Introduction to Hacking PostgreSQL

With lots of code review!

Gavin Sherry
gavin@alcove.com.au

Alcove Systems Engineering

January 14, 2007

Outline

Databases are cool

- Everyone uses them
- There are only a small number – begging for diversification
- Many open problems to be solved
    - Example: Storage capacity doubles each year, bandwidth remains constant and latency goes *up!*
- *You* can solve lots of the cool problems
- Database guys get the babes (or guys (or what ever))

Everyone wants database hackers

- The number of hard core database kernel developers in the world is low
- Demand is high
- Employers are cool: Google, MSR, CERN, NASA, JPL, Los Alamos, heaps of startups…

So, why PostgreSQL?

- Most database technologies come from “System/R” or “Berkeley/INGRES/POSTGRES”
- Many high end databases are based on Postgres
  - Netezza
  - ParACCEL
  - Greenplum DB
  - Streaming technologies like Telegraph CQ and StreamBase
  - ...
- Other database technologies came out of PostgreSQL
  - Informix
  - Illustra
- It’s BSD, you can do what you want
- Hacking on PostgreSQL is easier than hacking on any other large, mature source base I’ve looked at

Outline

1. Why hack on Postgres?
2. Development Environment
3. PostgreSQL Architecture
4. Conventions, and all that
5. Submitting a patch
6. Some real code

Development Environment

The basics

- Most of the Postgres developers use Unix
- You'll need to know C
- *C is easy*
- Code reading is hard, but worth getting good at

Development Tools

The Basics

- gcc, Bison, Flex, CVS, autotools, gdb
- Configure flags:
  - `--enable-depend`
  - `--enable-debug`
  - `--enable-cassert`
  - `CFLAGS=-O0`

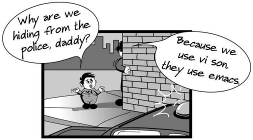
Development tools cont.

- tags or cscope are essential
  - "What is the defi nition of this function/type?"
  - "What are all the call-sites of this function?"
  - src/tools/make_[ce]tags
- ccache and distcc are useful, especially on slower machines
- valgrind can be useful for debugging memory errors

Text Editor

Coding

- Use a proper text editor
- Follow the style of the surrounding code (Basically, Allman BSD)
- **If you don't follow the style, it will just slow down the review process**



Outline

PostgreSQL Architecture

- Five main components:
  1. The **parser** - parse the query string
  2. The **rewriter** - apply rewrite rules
  3. The **optimizer** - determine an effi cient query plan
  4. The **executor** - execute a query plan
  5. The **utility processor** - process DDL like CREATE TABLE

PostgreSQL Architecture

- Five main components:
  1. The **parser** - parse the query string
  2. The **rewriter** - apply rewrite rules
  3. The **optimizer** - determine an effi cient query plan
  4. The **executor** - execute a query plan
  5. The **utility processor** - process DDL like CREATE TABLE

PostgreSQL Architecture

- Five main components:
  1. The **parser** - parse the query string
  2. The **rewriter** - apply rewrite rules
  3. The **optimizer** - determine an effi cient query plan
  4. The **executor** - execute a query plan
  5. The **utility processor** - process DDL like CREATE TABLE

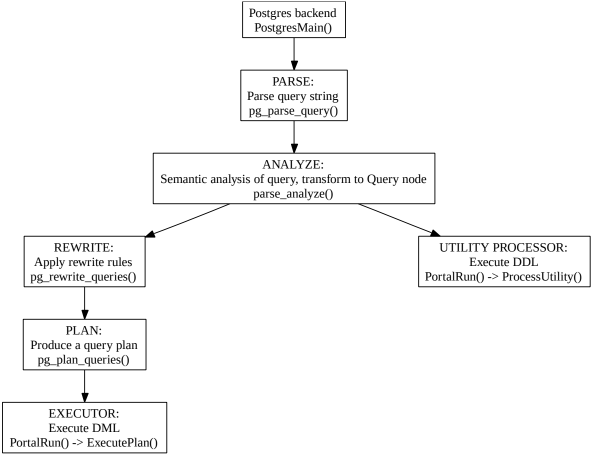
PostgreSQL Architecture

- Five main components:
  1. The **parser** - parse the query string
  2. The **rewriter** - apply rewrite rules
  3. The **optimizer** - determine an effi cient query plan
  4. The **executor** - execute a query plan
  5. The **utility processor** - process DDL like CREATE TABLE

PostgreSQL Architecture

- Five main components:
  1. The **parser** - parse the query string
  2. The **rewriter** - apply rewrite rules
  3. The **optimizer** - determine an effi cient query plan
  4. The **executor** - execute a query plan
  5. The **utility processor** - process DDL like `CREATE TABLE`

Architecture Diagram



The Parser

- Lex and parse the query string submitted by the user
  - `parser/gram.y` has the guts; entry point is `parser/parser.c`
- Produces a "raw parsetree": a linked list of parse nodes
  - Parse nodes are defi ned in `include/nodes/parsenodes.h`
- There is usually a simple mapping between grammar productions and parse node structure

Semantic Analysis

- In the parser itself, only **syntactic** analysis is done
- Next comes **semantic analysis**
  - `parser/analyze.c` and related code under `parser/`

Semantic analysis

- Resolve column references
- Verify that target schemas, tables and columns exist
- Check that the types used in expressions are consistent
- In general, check for errors that are impossible or diffi cult to detect in the parser itself

```
SELECT a, b, c FROM
t1, t2, t3 WHERE...
```

Rewriter, Planner

- The analysis phase produces a `Query`, which is the query's parse tree
- The rewriter applies rewrite rules: view definitions and ordinary rules. 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 ought to be executed
  - Only needed for "optimizable" statements (`INSERT`, `DELETE`, `SELECT`, `UPDATE`)

Executor, Utility Processor

- DDL statements are "executed" via the utility processor, which basically just calls the appropriate function for each different kind of DDL statement
  - `processUtility()` in `tcop/utility.c`; the implementation of the DDL statements is in `commands/`
- Optimizeable statements are processed via the Executor: given a `Plan`, it executes the plan and produces any resulting tuples
  - `executor/`; entry point is in `execMain.c`

Outline

Common Idioms: Nodes

- Postgres uses a very 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
- The first field of *any* `Node` is a `NodeTag`, which can be used to determine a `Node`'s specific type at runtime

Nodes, Cont.

Important things to keep in mind

- 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, Cont.

Important things to keep in mind

- 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, Cont.

Important things to keep in mind

- 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, Cont.

Important things to keep in mind

- 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, Cont.

Important things to keep in mind

- 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, Cont.

Important things to keep in mind

- 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` if your `Node` is to be serialised/deserialised
- Grepping for references to the node's type can be helpful to make sure you don't forget to update anything

Memory Management

- Postgres uses hierarchical, region-based memory management, and it absolutely rocks
  - `backend/util/mmgr`
- Memory is allocated via `palloc()`
- All allocations occur inside a *memory context*
- Default memory context: `CurrentMemoryContext`

Memory Management, cont.

- Allocations can be freed individually via `pfree()`
- When a memory context is reset, 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 is short lived, who cares
  - Calculate just how much memory you might consume in a worst case
- Be aware of the memory allocation assumptions made by functions you call
- Memory leaks, *per se*, are rare 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(3)`
  - `elog(ERROR)` walks back up the stack
  - 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

Error Handling, Cont.

- 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()` – think about why...and why not
- Use assertions (`Assert`) liberally to detect programming errors, but *never* errors the user might encounter

Example

```
Assert(BufferIsValid(buf);
PG_TRY();
{
  ...
}
PG_CATCH();
{
  ...
  if (...)
    PG_RE_THROW();
  else
    ...
}
PG_END_TRY();
```

Outline

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 belong in the backend?
- 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

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 behavior
  - Consider corner cases, and how the feature will relate to other parts of PostgreSQL (consistency is good)
  - Will any system catalog changes be needed?
  - Backward-compatibility?
- Try to reach a consensus with `-hackers` on how the feature ought to behave

Implementation

Step 3: Implement the patch

- A general strategy is to look at how similar parts of the system function
  - *Don't copy and paste* (IMHO)
  - 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 implementation advice as needed (`-hackers` or IRC)
- Consider posting work-in-progress versions of the patch

Thoroughness

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

Thoroughness

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

Submitting The Patch

Step 6: Update documentation

- `make check` in `doc/src/sgml` does a syntax check that is faster than building the whole SGML docs
- Check documentation changes visually in a browser

Step 7: Submit the patch

- Use context diff format: `diff -c`
- Review *every* hunk of the patch
  - Is this hunk necessary? Are there errors?
  - Does it needlessly change whitespace or existing code?
- Work with a code reviewer to make any necessary changes
- If your patch falls through the cracks, *be persistent*

Submitting The Patch

Step 6: Update documentation

- `make check` in `doc/src/sgml` does a syntax check that is faster than building the whole SGML docs
- Check documentation changes visually in a browser

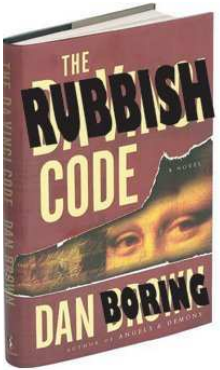
Step 7: Submit the patch

- Use context diff format: `diff -c`
- Review *every* hunk of the patch
  - Is this hunk necessary? Are there errors?
  - Does it needlessly change whitespace or existing code?
- Work with a code reviewer to make any necessary changes
- If your patch falls through the cracks, *be persistent*

Outline

Time for some actual code

- We'll look at a patch I "prepared earlier"
- It's against PostgreSQL CVS HEAD
- You can see it here: `http://alcove.com.au/when_clause.patch`



WHEN Clause

- We'll be walking you through the implementation of the `WHEN` clause for `CREATE TRIGGER`
  - You can see a patch at `http://neilconway.org/talks/hacking/when_clause.patch`
- Defined by SQL:2003, implemented by Oracle and others
- Optional clause; when the `WHEN` expression evaluates to `false` (or NULL), the associated trigger is not fired
- In the `WHEN` clause, `OLD` and `NEW` tuples can be referenced:
  - In `UPDATE` and `DELETE` triggers, `OLD` is the tuple being replaced
  - In `UPDATE` and `INSERT` triggers, `NEW` is the tuple being added

WHEN Clause Considerations

- Syntax is easy: defined by SQL spec

Syntax

```
CREATE TRIGGER name { BEFORE | AFTER }
    { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( expr ) ]
    EXECUTE PROCEDURE funcname ( arguments )
```

WHEN Clause Considerations, cont.

- Behavioral questions:
  - Should we allow `WHEN` clause for statement-level triggers? (SQL spec doesn't specify)
  - What subset of SQL should we allow? Aggregate functions, subqueries, . . . ?
- No backward-compat concerns
- Obviously needs to be in the backend
- Useful for at least SQL-spec compliance

Implementation Outline

1. Add support for the new syntax to the parser
2. Update the `CREATE TRIGGER` parsenode
3. Add support for `WHEN` clause to analysis phase
4. Add new field to `pg_trigger` system catalog, containing the `WHEN` clause
5. Modify implementation of `CREATE TRIGGER` to add the `WHEN` clause to the new `pg_trigger` row
6. Add support for the `WHEN` clause when firing triggers in the executor (most of the difficulty is here)
7. Update `pg_dump/psql` to support the `WHEN` clause

Implementation Outline

1. Add support for the new syntax to the parser
2. Update the `CREATE TRIGGER` parsenode
3. Add support for `WHEN` clause to analysis phase
4. Add new field to `pg_trigger` system catalog, containing the `WHEN` clause
5. Modify implementation of `CREATE TRIGGER` to add the `WHEN` clause to the new `pg_trigger` row
6. Add support for the `WHEN` clause when firing triggers in the executor (most of the difficulty is here)
7. Update `pg_dump/psql` to support the `WHEN` clause

Implementation Outline

1. Add support for the new syntax to the parser
2. Update the `CREATE TRIGGER` parsenode
3. Add support for `WHEN` clause to analysis phase
4. Add new field to `pg_trigger` system catalog, containing the `WHEN` clause
5. Modify implementation of `CREATE TRIGGER` to add the `WHEN` clause to the new `pg_trigger` row
6. Add support for the `WHEN` clause when firing triggers in the executor (most of the difficulty is here)
7. Update `pg_dump/psql` to support the `WHEN` clause

Implementation Outline

1. Add support for the new syntax to the parser
2. Update the `CREATE TRIGGER` parsenode
3. Add support for `WHEN` clause to analysis phase
4. Add new field to `pg_trigger` system catalog, containing the `WHEN` clause
5. Modify implementation of `CREATE TRIGGER` to add the `WHEN` clause to the new `pg_trigger` row
6. Add support for the `WHEN` clause when firing triggers in the executor (most of the difficulty is here)
7. Update `pg_dump/psql` to support the `WHEN` clause

Implementation Outline

1. Add support for the new syntax to the parser
2. Update the `CREATE TRIGGER` parsenode
3. Add support for `WHEN` clause to analysis phase
4. Add new field to `pg_trigger` system catalog, containing the `WHEN` clause
5. Modify implementation of `CREATE TRIGGER` to add the `WHEN` clause to the new `pg_trigger` row
6. Add support for the `WHEN` clause when firing triggers in the executor (most of the difficulty is here)
7. Update `pg_dump`/`psql` to support the `WHEN` clause

Implementation Outline

1. Add support for the new syntax to the parser
2. Update the `CREATE TRIGGER` parsenode
3. Add support for `WHEN` clause to analysis phase
4. Add new field to `pg_trigger` system catalog, containing the `WHEN` clause
5. Modify implementation of `CREATE TRIGGER` to add the `WHEN` clause to the new `pg_trigger` row
6. Add support for the `WHEN` clause when firing triggers in the executor (most of the difficulty is here)
7. Update `pg_dump/psql` to support the `WHEN` clause

Implementation Outline

1. Add support for the new syntax to the parser
2. Update the `CREATE TRIGGER` parsenode
3. Add support for `WHEN` clause to analysis phase
4. Add new field to `pg_trigger` system catalog, containing the `WHEN` clause
5. Modify implementation of `CREATE TRIGGER` to add the `WHEN` clause to the new `pg_trigger` row
6. Add support for the `WHEN` clause when firing triggers in the executor (most of the difficulty is here)
7. Update `pg_dump/psql` to support the `WHEN` clause

Implementation outline cont.

8. Lose much sleep

Parser Changes

- Trivial, as it turns out — see page 2

Grammar code

```
CreateTrigStmt:
            CREATE TRIGGER name TriggerActionTime TriggerEvents ON
            qualified_name TriggerForSpec TriggerWhen EXECUTE PROCEDURE
            func_name '(' TriggerFuncArgs ')'
{
    CreateTrigStmt *n = makeNode(CreateTrigStmt);
    /* ... */
    n->when = $10;
    $$ = (Node *) n;
}

TriggerWhen:
          WHEN '(' a_expr ')'                 { $$ = $3; }
          | /*EMPTY*/                         { $$ = NULL; }
```

Parsenode Changes

- The definition of the `CreateTrigStmt` parse node is closely derived from the syntax of `CREATE TRIGGER`
- Add a new field to the struct to stash the `WHEN` clause
- Be sure to update `equalfuncs.c` and `copyfuncs.c`
- See pages 3 and 4 of handout
- *Next*: update the analysis phase. How do we parse `WHEN` clauses like `OLD.a <> NEW.a`?

Expressions In Postgres

- The `WHEN` clause is a boolean expression
- An expression is a tree of `Expr` nodes
  - There are `Expr` subclasses for the different kinds of expressions: function calls, operator invocations, constants, variables, etc.
- `ExecEvalExpr()` evaluates an expression by recursing through this tree. For example:
  - A function is evaluated by fi rst evaluating its arguments, then calling the function itself
  - A constant value is trivial to evaluate
- See pages 4 through 6 of handout

Variable Expressions

- In an expression like $t.a > 10$, $t.a$ is a *range variable*, colloquially known as a table column
  - Represented by the `Var` expression type
- How are range variables implemented?
  - `Var.varno` identifi es the variable's table (*t* above)
  - `Var.varattno` is the attribute number of the variable's column
- `varno` is an index into the expression's *range table*
  - The range table is the set of relations that can be referenced in expressions — each `Query` has an associated range table

Analysis Phase

- The analysis phase is where we lookup identifiers; therefore, during the analysis phase, we need to add range table entries for the `NEW` and `OLD` relations
- Other analysis phase work is straightforward:
  - Exclusive-lock the target relation
  - Disallow subqueries and aggregates in the `WHEN` clause
- See pages 4 through 6 of the handout

System Catalogs

- The format of the system catalogs is defined by header files, in the `src/include/catalog` directory
  - These fi les are normal C headers, with some special macros
  - These macros are pre-processed for bootstrapping (`initdb`)
- Nice effect: access to system catalog fields is the same as accessing a C `struct`
- A compiled copy of the backend depends upon the exact definition of the system catalogs
  - If you modify the system catalog format, bump the catalog version to force `initdb`
- See pages 6 and 7

System Catalog Changes

- Triggers are stored in the `pg_trigger` catalog
- To add support for `WHEN`, we add a new field to `FormData_pg_trigger` in `pg_trigger.h`
- Add `tgqual` field, which stores a serialized version of the `WHEN` expression tree
  - Review: `nodeToString()` serializes a `Node`
  - We can use `stringToNode()` to reconstruct the expression tree when needed

CREATE TRIGGER Changes

- `CREATE TRIGGER` needs to store the textual representation of the `WHEN` clause in the new row it inserts into `pg_trigger`
- Also reject `WHEN` clause for statement-level triggers here
- Also create a dependency between the elements of the `WHEN` expression and the trigger
  - If the `WHEN` clause references column *a* of the table, `DROP COLUMN` a should be disallowed (without cascade)
- See page 8

TriggerDesc Updates

- The `Relation` struct contains metadata about an opened relation: the relation's `pg_class` row, a description of the format of its tuples, associated indexes, associated triggers, etc.
  - Stored in the *relcache*
- See pages 7 and 8

TriggerDesc Updates, cont.

- Trigger information is stored in a subsidiary struct, `TriggerDesc`, which itself contains a `Trigger` struct for each trigger on the relation
  - Add a field to `Trigger` to store the `WHEN` clause
  - Fill it in when `TriggerDesc` constructed
- Remember to update support functions!
  - `FreeTriggerDesc()`, `CopyTriggerDesc()`, `equalTriggerDescs()`

Executor Changes

- The guts of the required changes are in the executor
- We need to evaluate the `WHEN` clause before we fire a row-level trigger
- To do that, we need to:
  - Preprocess the `WHEN` clause to get it ready to be evaluated
  - Teach the executor to be able to evaluate expressions referencing the `NEW` and `OLD` relations
- See pages 9 through 12

OLD and NEW in Executor

- Review: `ExecEvalExpr()` evaluates expression trees
- To do so, it uses an `ExprContext`
  - All info needed to evaluate an expression
  - To evaluate an expression, you find an appropriate `ExprContext`, setup the necessary information, then use `ExecEvalExpr()`
    - The executor keeps a “per-tuple `ExprContext`” that we can use: it is reset for each tuple that is output
- See pages 10

Evaluating Variable Expressions

`ExecEvalVar()` is called by `ExecEvalExpr()` to evaluate `Var` expressions:

`ExecEvalVar` change

```
switch (variable->varno)
{
    case INNER: /* get the tuple from the inner node */
        slot = econtext->ecxt_innertuple;
        break;

    case OUTER: /* get the tuple from the outer node */
        slot = econtext->ecxt_outertuple;
        break;

    default:    /* get the tuple from the relation being scanned */
        slot = econtext->ecxt_scantuple;
        break;
}
```

Evaluating Variables

- Note that the `varno` is ignored, except for the special `INNER` and `OUTER` varnos
  - The code assumes that the caller will insert the current tuple into the `ExprContext`'s “scan tuple” slot before calling `ExecEvalExpr`
- This won’t work for us: the `WHEN` expression could reference *two* different tuples (`OLD` and `NEW`)
- How can we solve this?

Solution

- Add two more special varnos, `TRIG_OLD_VARNO` and `TRIG_NEW_VARNO`
- In the analysis phase, rewrite the varnos in the expression so that references to the special relations are assigned the right varno
  - Machinery for this exists: `ChangeVarNodes` walks an expression tree, changing varno $x \rightarrow y$ in every node of the tree
- Add two new slots to `ExprContext` to hold the `OLD` and `NEW` tuples, and setup these slots before calling `ExecEvalExpr`
- In `ExecEvalVar`, add two more special-cases for the two special varnos, fetching from the appropriate slots of the `ExprContext`

Checking The Qualification

- Before firing triggers, check the `WHEN` clause
- For `BEFORE` triggers, this is easy. Add code to invoke `ExecQual()` to:
  - `ExecBRDeleteTriggers()`
  - `ExecBRInsertTriggers()`
  - `ExecBRUpdateTriggers()`
- Use the current executor instance to get per-tuple `ExprContext`; try to avoid overhead by preparing `WHEN` expression the first time the trigger is fired for this command
- See pages 10 and 11

AFTER Trigger Support

- Unfortunately, supporting `AFTER` triggers is not so easy
- `AfterTriggerSaveEvent()` enqueues a trigger to be invoked later, such as at the end of the current query
- We *can't* check the `WHEN` condition here
- Instead, we need to check the `WHEN` condition when the saved events are fired — but we won't necessarily have an executor instance to use!
  - Should just be a Small Matter of Programming

Subqueries in `WHEN` clause

- Subqueries in the `WHEN` clause would be convenient
- Unfortunately, they're hard to implement
- We'd have to run the full-fledged query planner on the expression
- Postgres has the infrastructure to do this, it's just a matter of using it
- All the other code we've written is prepared to handle subqueries

psql Support

- psql's \d command includes the definitions of the triggers on a table. How do we get it to include the WHEN clause?
- psql gets trigger definitions by calling the backend function pg_get_triggerdef(), so we need to update it
- There is already machinery for pretty-printing expressions as SQL text, so we can reuse all that
- One hurdle: tgqual may contain the special TRIG_OLD_VARNO and TRIG_NEW_VARNO varnos, which the expression printing code doesn't understand
  - Quick hack: use ChangeVarNodes() to switch back to original varnos
- See pages 12 and 13

pg_dump Support

- We need to update pg_dump to dump `WHEN` clause
- pg_dump reconstructs the `CREATE TRIGGER` command for a trigger by examining the trigger's `pg_trigger` row
- For `WHEN`, this isn't so easy:
  - `tgqual` references `TG_OLD_VARNO` and `TG_NEW_VARNO`, so there is no easy way to reconstruct `tgqual` in a client app
- Change pg_dump to use `pg_get_triggerdef()` to send a fully-formed `CREATE TRIGGER` to the client
- See pages 12 and 13

Regression Tests

- Invoked by `make check`
- Run out of `src/test/regress`
- Put tests in `sql/triggers.sql`
- Reflect changes in `expected/triggers.out`
- See pages 13 and 14

Documentation

- Documentation is in DocBook SGML
- Located in `docs/src/sgml`
- SQL command reference in `ref/create_trigger.sgml`
- Be sure to add an example
- See page 14 and 15

TODO Items

- As implemented, the patch has some deficiencies:
  - No support for `AFTER` triggers
  - No support for subqueries in the `WHEN` clause
  - Leaks the `when` field in `FreeTriggerDesc()`
  - `setup_trigger_quals()` does some redundant work

Where to go from here

- Sign up to `-hackers`, `-patches` and `-committers`
- Review incoming patches, particularly those from active hackers, learn from them
- Review the todo list: `http://developer.postgresql.org`
- Choose some small project (denoted by % on the official TODO list)
- Read the code as much as possible

Where to go from here

- Sign up to `-hackers`, `-patches` and `-committers`
- Review incoming patches, particularly those from active hackers, learn from them
- Review the todo list: `http://developer.postgresql.org`
- Choose some small project (denoted by % on the official TODO list)
- Read the code as much as possible

Where to go from here

- Sign up to `-hackers`, `-patches` and `-committers`
- Review incoming patches, particularly those from active hackers, learn from them
- Review the todo list: `http://developer.postgresql.org`
- Choose some small project (denoted by % on the official TODO list)
- Read the code as much as possible

Where to go from here

- Sign up to `-hackers`, `-patches` and `-committers`
- Review incoming patches, particularly those from active hackers, learn from them
- Review the todo list: `http://developer.postgresql.org`
- Choose some small project (denoted by % on the official TODO list)
- Read the code as much as possible

Where to go from here

- Sign up to `-hackers`, `-patches` and `-committers`
- Review incoming patches, particularly those from active hackers, learn from them
- Review the todo list: `http://developer.postgresql.org`
- Choose some small project (denoted by % on the official TODO list)
- Read the code as much as possible