Introduction to Hacking PostgreSQL

Neil Conway
neilc@samurai.com

May 21, 2007

Outline

1 Prerequisites
- Why Should You Hack On PostgreSQL?
- What Skills Will You Need?
- What Tools Should You Use?

2 The Architecture of PostgreSQL
- System Architecture
- Components of the Backend

3 Common Code Conventions
- Memory Management
- Error Handling

4 Community Processes

5 Sample Patch

6 Conclusion

Outline

1 Prerequisites
- Why Should You Hack On PostgreSQL?
- What Skills Will You Need?
- What Tools Should You Use?

2 The Architecture of PostgreSQL
- System Architecture
- Components of the Backend

3 Common Code Conventions
- Memory Management
- Error Handling

4 Community Processes

5 Sample Patch

6 Conclusion

Why Hack On PostgreSQL?

Possible Reasons

- Databases are fun!

Why Hack On PostgreSQL?

Possible Reasons

- Databases are fun!
- Contribute to the community
  - We need more reviewers

Why Hack On PostgreSQL?

Possible Reasons

- Databases are fun!
- Contribute to the community
    - We need more reviewers
- Understand how PostgreSQL works more deeply

Why Hack On PostgreSQL?

Possible Reasons

- Databases are fun!
- Contribute to the community
  - We need more reviewers
- Understand how PostgreSQL works more deeply
- Become a better programmer
  - The PostgreSQL source is a good example to learn from

Why Hack On PostgreSQL?

Possible Reasons

- Databases are fun!
- Contribute to the community
  - We need more reviewers
- Understand how PostgreSQL works more deeply
- Become a better programmer
  - The PostgreSQL source is a good example to learn from
- Commercial opportunities

Skills

Essential

- Some knowledge of C
  - Fortunately, C is easy
- Some familiarity with Unix and basic Unix programming
  - Postgres development on Win32 is increasingly feasible

Helpful, but not essential

- Unix systems programming
- DBMS internals
- Autotools-foo
- Performance analysis

. . . depending on what you want to hack on

Development Tools

The Basics

$CC, Bison, Flex, CVS, autotools

- Configure flags: `enable-depend`, `enable-debug`, `enable-cassert`
- Consider `CFLAGS=-O0` for easier debugging (and faster builds)
  - With GCC, this suppresses some important warnings

Development Tools

The Basics

$CC, Bison, Flex, CVS, autotools

- Configure flags: `enable-depend`, `enable-debug`, `enable-cassert`
- Consider `CFLAGS=-O0` for easier debugging (and faster builds)
  - With GCC, this suppresses some important warnings

Indexing The Source

- A tool like `tags`, `cscope` or `glimpse` is essential when navigating *any* large code base
  - "What is the definition of this function/type?"
  - "What are all the call-sites of this function?"
  - `src/tools/make_[ce]tags`

Other Tools

- A debugger is often necessary: most developers use `gdb`
  - Or a front-end like `ddd`
  - Even MSVC?
- `ccache` and `distcc` are useful, especially on slower machines
- `valgrind` is useful for debugging memory errors and memory leaks in client apps
  - Not as useful for finding backend memory leaks

Other Tools

- A debugger is often necessary: most developers use `gdb`
  - Or a front-end like `ddd`
  - Even MSVC?
- `ccache` and `distcc` are useful, especially on slower machines
- `valgrind` is useful for debugging memory errors and memory leaks in client apps
  - Not as useful for finding backend memory leaks

Profiling

- `gprof` is the traditional choice; various bugs and limitations
  - Use `--enable-profiling` to reduce the pain
- `callgrind` works well, nice UI (`kcachegrind`)
- `oprofile` is good at system-level performance analysis
- DTrace

SGML Documentation

Understatement

The DocBook toolchain is less than perfect

SGML Documentation

Understatement

The DocBook toolchain is less than perfect

Authoring SGML

- I don't know of a good SGML editor, other than Emacs
  - Writing DocBook markup by hand is labour-intensive but not hard: copy conventions of nearby markup
- `make check` does a quick syntax check
- `make draft` is useful for previewing changes

Patch Management

- Most development is done by mailing around patches
  - `echo "diff -c -N -p" >> ~/.cvsrc`
  - `cvs diff > ~/my_patch-vN.patch`
- `interdiff` is a useful tool: "exactly what did I change between v5 and v6?"
- Remote `cvs` is slow: setup a local mirror of the CVS repository
  - `cvsup`, `csup`, `rsync`, `svnsync` (soon!)
- To include newly-added files in a CVS diff, either use a local CVS mirror or `cvsutils`
- For larger projects: akpm's Quilt, or a distributed VCS
  - For example, Postgres-R uses Monotone

Text Editor

- If you're not using a good programmer's text editor, start
- Teach your editor to obey the Postgres coding conventions:
  - Hard tabs, with a tab width of 4 spaces
  - Similar to Allman/BSD style; just copy the surrounding code
- Using the Postgres coding conventions makes it more likely that your patch will be promptly reviewed and applied

Useful Texts

- SQL-92, SQL:1999, SQL:2003, and SQL:200n
  - `http://www.wiscorp.com/SQLStandards.html` ("draft")
  - There are some books and presentations that are more human-readable
  - There's a samizdat plaintext version of SQL-92
- SQL references for Oracle, DB2, ...
- A textbook on the design of database management systems
  - I personally like *Database Management Systems* by Ramakrishnan and Gehrke
- Books on the toolchain (C, Yacc, autotools, ...) and operating system kernels

Outline

1 Prerequisites
- Why Should You Hack On PostgreSQL?
- What Skills Will You Need?
- What Tools Should You Use?

2 The Architecture of PostgreSQL
- System Architecture
- Components of the Backend

3 Common Code Conventions
- Memory Management
- Error Handling

4 Community Processes

5 Sample Patch

6 Conclusion

The Postmaster

Lifecycle

1. Initialize essential subsystems; perform XLOG recovery to restore the database to a consistent state

The Postmaster

Lifecycle

1. Initialize essential subsystems; perform XLOG recovery to restore the database to a consistent state
2. Attach to shared memory segment (SysV IPC), initialize shared data structures

The Postmaster

Lifecycle

1. Initialize essential subsystems; perform XLOG recovery to restore the database to a consistent state
2. Attach to shared memory segment (SysV IPC), initialize shared data structures
3. Fork off daemon processes: autovacuum launcher, stats daemon, bgwriter, syslogger

The Postmaster

Lifecycle

1. Initialize essential subsystems; perform XLOG recovery to restore the database to a consistent state
2. Attach to shared memory segment (SysV IPC), initialize shared data structures
3. Fork off daemon processes: autovacuum launcher, stats daemon, bgwriter, syslogger
4. Bind to TCP socket, listen for incoming connections
   - For each new connection, spawn a backend
   - Periodically check for child death, launch replacements or perform recovery

Daemon Processes

Types of Processes

autovacuum launcher: Periodically start autovacuum workers
bgwriter: Flush dirty buffers to disk, perform periodic checkpoints
stats collector: Accepts run-time stats from backends via UDP
syslogger: Collect log output from other processes, write to file(s)
normal backend: Handles a single client session

Daemon Processes

Types of Processes

autovacuum launcher: Periodically start autovacuum workers
bgwriter: Flush dirty buffers to disk, perform periodic checkpoints
stats collector: Accepts run-time stats from backends via UDP
syslogger: Collect log output from other processes, write to file(s)
normal backend: Handles a single client session

Inter-Process Communication

- Most shared data is communicated via a shared memory segment
- Signals, semaphores, and pipes also used as appropriate
  - Stats collector uses UDP on the loopback interface
- Subprocesses inherit the state of the postmaster after `fork()`

Consequences

Advantages

- Address space protection: most of the time, *not possible* for misbehaving processes to crash the entire DBMS
- IPC and modifications to shared data are explicit: all state is process-private by default

Consequences

Advantages

- Address space protection: most of the time, *not possible* for misbehaving processes to crash the entire DBMS
- IPC and modifications to shared data are explicit: all state is process-private by default

Disadvantages

- Shared memory segment is statically-sized at startup
  - Managing arbitrarily-sized shared data is problematic
- Some shared operations can be awkward: e.g. using multiple processors to evaluate a single query

Backend Lifecycle

Backend Lifecycle

1. Postmaster accepts a new connection, forks a new backend, then closes its copy of the TCP socket
   - All communication occurs between the backend and the client

Backend Lifecycle

Backend Lifecycle

1. Postmaster accepts a new connection, forks a new backend, then closes its copy of the TCP socket
   - All communication occurs between the backend and the client
2. Backend enters the “frontend/backend” protocol:
   1. Authenticate the client
   2. “Simple query protocol”: accept a query, evaluate it, return result set
   3. When the client disconnects, the backend exits

Stages In Query Processing

Major Components

1. The **parser** - lex & parse the query string
2. The **rewriter** - apply rewrite rules
3. The **optimizer** - determine an efficient query plan
4. The **executor** - execute a query plan
5. The **utility processor** - process DDL like `CREATE TABLE`

The Parser

- Lex and parse the query string submitted by the user
- Lexing: divide the input string into a sequence of *tokens*
  - Postgres uses GNU Flex
- Parsing: construct an abstract syntax tree (AST) from sequence of tokens
  - Postgres uses GNU Bison
  - The elements of the AST are known as parse nodes

The Parser

- Lex and parse the query string submitted by the user
- Lexing: divide the input string into a sequence of *tokens*
  - Postgres uses GNU Flex
- Parsing: construct an abstract syntax tree (AST) from sequence of tokens
  - Postgres uses GNU Bison
  - The elements of the AST are known as parse nodes
- Produces a "raw parsetree": a linked list of parse nodes
  - Parse nodes are defined in `include/nodes/parsenodes.h`
- Typically a simple mapping between grammar productions and parse node structure

Semantic Analysis

- In the parser itself, only syntactic analysis is done; basic semantic checks are done in a subsequent "analysis phase"
  - `parser/analyze.c` and related code under `parser/`
- Resolve column references, considering schema path and query context
  - ```
    SELECT a, b, c FROM t1, t2, x.t3
      WHERE x IN (SELECT t1 FROM b)
    ```
- Verify that referenced schemas, tables and columns exist
- Check that the types used in expressions are consistent
- In general, check for errors that are impossible or difficult to detect in the parser itself
    ```

Rewriter, Planner

- The analysis phase produces a `Query`, which is the query's parse tree (Abstract Syntax Tree) with additional annotations
- The rewriter applies rewrite rules, including view definitions. Input is a `Query`, output is zero or more `Query`s
- The planner takes a `Query` and produces a `Plan`, which encodes how the query should be executed
  - A query plan is a tree of `Plan` nodes, each describing a physical operation
  - Only needed for "optimizable" statements (`INSERT`, `DELETE`, `SELECT`, `UPDATE`)

Executor

- Each node in the plan tree describes a physical operation
  - Scan a relation, perform an index scan, join two relations, perform a sort, apply a predicate, perform projection, ...

Executor

- Each node in the plan tree describes a physical operation
  - Scan a relation, perform an index scan, join two relations, perform a sort, apply a predicate, perform projection, ...
- The planner arranges the operations into a plan tree that describes the data flow between operations
- Tuples flow from the leaves of the tree to the root
  - Leaf nodes are *scans*: no input, produce a stream of tuples
  - Joins are binary operators: accept two inputs (child nodes), produce a single output
  - The root of the tree produces the query's result set
- Therefore, the executor is "trivial": simply ask the root plan node to repeatedly produce result tuples

Query Optimization

- SQL is (ostensibly) a declarative query language
  - The query specifies the properties the result set must satisfy, not the procedure the DBMS must follow to produce the result set
- For a typical SQL query, there are many equivalent query plans

Query Optimization

- SQL is (ostensibly) a declarative query language
  - The query specifies the properties the result set must satisfy, not the procedure the DBMS must follow to produce the result set
- For a typical SQL query, there are many equivalent query plans

scan types:	Seq scan, index scan, bitmap index scan
join order:	Inner joins are commutative: reordered freely
join types:	Sort-merge join, hash join, nested loops
aggregation:	Hashed aggregation, aggregation by sorting
predicates:	Predicate push down, evaluation order
rewrites:	Subqueries and set operations → joins, outer joins → inner joins, function inlining, ...

Tasks Of The Query Optimizer

Basic Optimizer Task

Of the many ways in which we could evaluate a query, which would be the cheapest to execute?

Tasks Of The Query Optimizer

Basic Optimizer Task

Of the many ways in which we could evaluate a query, which would be the cheapest to execute?

Two Distinct Subproblems

1. Enumerate all the possible plans for a given query
2. Estimate the cost of a given query plan

In practice, too slow → do both steps at the same time

Stages in Query Optimization

The System R Algorithm

1. Rewrite the query to make it more amenable to optimization: pull up subqueries, rewrite `IN` clauses, simplify constant expressions, reduce outer joins, . . .

Stages in Query Optimization

The System R Algorithm

1. Rewrite the query to make it more amenable to optimization: pull up subqueries, rewrite `IN` clauses, simplify constant expressions, reduce outer joins, . . .
2. Determine the *interesting* ways to access each base relation
   - Remember the cheapest estimated access path, plus the cheapest path for each distinct sort order

Stages in Query Optimization

The System R Algorithm

1. Rewrite the query to make it more amenable to optimization: pull up subqueries, rewrite `IN` clauses, simplify constant expressions, reduce outer joins, . . .
2. Determine the *interesting* ways to access each base relation
   - Remember the cheapest estimated access path, plus the cheapest path for each distinct sort order
3. Determine the *interesting* ways to join each pair of relations

Stages in Query Optimization

The System R Algorithm

1. Rewrite the query to make it more amenable to optimization: pull up subqueries, rewrite `IN` clauses, simplify constant expressions, reduce outer joins, ...
2. Determine the *interesting* ways to access each base relation
   - Remember the cheapest estimated access path, plus the cheapest path for each distinct sort order
3. Determine the *interesting* ways to join each pair of relations
4. ...

Outline

1. Prerequisites
   - Why Should You Hack On PostgreSQL?
   - What Skills Will You Need?
   - What Tools Should You Use?
2. The Architecture of PostgreSQL
   - System Architecture
   - Components of the Backend
3. Common Code Conventions
   - Memory Management
   - Error Handling
4. Community Processes
5. Sample Patch
6. Conclusion

The Postgres Object System: Nodes

- Postgres uses a simple object system with support for single inheritance. The root of the class hierarchy is `Node`:

```
typedef struct
{
    NodeTag type;
} Node;
```

```
typedef struct
{
    NodeTag    type;
    int        a_field;
} Parent;
```

```
typedef struct
{
    Parent parent;
    int    b_field;
} Child;
```

- This relies on a C trick: you can treat a `Child *` like a `Parent *` since their initial fields are the same
- Unfortunately, this can require a lot of ugly casting
- The first field of *any* `Node` is a `NodeTag`, which can be used to determine a `Node`'s specific type at runtime

Nodes, Cont.

Basic Node Utility Functions

- Create a new `Node`: `makeNode()`

Nodes, Cont.

Basic Node Utility Functions

- Create a new `Node`: `makeNode()`
- Run-time type testing via the `IsA()` macro

Nodes, Cont.

Basic Node Utility Functions

- Create a new `Node`: `makeNode()`
- Run-time type testing via the `IsA()` macro
- Test if two nodes are equal: `equal()`

Nodes, Cont.

Basic Node Utility Functions

- Create a new Node: `makeNode()`
- Run-time type testing via the `IsA()` macro
- Test if two nodes are equal: `equal()`
- Deep-copy a node: `copyObject()`

Nodes, Cont.

Basic Node Utility Functions

- Create a new `Node`: `makeNode()`
- Run-time type testing via the `IsA()` macro
- Test if two nodes are equal: `equal()`
- Deep-copy a node: `copyObject()`
- Serialise a node to text: `nodeToString()`

Nodes, Cont.

Basic Node Utility Functions

- Create a new `Node`: `makeNode()`
- Run-time type testing via the `IsA()` macro
- Test if two nodes are equal: `equal()`
- Deep-copy a node: `copyObject()`
- Serialise a node to text: `nodeToString()`
- Deserialise a node from text: `stringToNode()`

Nodes: Hints

- *When you modify a node or add a new node, remember to update*
  - `nodes/equalfuncs.c`
  - `nodes/copyfuncs.c`
- You may have to update `nodes/outfuncs.c` and `nodes/readfuncs.c` if your `Node` is to be serialised/deserialised
- Grep for references to the node's type to make sure you don't forget to update anything
  - When adding a new node, look at how similar nodes are treated

Memory Management

- Postgres uses hierarchical, region-based memory management, and it absolutely rocks
  - `backend/util/mmgr`
  - Similar concept to Tridge's `talloc()`, "arenas", . . .
- All memory allocations are made in a memory context
- Default context of allocation: `CurrentMemoryContext`
- `palloc()` allocates in CMC
- `MemoryContextAlloc()` allocates in a given context

Memory Management, cont.

- Allocations can be freed individually via `pfree()`
- When a memory context is reset or deleted, all allocations in the context are released
  - Resetting contexts is both faster and less error-prone than releasing individual allocations
- Contexts are arranged in a tree; deleting/resetting a context deletes/resets its child contexts

Memory Management Conventions

- You should sometimes `pfree()` your allocations
  - If the context of allocation is known to be short-lived, don't bother with `pfree()`
  - If the code might be invoked in an arbitrary memory context (e.g. utility functions), you should `pfree()`
  - You can't `pfree()` an arbitrary `Node` (no "deep free")
- The exact rules are a bit hazy :-(

Memory Leaks

- Be aware of the memory allocation assumptions made by functions you call
- Memory leaks, *per se*, are rare in the backend
  - All memory is released eventually
  - A “leak” occurs when memory is allocated in a too-long-lived memory context: e.g. allocating some per-tuple resource in a per-txn context
  - `MemoryContextStats()` useful for locating the guilty context
- (Almost) never use `malloc()` in the backend

Error Handling

- Most errors reported by `ereport()` or `elog()`
  - `ereport()` is for user-visible errors, and allows more fields to be specified (SQLSTATE, detail, hint, etc.)
- Implemented via `longjmp`; conceptually similar to exceptions in other languages
  - `elog(ERROR)` walks back up the stack to the closest error handling block; that block can either handle the error or re-throw it
  - The top-level error handler aborts the current transaction and resets the transaction's memory context
    - Releases all resources held by the transaction, including files, locks, memory, and buffer pins

Guidelines For Error Handling

- Custom error handlers can be defined via `PG_TRY()`
- Think about error handling!
  - *Never* ignore the return values of system calls
- Should your function return an error code, or `ereport()` on failure?
  - Probably `ereport()` to save callers the trouble of checking for failure
  - *Unless* the caller can provide a better (more descriptive) error message, or might not consider the failure to be an actual error
- Use assertions (`Assert`) liberally to detect programming mistakes, but *never* errors the user might encounter

Outline

1 Prerequisites
- Why Should You Hack On PostgreSQL?
- What Skills Will You Need?
- What Tools Should You Use?

2 The Architecture of PostgreSQL
- System Architecture
- Components of the Backend

3 Common Code Conventions
- Memory Management
- Error Handling

4 Community Processes

5 Sample Patch

6 Conclusion

Mailing Lists

- The vast majority of communication occurs on mailing lists
  - `pgsql-hackers` is the main list
  - `pgsql-patches` and `pgsql-committers` can be useful to learn from
- Written communication skills are important
  - Good developers are often good writers
- Some developers are on IRC; internals questions are welcome
  - `irc.freenode.net`, `#postgresql`

Your First Patch

- *Step 1:* Research and preparation
  - Is your new feature actually useful? Does it just scratch your itch, or is it of general value?
  - Does it need to be implemented in the backend, or can it live in pgfoundry, `contrib/`, or elsewhere?
  - Does the SQL standard define similar or equivalent functionality?
    - What about Oracle, DB2, . . . ?
  - Has someone suggested this idea in the past?
    - Search the archives and TODO list
  - Most ideas are bad
  - **Don't** take the TODO list as gospel

Sending A Proposal

- *Step 2:* Send a proposal for your feature to `pgsql-hackers`
  - Patches that appear without prior discussion risk wasting your time
- Discuss your proposed syntax and behaviour
  - Consider corner cases, and how the feature will relate to other parts of PostgreSQL (consistency is good)
  - Will any system catalog changes be required?
  - Backward-compatibility?
- Try to reach a consensus with `-hackers` on how the feature ought to behave

Implementation

- *Step 4:* Begin implementing the feature
- A general strategy is to look at how similar parts of the system function
  - Don't copy and paste (IMHO)
    - Common source of errors
  - Instead, read through similar sections of code to try to understand how they work, and the APIs they are using
  - Implement (just) what you need, refactoring the existed APIs if required
- Ask for advice as necessary (`-hackers` or IRC)
  - Write down the issues you encounter as you write the code, include the list when you submit the patch
- Consider posting work-in-progress versions of the patch

Testing, Documentation

- *Step 4:* Update tools
  - For example, if you've modified DDL syntax, update `psql`'s tab completion
  - Add `pg_dump` support if necessary
- *Step 5:* Testing
  - Make sure the existing regression tests don't fail
  - No compiler warnings
  - Add new regression tests for the new feature
- *Step 6:* Update documentation
  - Writing good documentation is more important than getting the DocBook details completely correct
  - Add new index entries, if appropriate
  - Check documentation changes visually in a browser

Submitting The Patch

- *Step 7:* Submit the patch
  - Use context diff format: `diff -c`
    - Unified diffs are okay for SGML changes
  - First, review every hunk of the patch
    - Is this hunk necessary?
    - Does it needlessly change existing code or whitespace?
    - Does it have any errors? Does it fail in corner cases? Is there a more elegant way to do this?
  - Work with a code reviewer to make any necessary changes
  - If your patch falls through the cracks, *be persistent*
    - The developers are busy and reviewing patches is difficult, time-consuming, and unglamourous work

Outline

1 Prerequisites
- Why Should You Hack On PostgreSQL?
- What Skills Will You Need?
- What Tools Should You Use?

2 The Architecture of PostgreSQL
- System Architecture
- Components of the Backend

3 Common Code Conventions
- Memory Management
- Error Handling

4 Community Processes

5 Sample Patch

6 Conclusion

The TABLESAMPLE Clause

- The TABLESAMPLE clause is defined by SQL:2003 and implemented by SQL Server and DB2
  - Oracle calls it SAMPLE, slightly different syntax
- Example query:

```
SELECT avg(salary)
FROM emp TABLESAMPLE SYSTEM (50);
```

The TABLESAMPLE Clause

- The TABLESAMPLE clause is defined by SQL:2003 and implemented by SQL Server and DB2
  - Oracle calls it SAMPLE, slightly different syntax
- Example query:

```
SELECT avg(salary)
FROM emp TABLESAMPLE SYSTEM (50);
```

- TODO item: "estimated_count(*)"

```
SELECT count(*) * 10
FROM t TABLESAMPLE SYSTEM (10);
```

The TABLESAMPLE Clause

- The TABLESAMPLE clause is defined by SQL:2003 and implemented by SQL Server and DB2
  - Oracle calls it SAMPLE, slightly different syntax
- Example query:

```
SELECT avg(salary)
FROM emp TABLESAMPLE SYSTEM (50);
```

- TODO item: "estimated_count(*)"

```
SELECT count(*) * 10
FROM t TABLESAMPLE SYSTEM (10);
```

- Straightforward to implement, but requires modifying some interesting parts of the system
- http://neilconway.org/tmp/tablesample.patch

What Does The Standard Say?

- Deciphering the SQL standard is notoriously difficult
  - I usually start with the index
- The `BERNOULLI` sample method sounds hard to implement
- `REPEATABLE` provides a way to seed the random number generator

Implementation Ideas

How Should We Implement Sampling?

- Simple approach: sequentially walk the heap, decide whether to skip a block using `random()` and the sampling percentage
- Therefore, add "sample scan" as a new scan type, analogous to sequential scan or index scan

Implementation Ideas

How Should We Implement Sampling?

- Simple approach: sequentially walk the heap, decide whether to skip a block using `random()` and the sampling percentage
- Therefore, add "sample scan" as a new scan type, analogous to sequential scan or index scan

Deficiencies

1. Non-uniform sampling when either
   - row size is non-uniform
   - distribution of live tuples is non-uniform
2. Consumes a lot of entropy
3. Could be optimized to reduce random I/O

Behavioral Questions

1. Can we specify `TABLEAMPLE` for non-base relation `FROM`-clause items? (Subqueries, SRFs, . . .)

Behavioral Questions

1. Can we specify `TABLEAMPLE` for non-base relation `FROM`-clause items? (Subqueries, SRFs, . . .)
2. Can we specify `TABLESAMPLE` for `UPDATE` or `DELETE`?

Behavioral Questions

1. Can we specify TABLEAMPLE for non-base relation FROM-clause items? (Subqueries, SRFs, . . .)
2. Can we specify TABLESAMPLE for UPDATE or DELETE?
3. Can we sample from the results of an index scan?

Behavioral Questions

1. Can we specify `TABLEAMPLE` for non-base relation `FROM`-clause items? (Subqueries, SRFs, . . .)
2. Can we specify `TABLESAMPLE` for `UPDATE` or `DELETE`?
3. Can we sample from the results of an index scan?
4. How does this interact with inheritance? Joins?

Implementation Plan

1. Modify the grammar to add support for parsing the `TABLESAMPLE` clause
2. Modify the nodes of the parse tree to allow `TABLESAMPLE` to be encoded in the AST

Implementation Plan

1. Modify the grammar to add support for parsing the `TABLESAMPLE` clause
2. Modify the nodes of the parse tree to allow `TABLESAMPLE` to be encoded in the AST
3. Create a new executor node for sample-based scans of a relation
4. Modify the planner to choose sample scans when appropriate, and to estimate the cost of evaluating a sample scan

Implementation Plan

1. Modify the grammar to add support for parsing the `TABLESAMPLE` clause
2. Modify the nodes of the parse tree to allow `TABLESAMPLE` to be encoded in the AST
3. Create a new executor node for sample-based scans of a relation
4. Modify the planner to choose sample scans when appropriate, and to estimate the cost of evaluating a sample scan
5. Implement the guts of the `SampleScan` executor node

Implementation Plan

1. Modify the grammar to add support for parsing the `TABLESAMPLE` clause
2. Modify the nodes of the parse tree to allow `TABLESAMPLE` to be encoded in the AST
3. Create a new executor node for sample-based scans of a relation
4. Modify the planner to choose sample scans when appropriate, and to estimate the cost of evaluating a sample scan
5. Implement the guts of the `SampleScan` executor node
6. Add support for `REPEATABLE`
7. Add support for `DELETE` and `UPDATE`
8. Update documentation
   - Can't easily add regression tests

Grammar Modifications

- Parsing TABLESAMPLE itself is quite easy
  - Add some new keywords: TABLESAMPLE and REPEATABLE must be made semi-reserved to avoid shift-reduce conflicts
- Checking SelectStmt reveals that relation_expr is the production for a base relation in the FROM clause with an optional alias and inheritance spec
- Unfortunately, relation_expr is also used by DDL commands, so create a new production and use it in the places we want to allow TABLESAMPLE

Parse Node Updates

- New parse node for the data `TABLESAMPLE` clause
- Need to attach new parse node to the AST somehow
  - The parser constructs a `RangeVar` for each `FROM` clause entry, so use that

Parse Node Updates

- New parse node for the data `TABLESAMPLE` clause
- Need to attach new parse node to the AST somehow
  - The parser constructs a `RangeVar` for each `FROM` clause entry, so use that

Range Table

The parse-analysis phase constructs a "range table" consisting of the `FROM` clause elements

- When converting the `FROM` clause RVs into range table entries (RTEs), attach the `TableSampleInfo`

Optimizer Terminology

RelOptInfo: Per-relation planner state. For each base rel or join, stores the estimated row count, row width, cheapest path, . . .

Path: Planner state for a particular way accessing a relation (or join relation); each `RelOptInfo` has a list of candidate paths

Optimizer Terminology

RelOptInfo: Per-relation planner state. For each base rel or join, stores the estimated row count, row width, cheapest path, . . .

Path: Planner state for a particular way accessing a relation (or join relation); each `RelOptInfo` has a list of candidate paths

Plan: A "finalized" output path: a node of the plan tree passed to the executor

- Once the planner has decided on the optimal `Path` tree, produce a corresponding `Plan` tree

Optimizer Modifications

- We need only modify stage 1 of the System R algorithm: finding the cheapest interesting paths for each base relation
  - Joins between sample scans not fundamentally different than normal joins
  - We *don't* need a `SamplePath` node; just use `Path`
- *Only* consider sample scans when a `TABLESAMPLE` clause is specified
- Simple cost estimation: assume we need to do a single I/O for each sampled page

Plan Trees

- Review: the planner produces a tree of `Plan` nodes
  - `Plan` nodes are treated as immutable by the executor
- The executor constructs a tree of `PlanState` nodes to describe the run-time state of a plan-in-execution
  - Each `PlanState` is associated with exactly one `Plan` node
  - `PlanState.plan` holds a `PlanState`'s associated `Plan` node

The "Iterator" API

Implemented By Each Executor Node

Mandatory

- **InitNode:** Given a `Plan` tree, construct a `PlanState` tree
- **ProcNode:** Given a `PlanState` tree, return next result tuple
  - Some plan nodes support bidirectional scans
- **EndNode:** Shutdown a `PlanState` tree, releasing resources

The “Iterator” API

Implemented By Each Executor Node

Mandatory

InitNode: Given a `Plan` tree, construct a `PlanState` tree

ProcNode: Given a `PlanState` tree, return next result tuple

- Some plan nodes support bidirectional scans

EndNode: Shutdown a `PlanState` tree, releasing resources

Optional

ReScan: Reset a `PlanState` so that it reproduces its output

MarkPos: Record the current position of a `PlanState`

RestrPos: Restore the position of a `PlanState` to last mark

The Buffer Manager

- Storage is organized into pages: constant-sized units of bytes
  - Pages are identified by their page number within a given file
- Almost all I/O is not done directly: to access a page, a process asks the buffer manager for it

The Buffer Manager

- Storage is organized into pages: constant-sized units of bytes
  - Pages are identified by their page number within a given file
- Almost all I/O is not done directly: to access a page, a process asks the buffer manager for it
- The buffer manager implements a hash table in shared memory, mapping page identifiers → buffers
  - If the requested page is in `shared_buffers`, return it
  - Otherwise, ask the kernel for it and stash it in `shared_buffers`
    - If there are no free buffers, replace an existing one
- Keep a pin on a page, to ensure it isn't replaced while in use

Executor Terminology

Block: A page on disk. Identified by a `BlockNumber`

Buffer: A page in memory. The buffer manager loads blocks from disk into buffers (`shared_buffers`)

OffsetNumber: Identifies an item within a page

Datum: An instance of a data type in memory

HeapTuple: A collection of `Datums` with a certain schema

EState: Run-time state for a single instance of the executor

Projection: The act of applying a target list

The Executor's TupleTable

- Tuples are passed around the executor using TupleTableSlots
- Different kinds of tuples:
  - Pointers into buffer pages
    - The output of a scan node, no projection
    - Need to drop pin on buffer when finished with tuple
  - Pointers into heap-allocated memory
    - Result of applying an expression: projection, SRFs, ...
    - Can be "minimal" tuples: no MVCC metadata needed
    - Need to `pfree()` tuple when finished
  - "Virtual" tuples
- The `TupleTableSlot` abstraction papers over all these details

Implementing The Executor Node

Initialization

- Most of this is boilerplate code :-(
- Initialize executor machinery needed to evaluate quals and do projection
- Read-lock the relation: no DDL changes allowed while we're scanning

Implementing `REPEATABLE`

- Simple implementation: pass the repeat seed to `srandom()`

Implementing `REPEATABLE`

- Simple implementation: pass the repeat seed to `srandom()`
- Wrong: if the execution of multiple sample scans is interleaved, they will stomp on the other's PRNG state
- Therefore, use `initstate()` to give each sample scan its own private PRNG state

Supporting UPDATE and DELETE

Implementation of UPDATE and DELETE

- Run the executor to get "result tuples"
- Mark the result tuples as expired ("deleted by my transaction") on disk
- If `UPDATE`, insert a new tuple

Supporting UPDATE and DELETE

Implementation of UPDATE and DELETE

- Run the executor to get "result tuples"
- Mark the result tuples as expired ("deleted by my transaction") on disk
- If `UPDATE`, insert a new tuple

TABLESAMPLE support

- Quite easy: basically comes for free!
- `relation_expr` is already used by the `DELETE` and `UPDATE`
  - Modify to use `relation_expr_opt_sample`
- Hackup parse-analysis to attach `TableSampleInfo`

Possible Improvements

1. Implement the `BERNOULLI` sample method

Possible Improvements

1. Implement the `BERNOULLI` sample method
2. Support non-integer sample percentage and repeat seed

Possible Improvements

1. Implement the `BERNOULLI` sample method
2. Support non-integer sample percentage and repeat seed
3. Take advantage of optimizer statistics to produce a more accurate sample

Possible Improvements

1. Implement the `BERNOULLI` sample method
2. Support non-integer sample percentage and repeat seed
3. Take advantage of optimizer statistics to produce a more accurate sample
4. Support mark-and-restore; allow a `SampleScan` to be re-scanned when possible

Possible Improvements

1. Implement the `BERNOULLI` sample method
2. Support non-integer sample percentage and repeat seed
3. Take advantage of optimizer statistics to produce a more accurate sample
4. Support mark-and-restore; allow a `SampleScan` to be re-scanned when possible
5. Provide information about the degree of confidence in the sampled results

Possible Improvements

1. Implement the `BERNOULLI` sample method
2. Support non-integer sample percentage and repeat seed
3. Take advantage of optimizer statistics to produce a more accurate sample
4. Support mark-and-restore; allow a `SampleScan` to be re-scanned when possible
5. Provide information about the degree of confidence in the sampled results
6. "Page at a time" scan mode

Outline

1 Prerequisites
- Why Should You Hack On PostgreSQL?
- What Skills Will You Need?
- What Tools Should You Use?

2 The Architecture of PostgreSQL
- System Architecture
- Components of the Backend

3 Common Code Conventions
- Memory Management
- Error Handling

4 Community Processes

5 Sample Patch

6 Conclusion

Next Steps

1. Sign up to the development lists
2. Setup your local development environment
3. Participate in development discussions
   - Read design proposals, ask questions/give feedback
   - Try to reproduce (and fix!) reported bugs
   - Look at proposed patches
   - Help out with administrativia, contribute to the documentation
4. Read the code!
5. Look for a small project that piques your interest, and get started!

Q & A

Any questions?