

# PostgreSQL Executor: Project Assignment 2

CS186 Introduction to Database Systems, UC Berkeley

Sept 22, 2005

Due: Oct 6 (Part 2); Oct 11, 2005 (Part 1)

(Updates in red)

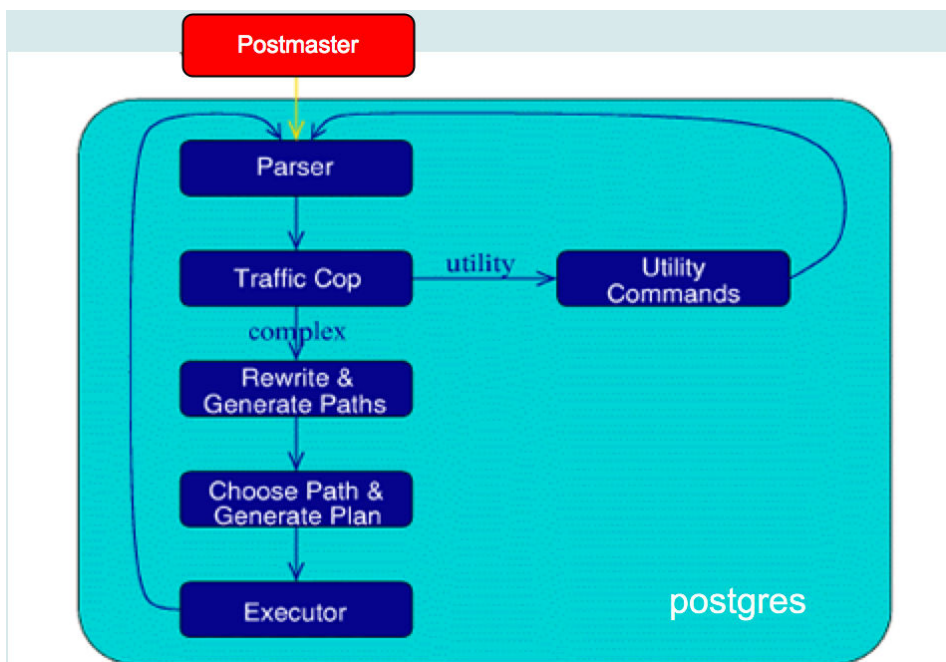
## Overview

In Project 1, you studied how to change the page replacement policy of the PostgreSQL buffer manager. In this project you will move to a higher level in the system and add functionality to the PostgreSQL executor. This project will be considerably more complex than Project 1, both in terms of the amount of coding involved and in understanding existing code. The major parts of the project are:

1. A “big picture” understanding of postgres executor
2. Adding new statistical sampling functionality to PostgreSQL
  - a. Implementing bernoulli sampling in the query executor (20%)
  - b. Implementing page sampling (also called system sampling) in the executor (50%)
3. Understanding sampling pitfalls (30%)

Part 2 and Part 3 are to be done in groups of 2.

## PostgreSQL Backend Flowchart



Before diving into the executor, it is useful to understand the "life of a query" in PostgreSQL. Recall that every client application (e.g. psql, pgaccess, or a web-server running a PHP script) first connects over a network or a local UNIX socket to the `postmaster` process, which assigns a `postgres` "backend

process" to that client. Subsequent interaction is done between the client and its associated backend process. When an SQL command string is received by the postgres backend, it is passed to the `parser` which identifies the query type, and loads the proper query-specific data structure, like `CreateStmt` or `SelectStmt`.

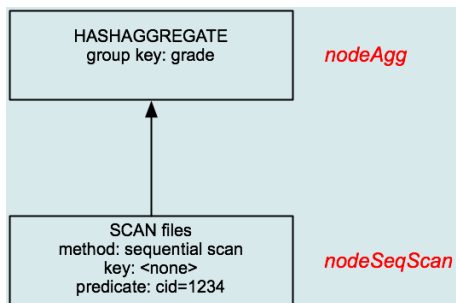
The statement is then identified as complex (`SELECT` / `INSERT` / `UPDATE` / `DELETE` -- basically the DML sublanguage of SQL) or as simple, e.g various DDL and utility commands like `CREATE USER`, `ANALYZE`, etc. The utility commands are processed by special-case functions in `backend/commands`. Complex statements are more interesting for our purposes in this assignment, since they eventually have to be passed down to the executor for processing. For more details on how a query is processed you can look at <http://www.postgresql.org/developer/ext.backend.html>.

By the time a query is passed to the Postgres executor, it has been converted into a data structure consisting of a tree of plan nodes. Each node is a single query processing operator (join, disk scan, sort, etc) that consumes relations and produces relations. As described in class, PostgreSQL uses an "iterator" model to draw tuples upward from the leaves of the tree (disk scans) up to the root. To get a row, a node "pulls" on its child node, which in turns pulls on its child nodes as needed. Note that each node must "save" its state to generate the tuples in an incremental on-demand fashion. To produce a result set, the main code of the executor pulls on the root node of the plan tree.

As a concrete example, consider the following single-table SQL query:

```
SELECT grade, COUNT(*)
FROM Enroll
WHERE cid = 1234
GROUP BY grade
```

A plan tree for the above query looks like:



Note, the text in **red** are executor nodes that you can find in `src/backend/executor/`.

## Approximate Query Answering using Samples: An Introduction

Data volumes are growing in modern database and "data warehouse" systems, and query-workloads are growing more complex. For many complex queries it can be almost impossible to support fast, interactive query-response times for users: conventional query-processing engines can take hours or even days in order to compute the exact answer for a very complex SQL query over Terabytes of disk-resident relational data. For several application scenarios, however, exact query answers are not really required, and, in fact, users would be much happier with a *fast, approximate answer* to their query

(along, perhaps, with some error guarantees for the quality of the approximation). For instance, in exploratory data-analysis sessions, a user may pose aggregation queries in order to quickly discover “interesting” regions of the database or validate a hypothesis on the underlying data – clearly, in such cases, the full precision of the exact answer is not needed, and the user would actually prefer a fast, accurate estimate of the first few digits of precision for the aggregate (e.g., the leading few digits of a total in the millions or the nearest percentile of a percentage).

One reasonable way of enabling such fast, approximate query answers is to have the executor fetch a *random sample* of a table rather than all the tuples in the table; this feature is supported in the DB2 and Oracle database systems. Typically, each of the table’s data units (also known as *sampling units*) are selected for inclusion in the query processing with some probability  $p$  that is specified in the query. The *sampling rate*,  $p$ , specifies the size of the sample to be approximately  $1/p$  times the size of the original table. (As we will see later, our “sampling units” can be either tuples or disk pages, giving rise to *tuple-level sampling* and *page-level sampling* schemes.) For instance, sampling at a rate of  $p=0.01$  (or, 1%) would reduce the data volume by a factor of 100 – clearly, running our SQL query using the sample (instead of all tuples in the table) would result in far shorter response times. Several research studies have demonstrated that concise random samples of a table  $R$  can be used to accurately estimate several aggregate queries (such as AVG, SUM, and COUNT) over the attributes of  $R$  (and, at the same time, provide approximation-error guarantees based on classical statistical-sampling theory -- a detail we will not pursue in this homework).

For this homework, we will be adding support for random-sampling operators to PostgreSQL. Following the SQL 2003 standard, we will indicate sampling within an SQL query using the TABLESAMPLE clause, which can be added to each table specification in the FROM clause of an SQL statement. The TABLESAMPLE clause includes two parameters:

1. the *sampling type specification*, indicating the specific sampling method used: in accordance with the SQL’2003 standard, we will be implementing two different sampling methods, namely *BERNOULLI tuple-level sampling* and *SYSTEM page-level sampling*
2. the *approximate sampling percentage*, indicating the percentage of the table to be sampled (i.e.,  $p*100$  for a sampling rate of  $p$ ).

Thus, a “FROM R TABLESAMPLE SYSTEM (10)” clause means that only about 10% of table  $R$ ’s disk pages will be used during query evaluation. As an example, the following SQL statement illustrates the use of the TABLESAMPLE clause -- it estimates the total individual donations made to all political parties (committees) in the year 2003-04 using a 10% random sample of the pages of the “indivdonations” relation:

```
SELECT sum(amount)  as total, committee_id
FROM indivdonations TABLESAMPLE SYSTEM(10) REPEATABLE(20)
GROUP BY committee_id
ORDER BY total;
```

The optional REPEATABLE(*seed*) clause above is used to fix the (positive INTEGER) *seed* value for the random-number generator used during the sampling process. Thus, using the same *seed* value through a REPEATABLE clause allows users to consistently generate the same result sample sets across different runs of the SQL statement. Such repeatable results are often important when testing or debugging an implementation, or in any situation where multiple runs of the same query are desired.

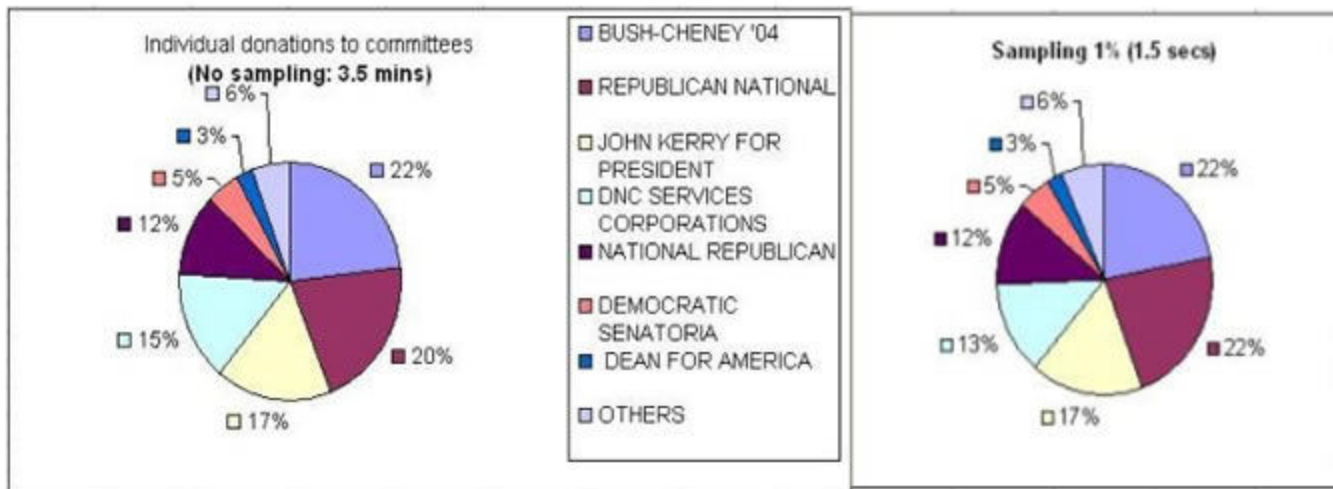
We now turn to the sampling-method specifics:

- `BERNOULLI(c)` (tuple-level) sampling is sometimes referred to as "coinflip" sampling: each tuple of the file is examined in turn, and the tuple is included in the query processing with probability  $c/100$  by using a random number generator. `BERNOULLI` sampling requires all blocks of the file to be accessed during the scan, but lowers the number of tuples handles by other operators in the query.
- `SYSTEM(c)` (page-level) sampling selects data for the sample on a page-by-page basis: before accessing a page via the buffer pool, a random number generator is invoked, and with probability  $c/100$  the page is chosen to be included in the query. In that case, all tuples on that page are included in the query. `SYSTEM` sampling is generally faster than `BERNOULLI` since it accesses only a few of the disk pages during the scan. However, it may result in samples that are *biased* (i.e., not chosen independently at random) and which are, therefore, poor representatives of the underlying data distribution. Typically, this bias arises when either the data values stored together on a page are correlated, or the number of tuples on a page varies widely. This bias can result in poor sampling-based estimates of the true aggregate value in the answer. (Various research has proposed techniques that correct for such bias in the sample, but this research is not reflected in the commercial DBMSs yet, and we will not implement it for Postgres here.)

When estimating aggregates using a `TABLESAMPLE` clause, note that, while `AVG` aggregates can be estimated directly over the sample, the results of `COUNT` and `SUM` functions will need to be scaled by the sampling rate in order to produce a correct estimate for the whole table. For instance, running a `SUM` aggregate over a 20% sample (sampling rate  $p=0.2$ ) will only sum values across about 20% of the tuples; to produce a correct estimate for the full table, the sample `SUM` must be multiplied by  $1/p=5$ . In general, any `COUNT` or `SUM` aggregate estimated over a sample must be divided by the sampling rate to scale the result to the full data set.

The time to execute the following query (similar to the previous one, except that it finds out the committee names as well) is an order of magnitude faster than running it against the entire data set. The following chart illustrates the results (converted to % donations) returned from the query run against the entire database, and then using one percent and ten percent sampling:

```
SELECT sum(a.amount)/ (1/10)  as total, b.committee_name
FROM indivdonations TABLESAMPLE SYSTEM(10) REPEATABLE(20) AS a,
committees AS b
WHERE a.committee_id = b.committee_id
GROUP BY a.committee_id, b.committee_name,
ORDER BY total;
```



As an aside, when dealing with SQL statements over multiple tables, it is conceptually easier to use the TABLESAMPLE clause over only one of the tables in the query. Specifying sampling clauses over more than one table introduces subtle and non-trivial statistical-estimation issues (especially, when join operations are involved) that we will not try to address here.

In addition to generating random samples from tables “on-the-fly” during query processing, the TABLESAMPLE clause can also be used to pre-compute and store sample tuples as tables in the database (reflecting a concise summary or demographics of the full data). Such *sample tables* can be directly used for fast approximate query processing, e.g., allowing data analysts and developers to quickly test their hypotheses or code logic without having to scan large data tables in their entirety. The following SQL statement will populate a sample table with a 5% BERNOULLI sample of the original “TRANSACTIONS” table:

```
INSERT INTO TEMP_TRANSACTIONS
SELECT * FROM TRANSACTIONS TABLESAMPLE BERNOULLI (5);
```

## Setup:

**Cleanup all data and src from your previous homeworks.** You are going to need all the space you can free up. To find how much disk space you are using, type the following on command prompt: "du -hs \$HOME". To setup for the subsequent parts, use the following:

```
1 | mkdir Hw2
2 | cd Hw2
3 | cp ~cs186/hw2/setup.py .
4 | ./setup.py
```

This should create a postgresql-8.0.3 directory for you. All files except the ones that you will be changing would be soft-linked rather than copied into your directory. We have soft-linked the object files too, in order to reduce disk space usage and compile time. However **this means that you can only**

**work on x86 instructional machines that run Solaris.** For those of you who would like to work on your own machine, you can download the tar ball of the modified source tree from [here](#). You will have to setup 2 environment variables in order to make the scripts (in test/ directory) work:

1. PROJ2\_BIN\_DIR = directory where psql/createdb/initdb are (usually in \$PREFIX/bin)
2. PROJ2\_DATA\_DIR = pgdata dir.

## Project Part 1(a): (20%)

For part 1(a), you have to implement BERNOULLI sampling in postgres. As described earlier, BERNOULLI sampling examines each row and flips a coin (generates a random number) that determines whether the row should be selected or not. To implement BERNOULLI sampling you will modify *nodeSeqscan.c* in the postgres executor (*src/backend/executor*). *nodeSeqscan* implements the sequential scan operator in postgres. We would like to modify it in such a way that it can do BERNOULLI sampling when requested, and a plain sequential scan otherwise.

## Postgres executor recap:

As described earlier, the executor processes a tree of "plan nodes". The plan tree is essentially a demand-pull pipeline of tuple processing operations. Each node, when called, will produce the next tuple in its output sequence, or **NULL** if no more tuples are available. If the node is not a primitive relation-scanning node, it will have child node(s) that it calls in turn to obtain input tuples.

Every primitive relation-scanning executor node implements the following 5 basic functions:  
(Note: Italicized *Node* stands for any node, viz Seqscan, Tidscan, Indexscan et al)

1. *ExecInitNode()*: Initialises the node
2. *ExecNodeNext()*: Fetches the next tuple from the node.
3. *ExecEndNode()*: Ends the node processing. Release any storage et al.
4. *ExecNodeRescan()*: Start the node's processing from the beginning.
5. *ExecNodeMarkPos()*: Mark current position.
6. *ExecNodeRestrPos()*: Restore scan position to some marked position.

Note that the *ExecNodeNext()* function call of a node is invoked by its parent, and returns with a single tuple. When that call returns, it must have explicitly stored enough information ("state") in the node's iterator variables such that the next time that *ExecNodeNext()* call is invoked, it can resume from wherever it left off and supply the correct next tuple.

This is a sketch of the control flow for full query processing:

```

CreateQueryDesc() [src/backend/tcop/pquery.c]

ExecutorStart() [src/backend/executor/execMain.c]
  CreateExecutorState() [src/backend/executor/execUtils.c]
    ExecInitNode() [src/backend/executor/nodeX.c]
      ExecInitNode recursively scans plan tree

ExecutorRun() [src/backend/executor/execMain.c]
  Loop until no more Tuples:
    ExecNodeNext() [src/backend/executor/nodeX.c]

```

*recursively calls the ExecNodeNext of the child nodes*

```

ExecutorEnd() [src/backend/executor/execMain.c]
ExecEndNode() [src/backend/executor/nodeX.c]
    recursively releases resources
FreeExecutorState()
    frees per-query context and child contexts

FreeQueryDesc() [src/backend/tcop/pquery.c]

```

## Implementing Part 1(a):

We want you to modify the seqscan node to do BERNOULLI sampling as well as regular sequential scan (in absence of any sampling). As described earlier, executor nodes generate tuples on demand and they store their current iterator state to facilitate this. The sequential scan node (nodeSeqscan) uses the `SeqScanState` data structure to store its state. We have added 2 fields to `SeqScanState`: `bool` `tablesample` and `struct SampleInfo`. These fields are initialized in `ExecInitSeqScan()` and you can assume that our code will set them to have valid data (i.e `sampletype = SAMPLE_BERNOULLI`, if `tablesample` is true; `sampleseed` is a positive int; and `samplepercent` is in `[1,100]`).

```

1 | struct SeqScanState {
2 |     ....
3 |     bool tablesample;
4 |     struct SampleInfo {
5 |         SampleType sampletype;
6 |         int sampleseed;
7 |         int samplepercent;
8 |     } sampleinfo;
9 | }
10| where SampleType is an enum {SAMPLE_BERNOULLI, SAMPLE_SYSTEM}

```

**All executor code resides in `src/backend/executor/` within the postgres directory. Henceforth any reference to executor source files without explicit paths would mean that it resides in the executor directory. All paths would be relative to the `postgresql-8.0.3/` directory.**

Your tasks for this part are:

1. Understand `nodeSeqscan.c`: Observe the control flow (as described above) and note how the current iterator state is stored in `SeqScanState` each time a tuple is returned.
2. Modify `SeqNext()` to sample a table if `tablesample` is true. Use `GetRandom()` to get a random integer between `[1,100]` and emit a tuple if the random number is **less than or equal to** sample percent. (NOTE: The node should perform a sequential scan if `tablesample` is false)
3. Test BERNOULLI sampling by running the sample queries (and expected responses) provided in the `test` directory (in the `postgresql-8.0.3` directory).

We have provided the `GetRandom()` logic to ensure uniformity in everyone's implementation of random "coin flipping". If you use any other method to generate the random numbers then you will not get the answers we expect.

## Project Part 1(b):(50%)

In this part you will implement a new executor node *nodeSeqBlockscan*. We have provided you with all the required glue code to ensure that a SYSTEM sample will call the appropriate methods in *nodeSeqBlockscan*. Your task is to fill in the *nodeSeqBlockScan* skeleton provided to you.

What does *nodeSeqBlockscan* do?

As the name suggests, this node scans a relation block-by-block (i.e. page-by-page), skipping some blocks without actually fetching them from the disk. Our goal is to implement SYSTEM sampling which flips a coin for every block (instead of every row). The advantage is that if we decide to not take a block, then we don't have to read it off the disk and this saves us I/O. If we decide to take a block, then we take all its *valid* tuples. (Postgres sometimes leaves invalid -- e.g. deleted -- tuples on disk pages for some time; these should be ignored.)

How to implement *nodeSeqBlockscan*?

*SeqBlockScanState* stores the iterator state for *nodeSeqBlockscan*. We have added the following to *SeqBlockScanState*:

```

1 | struct SeqBlockScanState {
2 |     .....
3 |     /* Data structs for facilitating TABLESAMPLE */
4 |     bool tablesample;
5 |     SampleInfo sampleinfo;
6 |     /* Data to maintain block sample state */
7 |     BlockSampleState blocksampler;
8 |     BlockSampleState mblocksampler; /* Mark position */
9 | } SeqBlockScanState;
```

`bool tablesample` and `SampleInfo` are as described in the previous part. They are initialized for you with valid data in the `ExecInitNode()`. `BlockSampleState` is provided to keep track of the current iterator state while scanning the relation block-by-block. We have partly filled it with the following fields:

```

1 | typedef struct
2 | {
3 |     uint32 totalblocks; /* total number of blocks in the relation */
4 |
5 |     /* Fill in more as appropriate */
6 |
7 | } BlockSampleState;
```

`totalblocks` is initialized in `ExecInitNode()` with the total number of blocks in the relation.

Your task for this part is:

1. Fill in `SeqBlockNext()`: This function has to return a tuple-slot each time it is called or NULL-slot if there are no more tuples. You need to scan the relation **block-by-block** and `GetRandom()` number for each block. If the random number is **less or equal to** the `samplepercent` then you

select this block and return all valid tuples on it one at a time. To facilitate this you would have to store the current state (hint: block number, tuple offset et al) in `BlockSampleState`.

2. Fill in `BlockSampleState`: Decide what variables you need to create in `BlockSampleState` in order to save the current iterator state of `SeqBlockNext()`. Fill this in. Remember to initialize those variables in `InitBlockSampler()`. `BlockSampleState` is defined in `src/include/nodes/execnodes.h`.
3. Test your code: Use the SYSTEM sample queries in `test` directory (in `postgresql-8.0.3`) to see if your code works as expected.

We provide you with a few utility functions that might be helpful to you (these functions bundle the buffer manager functions exposing a relatively simpler interface to you):

1. *Buffer* `GetBuffer(Relation rel, int currblock)`: Fetches the particular block of the relation in memory and pins it. Returns the *Buffer* number. Blocks are numbered from `[0, totalblocks)` in postgres.
2. *int* `GetMaxTuples(Buffer)`: Returns the maximum number of tuples in the page. Note: We say "maximum number" as the page might have deleted tuples too.
3. *HeapTuple* `heap_fetch_tuple(Relation rel, Buffer buf, uint32 blockNum, uint32 tupleNum)`: Returns `HeapTuple` if a valid tuple is found at the `tupleNum` in the given `blockNum`; Else returns `NULL`. `tupleNum` goes from `[1, MaxNumberOfTuples]` (where `MaxNumberOfTuples = GetMaxTuples(Buffer)`).
4. *void* `Release(Buffer buf)`: Release the buffer and. decrements the `refcount`. Each time you call `GetBuffer()`, it pins the page for you. Hence *you must remember to unpin it as soon as possible* to prevent the buffer manager from running out of free frames (a "buffer leak").
5. `SeqScanState/SeqBlockScanState`: These store the state information for respective nodes.
6. *TupleTableSlot* `ExecStoreTuple(HeapTuple tuple, TupleTableSlot slot, Buffer buf, bool pfree)`: Stores that tuple that you found (using `heap_fetch_tuple`) in the tuple slot. `buf` should be the one that contains the tuple. `pfree` indicates if the tuple should be free'ed (true in this case as we do a `heap_copytuple` in `heap_fetch_tuple`). `slot` is the one in `node->ss_ScanTupleSlot`. Note: You should invoke `ExecStoreTuple` only if `tuple` is not `NULL`.
7. *void* `ExecClearTuple(TupleTableSlot slot)`: Clears the tuple slot. Should be called before `ExecStoreTuple`.

## Compiling and debugging:

We recommend you to compile your code as follows (we have enabled the following by default, `--enable-debug`, `--enable-cassert`, `--prefix=/home/tmp/$USER/pgsql`):

[view plain](#) | [print](#) | [copy to clipboard](#) | ?

```

1 | ./configure
2 | gmake
3 | gmake install
4 | For quick recompile and install, use:
5 | gmake -C src/backend/ install-bin
6 | To debug:
7 | gdb postgres
8 | (gdb) run testdb
9 | postgresql>select count(*) from test TABLESAMPLE BERNOULLI(10);
10 |
```

## Testing:

We have provided you some scripts in `testdirectory` (under `postgresql-8.0.3/`). Using these scripts you can check the correctness of your code and play around with different kinds of sampling queries (measure time/io-performance). If you are using instructional machines, use the following commands to setup the `pgdata` directory for this project (since the databases that we will be creating would be much bigger and will exceed your quota if you use your home directory for `pgdata`):

[view plain](#) | [print](#) | [copy to clipboard](#) | ?

```

1 | mkhometmpdir
2 | (This will create a tmp dir for you /home/tmp/class-account)
3 | mkdir /home/tmp/$USER/pgdata
4 | (Use /home/tmp/$USER/pgdata, as your pgdata directory for this
5 | assignment)
6 |

```

**If you are using the instructional machines, then use `DATADIR` as `/home/tmp/cs186-??/pgdata` (where `??`=your class account). Do not use your home directory for `pgdata` as it will run out of disk space.**

We provide you the following scripts:

1. `dolnitDB.sh`: You need to run this only once. (Unless you delete the `pgdata` directory). This script initializes the database and adds a modified `postgres.conf`. Creates a database `testdb`.
2. `createSmallTable.sh`: Creates and populates a small table (`test`). Useful when debugging your application.
3. `testOnSmallTable.sh`: Runs a few `TABLESAMPLE` queries on the tables created by (2). diff's the output with the responses we expect.
4. `createBigTables.sh`: Creates a bunch of big tables. The tables contain donations made to various political parties and candidates in 2003-04. This is a modified subset of the data available at [http://www.fec.gov/finance/disclosure/ftpdet.shtml#2003\\_2004](http://www.fec.gov/finance/disclosure/ftpdet.shtml#2003_2004). View the list of tables using `\d` and the schema of each table using `\d tablename`.
5. `testOnBigTables.sh`: Runs a few `TABLESAMPLE` queries on the donations database created in (4). We recommend you to run these only when you have cleared (3).

We encourage you to try more and different kinds of `TABLESAMPLE` queries than what we have provided.

## Measuring IO:

You can find the IO incurred by your query using:

[view plain](#) | [print](#) | [copy to clipboard](#) | ?

```

1 | select * from pg_statio_all_tables where relname='tablename'
2 |

```

Note, this table keeps track of **total** IO incurred on the `tablename` **since the database was last started**.

## Measuring Time:

\timing on psql prompt toggles the timing option.

[view plain](#) | [print](#) | [copy to clipboard](#) | ?

```
1 | pgsql=# \timing
2 | pgsql=# select * from whatever-your-query
```

## Project part 2: (30%)

While the TABLESAMPLE feature can be very useful for quick analysis, SQL development, and testing, you should remember that these results reflect sampling of values and not the entire table! A bad choice of these random tuples (or blocks, in the case of SYSTEM sampling) might give us very distorted results. In this part of the project your task is to explore a few scenarios where sampling fails and try to investigate the reasons behind it.

You will be examining 3 scenarios, 2(a), (b) and (c). You will be using a trimmed and slightly modified version of the US government's political party donations database from [http://www.fec.gov/finance/disclosure/ftpdet.shtml#2003\\_2004](http://www.fec.gov/finance/disclosure/ftpdet.shtml#2003_2004). Our scripts will set up this database for you. Its schema is described below:

```
1 | CREATE TABLE committees (committee_id varchar(9) PRIMARY KEY,
2 |   committee_name varchar(20), treasurer varchar(18),
3 |   city varchar(18), state varchar(2), party varchar(3),
4 |   organization varchar(30));
5 |   -- list of all committees with their details
6 |
7 | CREATE TABLE candidates (candidate_id varchar(9) PRIMARY KEY,
8 |   candidate_name varchar(20), designated_party varchar(3),
9 |   city varchar(18), state varchar(2) cluster on (state));
10 |   --lists of all candidates with their details
11 |
12 | CREATE TABLE CommitteeDonations (committee_id varchar(9),
13 |   donation_date Date, amount integer, candidate_id varchar(9));
14 |   -- donations made BY committees TO candidates
15 |
16 | CREATE TABLE IndividDonations (indiv_name varchar(34),
17 |   indiv_occupation varchar(35), committee_id varchar(9),
18 |   date Date, amount integer);
19 |   -- donations made TO committees by individuals
20 |
```

## How to run the queries:

view plain | print | copy to clipboard | ?

```

1 ~cs186/hw2/pgsql/bin/psql -U student -p 9991 test
2 test=# \d -- list of tables
3 test=# \d tablename --tablename schema
4 test=# select * query
5

```

In this part of the assignment, you will connect to a postmaster that we have installed on the following SOLARIS x86 instructional machines:

1. po
2. rhombus

(more to be added soon).

This postmaster runs a version of postgres that includes a correct implementation of sampling. The postmaster on each instructional machine will only allow 20 simultaneous connections. Try using a different instructional machine if you are unable to connect on one. We recommend you to do this part early and avoid heavy system loads at the last minute. **We will not give any discounts for students who are unable to complete this part of the homework due to the postmaster being inaccessible a few hours before the deadline.** A periodic script monitors the postmaster health and restarts it in case it crashes.

### Part 2(a) "John Kerry for President Inc " vs "Bush-Cheney '04 Inc." (10%)

These are the two big committees that received a substantial amount of donations from individuals. If you observe the above schema carefully then you will see that individuals donate to committees while committees donate to candidates. To avoid forcing you to look at join queries in this homework, we have already looked up the ids of the above committees for you:

- John Kerry for President Inc: Committee ID = C00383653
- Bush-Cheney '04 Inc. : Committee ID = C00386987

For this subpart your task is to run the following 4 queries and fill in the results table below. For the queries involving sampling, we ask you to run them 10 times with 10 different seeds (1 through 10), and note the result you get each time. The table is provided for your reference and you do not have to turn it in.

view plain | print | copy to clipboard | ?

```

Q1.no_sample: SELECT avg(amount) from indivdonations
where committee_id='C00386987';

```

```

Q1: SELECT avg(amount) from indivdonations TABLESAMPLE SYSTEM(10)
REPEATABLE(r) where committee_id='C00386987';

```

```

Q2.no_sample: SELECT avg(amount) from indivdonations
where committee_id='C00383653';

```

```

Q2: SELECT avg(amount) from indivdonations TABLESAMPLE SYSTEM(10)
REPEATABLE(r) where committee_id='C00383653';

```

Query	Avg(amount)
Q1.no_sample	
Q2.no_sample	

Query	Seed (r) ->	1	2	3	4	5	6	7	8
Q1									
Q2									

For the above table, the last column is the sample standard deviation where mean = 'Qi.no\_sample' ?  
 [For details on calculating the sample standard deviation, refer to <http://mathworld.wolfram.com/StandardDeviation.html>]. PostgreSQL provides the STDDEV aggregate function to compute a standard deviation as well, if you would like to load your results into a database table.

You need to turn in answers to the following questions:

- How does sampling perform for Q1 and Q2? (versus no sampling) (answer in 1 line)
- How does the sample standard deviation for Q1 and Q2 correlate with sampling accuracy? (answer in 1 line)
- Why do you think that sampling did not do a good job for Q1 / Q2?  
 [Hint: You might want to start your investigation with the following queries:  
*select max(amount) from indivdonations where committee\_id='...'*  
*select min(amount) from indivdonations where committee\_id='...'*  
*select stddev(amount) from indivdonations where committee\_id='...'*]

Part 2(b) Ever heard of "The Media Fund Inc" Committee? (10%)

Neither did we! But its committee ID is C30000053.

For this part you will run the following queries and build a table. Finally you will answer the questions that follow.

[view plain](#) | [print](#) | [copy to clipboard](#) | ?

```
Q3.no_sample: SELECT avg(amount) from indivdonations
where committee_id='C30000053';
```

```
Q3.sys_sample: SELECT avg(amount) from indivdonations TABLESAMPLE
SYSTEM(10) REPEATABLE(r) where committee_id='C30000053';
```

```
Q3.bern_sample: SELECT avg(amount) from indivdonations TABLESAMPLE
BERNOULLI(10) REPEATABLE(r) where committee_id='C30000053';
```

Query	Avg(amount)
Q3.no_sample	

Query	Seed (r) ->	1	2	3	4	5	6	7	8

Q3.SYS_SAMPLE									
Q3.BERN_SAMPLE									

Again, the last column above stands for sample standard deviation where mean = Q3.no\_sample.

Now answer the following questions:

1. How did SYSTEM and BERNOULLI sampling perform? (1 line)
2. If you say that they did not perform well, briefly outline why did they fail?  
 [Hint: You might want to start with  
*select count(\*) from indivdonations where committee\_id='C30000053' vs  
 select count(\*) from indivdonations]*

Part 2(c) Number of candidates from California? (10%)

From the candidates table you can find out that the number of candidates contesting from california were the largest. For this part we want you to run the following queries:

[view plain](#) | [print](#) | [copy to clipboard](#) | ?

```
Q4.no_sample: select count(*) from candidates where state='CA';
```

```
Q4.sys_sample: select count(*) * 100/p from candidates tablesample system(p) where state='CA'
```

```
Q4.bern_sample: select count(*) * 100/p from candidates tablesample bernoulli(p) where state='CA'
```

Query	count(*)
Q4.no_sample	

Query	Percent (p) ->	1	5	10	20	25	40	50	75
Q4.SYS_SAMPLE									
Q4.BERN_SAMPLE									

**Note, we are varying the sampling percent in the above query and not the random seed.**

Now answer the following questions:

1. At what sampling percent (from the ones in the table) does SYSTEM sampling give an answer within **+15%** error range? (1 line)
2. At what sampling percent (from the ones in the table) does BERNOULLI sampling give an answer with **+15%** error range? (1 line)
3. Why do you think BERNOULLI sampling performs better than SYSTEM sampling?  
 [Hint: revisit the DDL above for the candidates table (\d candidates). You should not have to run any queries.]

## How to submit:

### Submitting part 1 **Due Oct 11, 2005**

Create a directory, hw2p1:

Copy the following files to this directory:

1. nodeSeqscan.c (from src/backend/executor/)
2. nodeSeqBlockscan.c
3. execnodes.h (from src/include/nodes)

### Submitting part 2 **Due Oct 6, 2005**

Create a directory hw2p2

Create answers.txt and put your responses for part 2(a)[1-3], 2(b)[1-2], 2(c)[1-3] in this file. Limit the total file size to 300 words. **We will automatically truncate your responses after 300 words.** To know the number of words in your file run: "wc -w filename" on the command prompt. Be terse and to the point. You need not explain in gory details but highlight the main factors affecting sampling in various scenarios.