note that the timings reported for N = 1 in Table 10 are slightly different from those shown for the non-concurrency tests in Table 9 corresponding to 220 million rows. This is due to the way in which the database restarts/cache clearing were performed. In the non-concurrent tests we did this in between each query. This would not have been possible during concurrent testing, so instead the restarts/cache clearing was done just between

each value of N. Also, different queries were performed by each of the N users<sup>7</sup> in order to minimize the effect of one user's query on another user's queries. However, the data read in answering a user's query was left in cache, thereby speeding up the same user's next query.

---

<sup>7</sup> The structure of the queries was the same but the specific bounds used were different, and, for any given user, the same bounds were used in the native PostgreSQL query as in the PostgreSQL with DBXten query.

## Appendix A – Abridged NetCDF File Header (ncdump)

```

dimensions:
    TIMESTEP = 1 ;
    LONGITUDE_T = 447 ;
    LATITUDE_T = 467 ;
    LONGITUDE_U = 447 ;
    LATITUDE_U = 467 ;
    DEPTH = 66 ;
    DEPTH_EDGES = 67 ;
variables:
    int TIMESTEP(TIMESTEP) ;
        TIMESTEP:long_name = "Timestep" ;
    float LONGITUDE_T(LONGITUDE_T) ;
        LONGITUDE_T:long_name = "Longitude" ;
        LONGITUDE_T:units = "degrees" ;
        LONGITUDE_T:Format = "F10.4" ;
    float LATITUDE_T(LATITUDE_T) ;
        LATITUDE_T:long_name = "Latitude" ;
        LATITUDE_T:units = "degrees" ;
        LATITUDE_T:Format = "F10.4" ;
    float DEPTH(DEPTH) ;
        DEPTH:long_name = "Depth" ;
        DEPTH:units = "cm" ;
        DEPTH:Format = "F5.2" ;
        DEPTH:positive = "down" ;
        DEPTH:edges = "DEPTH_EDGES" ;
    float POTENTIAL_TEMPERATURE__MEAN_(DEPTH, LATITUDE_T,
LONGITUDE_T) ;
        POTENTIAL_TEMPERATURE__MEAN_:long_name = "potential
temperature (mean)" ;
        POTENTIAL_TEMPERATURE__MEAN_:units = "C" ;
        POTENTIAL_TEMPERATURE__MEAN_:FillValue = 0.f ;
        POTENTIAL_TEMPERATURE__MEAN_:LEVEL = 0 ;
        POTENTIAL_TEMPERATURE__MEAN_:T_GRID = -1 ;
    float SALINITY__MEAN_(DEPTH, LATITUDE_T, LONGITUDE_T) ;
        SALINITY__MEAN_:long_name = "salinity (mean)" ;
        SALINITY__MEAN_:units = "(PSU-35)/1000" ;
        SALINITY__MEAN_:FillValue = 0.f ;
        SALINITY__MEAN_:LEVEL = 0 ;
        SALINITY__MEAN_:T_GRID = -1 ;
...

// global attributes:
        :parentfile =
"/scratch/arther3/occamp083/run401/monthly/nov2003.h5m1" ;
        :creation_command =
"/noc/users/acc/SUBVOL_UTILS/h52ncsubvol.novel -f
/scratch/arther3/occamp083/run401/monthly/nov2003.h5m1 -o /noc/om
f/scratch/lsm/javad/WORKING/4739/nov2003.h5m1subvol -include
\"POTENTIAL TEMPERATURE, SALINITY, U VELOCITY, V VELOCITY,\" -sw 726
264 -ne 1172 730 -k 1
66 -stride 1 1 1 -domain 66 4320 1735" ;
        :FMODE = 2 ;
        :FTYPE = 2 ;
        :ROTATION = 1 ;

```

data:

```
TIMESTEP = 2077464 ;
```

...

## Appendix B – Source of pgFetchData Program

```
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <DSError.h>
#include <genparseopt.h>
#include <dschip_const.h>
#include <dschip_exports.h>

static FILE *outfile;
static int verbose = 0;
static char *username = "demo",
           *hostname = "localhost",
           *dbname = "demo";

static char *outFileName = NULL;
static char *tableName = NULL;
static char *chipColumnName = NULL;
static char *boxColumn = NULL;

#define STMT_TAG "STMT_1"
#define NUM_TABLE_COLUMNS (1)

#define maxCOLUMNS (40)
static int numColumns = 0;
static char *columns[maxCOLUMNS];

#define maxRANGES (40)
int numRanges = 0;
char *rangeColumn[maxRANGES];
char *rangeLow[maxRANGES];
char *rangeHigh[maxRANGES];
double rangeHighVal[maxRANGES];
double rangeLowVal[maxRANGES];
int rangeIsKey[maxRANGES];
int numKeys = 0;

static void SetBoxColumn(char *i)
{
    boxColumn = i;
}

static void SetChipName(char *i)
{
    chipColumnName = i;
}

static void SetFileName(char *i)
{
```

```
    outFileNames[i] = i;
}

static void SetTableName(char *i)
{
    tableName = i;
}

static void SetColumn(char *i)
{
    char *item;
    for(;;) {
        item = strtok(i, ",");
        if( !item ) break;

        columns[numColumns++] = item;
        i = NULL;
    }
}

static double ParseScalar(char *strValue)
{
    if( strchr(strValue, '-') > strValue ) {
        return DSGMTStringToDouble(strValue);
    }
    else {
        double val;
        if( sscanf(strValue, "%lf", &val) != 1 ) {
            DSUserError("Bad numeric value '%s'", strValue);
        }
        return val;
    }
}

static void SetRangeCommon(char *i) {
    char buffer[255];
    char *name;
    char *lowStr;
    char *highStr;
    double lowVal, highVal;

    strncpy(buffer, i, sizeof(buffer));
    rangeColumn[numRanges] = strtok(i, ",");
    rangeLow[numRanges] = strtok(NULL, ",");
    rangeHigh[numRanges] = strtok(NULL, ",");
    if( !rangeHigh[numRanges] ) {
        DSUserError("bad range '%s'", buffer);
    }
    rangeLowVal[numRanges] = ParseScalar(rangeLow[numRanges]);
    rangeHighVal[numRanges] = ParseScalar(rangeHigh[numRanges]);

    numRanges++;
}
```

```
}

static void SetRange(char *i){
    rangeIsKey[numRanges] = 0;
    SetRangeCommon(i);
}

static void SetKey(char *i) {
    rangeIsKey[numRanges] = 1;
    numKeys++;
    SetRangeCommon(i);
}

static void ProcessSingleChip(DSChip *chip)
{
    int chipRows, numVars;
    int i, j;
    DSChipVar *vars[maxCOLUMNS];
    DSChipVar *rangeVars[maxCOLUMNS];
    char format[maxCOLUMNS][10];
    int types[maxCOLUMNS];

    chipRows = DSChipGetNumRows(chip);
    for(i = 0; i < numColumns; i++ ) {
        vars[i] = DSChipGetVarByName(chip, columns[i]);
        if( vars[i] == NULL ) {
            if( verbose ) {
                fprintf(stderr, "skipping chip without column %s",
columns[i]);
            }
            return;
        }
        types[i] = DSChipVarGetType(vars[i]);
        if( types[i] == DSVarTypeDOUBLE ) {
            int fractionLength;
            double tolerance;
            tolerance = DSChipVarGetTolerance(vars[i]);
            if( tolerance == 0 ) {
                fractionLength = 14;
            }
            else if( tolerance >= 1 ) {
                fractionLength = 0;
            }
            else {
                fractionLength = (int)ceil(-log10(tolerance));
            }
            sprintf(format[i], "%%.%df", fractionLength);
        }
    }

    for(i = 0; i < numRanges; i++ ) {
        rangeVars[i] = DSChipGetVarByName(chip, rangeColumn[i]);
    }

    for( j = 0; j < chipRows; j++ ) {
```

```

    int isInside = 1;

    for( i = 0; i < numRanges; i++ ) {
        double val = DSChipVarGetDouble(rangeVars[i], j);
        if( val < rangeLowVal[i] || val > rangeHighVal[i] ) {
            isInside = 0;
            break;
        }
    }
    if( !isInside ) continue;

    for( i = 0; i < numColumns; i++ ) {
        if( i > 0 ) {
            fprintf(outfile, ",");
        }
        switch( types[i] ) {
            case DSVarTypeDATE:
                {
                    char timeBuffer[80];
                    DSDoubleToGMTString(timeBuffer,
                                        DSChipVarGetDouble(vars[i], j), 4);
                    fprintf(outfile, "%s", timeBuffer);
                }
                break;
            case DSVarTypeDOUBLE:
                fprintf(outfile, format[i], DSChipVarGetDouble(vars[i],
j));
                break;
            case DSVarTypeSTRING:
                fprintf(outfile, "%s",
                    DSChipVarGetString(vars[i], j));
                break;
            case DSVarTypeINT:
                fprintf(outfile, "%d",
                    DSChipVarGetInt(vars[i], j));
                break;
        }
    }
    fprintf(outfile, "\n");
}

static char *BuildWhereClause()
{
    int i;
    int keyCount;
    char *buffer;
    int bufferLen;

    if( numKeys == 0 ) {
        return "";
    }
    if( boxColumn == NULL ) {
        DSUserError("keys specified but no boxcolumn argument");
    }
}

```

```

bufferLen = numRanges*256+1 + 256;
buffer = DSMalloc(bufferLen);
buffer[0] = '\0';

strncat(buffer," where dsascubestring(", bufferLen);
strncat(buffer,chipColumnName, bufferLen);
strncat(buffer," ", bufferLen);
strncat(buffer, boxColumn, bufferLen);
strncat(buffer,"'::bpchar)::cube && DSRangeToCube('", bufferLen);
keyCount = 0;
for( i = 0; i < numRanges; i++ ) {
    if( rangeIsKey[i] ) {
        if( keyCount > 0 ) {
            strncat(buffer, ",", bufferLen);
        }
        strncat(buffer, rangeColumn[i], bufferLen);
        strncat(buffer, " ", bufferLen);
        strncat(buffer, rangeLow[i], bufferLen);
        strncat(buffer, " ", bufferLen);
        strncat(buffer, rangeHigh[i], bufferLen);
        keyCount++;
    }
}
strncat(buffer,"')::cube", bufferLen);
return buffer;
}

static char * BuildQueryText() {
    int bufferLen;
    char *buffer;
    char *whereClause;

    whereClause = BuildWhereClause();
    bufferLen = strlen(whereClause) + numColumns*80 + 256;
    buffer = DSMalloc(bufferLen);
    buffer[0] = '\0';
    strncat(buffer, "select ", bufferLen);
    strncat(buffer, chipColumnName, bufferLen);
    strncat(buffer, " from ", bufferLen);
    strncat(buffer, tableName, bufferLen);
    strncat(buffer, whereClause, bufferLen);
    return buffer;
}

static PGconn *CreateConnection()
{
    PGconn *conn;
    char *connectionStringFmt = "host=%s dbname = %s user=%s";
    char connectionString[256];

    sprintf(connectionString, connectionStringFmt, hostname, dbname,
username);

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

```

```
    }
    return conn;
}

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

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

static void FetchChips( PGconn *conn)
{
    char *queryText;
    DSChip *chip;
    int chipCnt = 0;
    char *key_data[1]; /* not really used */
    int lengths[1]; /* not really used */
    int isBinary[1]; /* not really used */
    int nResults;
    int i;
    PGresult *res;

    queryText = BuildQueryText();
    if( verbose ) {
        fprintf(stderr, "query was:\n%s\n", queryText);
    }

    CheckResult(conn, PQprepare(conn, STMT_TAG, queryText, 1, NULL));

    res = PQexecPrepared(conn, STMT_TAG, 0,
        (const char *const *)key_data,
        lengths, isBinary, 1);

    CheckResult(conn, res);

    nResults = PQntuples(res);
    for( i = 0; i < nResults; i++ ) {
        void *chipData;
```

```
        int chipDataLen;
        DSChip *chip;
        chipData = PQgetvalue(res, i, 0);
        chipDataLen = PQgetlength(res, i, 0);
        chip = DSChipFromBytes(chipData, chipDataLen);
        ProcessSingleChip(chip);
        chipCnt++;
    }
    PQclear(res);

    if( verbose ) {
        fprintf(stderr, "chip count was %d\n", chipCnt);
    }
}

static void SetVerbose() {
    verbose = 1;
}

static void SetDatabaseName(char *i)
{
    dbname = i;
}

static void SetUserName( char *i)
{
    username = i;
}

static void SetHostName(char *i)
{
    hostname = i;
}

static OptLstType MyOpts[] = {
    { "o-", SetFileName },
    { "table-", SetTableName },
    { "boxcolumn-", SetBoxColumn},
    { "columns-", SetColumn },
    { "chip-", SetChipName },
    { "key-", SetKey },
    { "range-", SetRange },
    { "verbose", SetVerbose },
    { "d-", SetDatabaseName },
    { "u-", SetUserName },
    { "h-", SetHostName },
    { "\0", NULL }
};

static void printUsage(char *name) {
    fprintf(stderr, "Usage: %s arguments\n", name);
    fprintf(stderr, "where arguments include:\n");
    fprintf(stderr, "\t-o output filename # default is stdout\n");
    fprintf(stderr, "\t-table tablename\n");
    fprintf(stderr, "\t-boxcolumn boxcolumn # for rtree indexing\n");
}
```

```
    fprintf(stderr,
        "\t-columns outputColumns # comma separated, may be
repeated\n");
    fprintf(stderr, "\t-chip name_of_dschip_column\n");
    fprintf(stderr, "\t-range 'columnname,lowValue,highValue'\n");
    fprintf(stderr, "\t-key 'columnname,lowValue,highValue'\n");
    fprintf(stderr, "Note: keys are special cases of range.\n");
}

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

    DSErrSource(argv[0]);

    if( argc == 1 ) {
        printUsage(argv[0]);
    }
    argc = GenParseOpt( argc, argv, MyOpts);
    if( argc != 1 ) {
        fprintf(stderr,"unrecognized arg: %s\n", argv[1]);
        printUsage(argv[0]);
    }
    if( tableName == NULL ) {
        DSUserError("missing -table argument");
    }
    if( numColumns == 0 ) {
        DSUserError("No -columns argument");
    }
    if( chipColumnName == NULL ) {
        DSUserError("No -chip argument");
    }
    if( outFileName != NULL ) {
        outfile = fopen(outFileName, "w");
        if( !outfile ) {
            DSUserError("Unable to open file '%s' for output");
        }
    }
    else {
        outfile = stdout;
    }
    conn = CreateConnection();
    FetchChips(conn);
    if( outfile != stdout ) {
        fclose(outfile);
    }
    CloseConnection(conn);
    return 0;
}
```