Hacking the Query Planner

Tom Lane
Red Hat
PGCon 2011

Why this talk?

One thing I could really use is an overview of what order things get done in the planner. What are the major phases of processing and what's the function of each one? — R. Haas

Agenda

- **What's the problem we need to solve?**
- Structure of the planner
- Some key data structures
- Cost estimation
- Future work

Overall backend structure

- Parser
  - Determines the semantic meaning of a query string
- Rewriter
  - Performs view and rule expansion
- Planner
  - Designs an execution plan for the query
- Executor
  - Runs the plan

The planner’s problem

- Find a good query plan
- Don't spend too much time (or memory) finding it
- Support the extensible aspects of Postgres
  - eg, custom data types, operators, functions
  - this means hard-wired knowledge about data types and operators should be avoided as much as possible

What's a plan, exactly?

- A plan is a tree of plan nodes
- Each plan node represents a specific type of processing to do, with all details executor needs
- At execution, a node yields a stream of tuples
- Relation scan nodes get their tuples from a table
- Most other node types read tuple stream(s) from child plan nodes, and process them somehow to create their result stream

Types of plan nodes

- Relation scan nodes
  - Sequential, plain index, bitmap index
- Join nodes
  - Nestloop, nestloop with inner indexscan, hash, merge
- Special plan nodes
  - Sort, aggregate, set operations (`UNION` etc)

Attributes of a plan node

- Data source
  - Relation to scan, if a table scan
  - Input plan node, if a “processing” node
  - Two input plan nodes, if a join node
- Target list (expressions to compute and return)
  - Think `SELECT` list
- Selection conditions (“qualifiers” or “quals”)
  - Think `WHERE` conditions

Estimates for a plan node

- Output row count
  - Need this to estimate sizes of upper joins
- Average row width
  - Need both rowcount and width to estimate workspace for sorts, hashes, etc that must store node's output
- Total cost
  - Usually, the thing we want to minimize
- Startup cost
  - For `LIMIT` queries, we interpolate between startup and total costs, since not all rows will be fetched

Agenda

- What's the problem we need to solve?
- **Structure of the planner**
- Some key data structures
- Cost estimation
- Future work

But first, some jargon

- **Var** = variable = table column reference
- **Rel** = relation = real or virtual table
- **Base rel** = primitive `FROM` item (actual table, or separately-planned subquery, or set-returning function)
- **Join rel** = join relation (the result of joining a specific set of base relations)
- **Qual** = qualifier = `WHERE` clause or filter condition
- **Join qual** = qualifier that uses Vars from more than one base relation

Phases of planning

- Preprocessing
  - simplify the query if possible; collect information
- Scan/join planning
  - decide how to implement `FROM`/`WHERE`
- Query special feature handling
  - deal with plan steps that aren't scans or joins
- Postprocessing
  - convert results into form the executor wants

Early preprocessing

- Simplify scalar expressions
- Expand simple SQL functions in-line
- Simplify join tree

Simplify scalar expressions

What we know how to do is mainly constant-folding:

```
2 + 2  ⇒  4
```

```
CASE WHEN 2+2 = 4 THEN x+1
ELSE 1/0 END
```

```
   ⇒ x+1
```

… **not** “ERROR: division by zero”, please

Why bother simplifying?

- Do computations only once, not once per row
- View expansion and SQL function inlining can expose constant-folding opportunities not visible in the original query, so query author wasn't necessarily stupid
- Simplifying takes a lot of load off the estimation functions, which by and large can't cope with anything more complex than Variable = Constant

Constant-folding is simple

- All we need for constant-folding is the ability to hand an expression tree to the executor to execute; the planner need know nothing of the operation's detailed semantics

- People are sometimes surprised that we don't simplify cases like reducing `x + 0` ⇒ `x`

- Problem is that would require a **lot** of datatype-specific and operator-specific knowledge, plus infrastructure for extensions to add such knowledge

- … but maybe someday …

Expand simple SQL functions

```
CREATE FUNCTION incr(int) RETURNS int
AS ‘SELECT $1 + 1’ LANGUAGE SQL;
```

```
SELECT incr(col) FROM tab;
```

$\Rightarrow$

```
SELECT col + 1 FROM tab;
```

Simplify join tree

- Flatten ("pull up") sub-selects if possible
  - else, we'll recurse to generate sub-plans
- Flatten `UNION ALL`, expand inheritance trees
- Reduce join strength (outer join $\Rightarrow$ inner join)
- Convert `IN`, `EXISTS` sub-selects to semi-joins
- Identify anti-joins

Flattening a simple view

```
CREATE VIEW v AS
 SELECT a, b+c AS d FROM t WHERE x > 0;
```

```
SELECT v.a, v.d FROM v WHERE v.a = 42;
```

Rewriter produces:

```
SELECT v.a, v.d FROM
 (SELECT a,b+c AS d FROM t WHERE x > 0) v
 WHERE v.a = 42;
```

Sub-select flattening produces:

```
SELECT t.a, t.b + t.c FROM t
 WHERE t.x > 0 AND t.a = 42;
```

I lied about the ordering ...

- Actually, these preprocessing steps are done in a very specific order so that opportunities exposed by one step can be exploited later
- Some of them have to be intermixed in a single recursive pass over the query tree
- Getting all the optimizations to happen without duplicate processing is a bit tricky

Later preprocessing

- Determine where `WHERE` clauses ("quals") should be evaluated
  - In general, we want to use each qual at the lowest possible join level
- Identify all referenced table columns (Vars), and find out how far up in the join tree their values are needed
- Build equivalence classes for provably-equal expressions
- Gather information about join ordering restrictions
- Remove useless joins (needs results of above steps)

Scan/join planning

- Basically deals with the `FROM` and `WHERE` parts of the query
- Knows about `ORDER BY` too
  - mainly so that it can design merge-join plans
  - but also to avoid final sort if possible
- Cost estimate driven

Scan/join planning

- Identify feasible scan methods for base relations, estimate their costs and result sizes
- Search the join-order space, using dynamic programming or heuristic "GEQO" method, to identify feasible plans for join relations
- Honor outer-join ordering constraints
- Produce one or more "Path" data structures

Paths versus Plans

- A **Path** is a simplified representation of a potential plan tree
- We build many Paths during planning, but convert only the finally selected Path to a full-fledged Plan that the executor could handle
- Saves time, memory space when considering a large number of competing alternative plans

Path generation/comparison

- Generate a Path data structure for each feasible method of performing a scan or join
- Compare cost to previously-generated Paths for the same base relation or join relation
- Immediately discard inferior Paths
  - Keep only those that are cheapest (in either total or startup cost) for a given output sort ordering of their relation
  - Get rid of useless sort orderings, too

Join searching

- Multi-way joins have to be built up from pairwise joins, because that's all the executor knows how to do

- For any given pairwise join step, we can identify the best input Paths and join methods via straightforward cost comparisons, resulting in a list of Paths much as for a base relation

- Finding the best ordering of pairwise joins is the hard part

Join searching

- We usually have many choices of join order for a multi-way join query, and some orders will be cheaper than others
- If the query contains only plain inner joins, we can join the base relations in any order
- Outer joins can be re-ordered in some but not all cases; we handle that by checking whether each proposed join step is legal

Standard join search method

- Generate paths for each base relation
- Generate paths for each possible two-way join
- Generate paths for each possible three-way join
- Generate paths for each possible four-way join
- Continue until all base relations are joined into a single join relation; then use that relation's best path
- This "dynamic programming" approach was invented many years ago for IBM's System R

Inner indexscans are special

- The dynamic programming method supposes that any join is formed from independent Paths for the two input relations
- Doesn't work for nestloop with inner indexscan, because using a join clause as an index condition requires the outer variable(s) to be available from the particular outer relation being joined to
- We keep separate lists of "inner indexscan" Paths for each base relation that has any indexable join clauses, organized according to required outer rels

Join searching is expensive

- An **n**-way join problem can potentially be implemented in **n!** (**n** factorial) different join orders
- Considering all possibilities gets out of hand **real fast**, and is not feasible for queries with more than around ten base relations
- We use a few heuristics, like not considering clause-less joins
- With too many relations, fall back to “GEQO” (genetic query optimizer) search, which is even more heuristic and tends to fail to find desirable plans

Heuristics used in join search

- Don't join relations that are not connected by any join clause, unless forced to by join-order restrictions
  - Implied equalities count as join clauses, so this rule seldom leads us astray
- Break down large join problems into sub-problems according to the syntactic `JOIN`/sub-select structure
  - Actually, it's done by not merging sub-problems to make a big problem in the first place (see "collapse limits")
  - This frequently sucks; would be useful to look for smarter ways of subdividing large join trees

Genetic query optimizer

- Treats join order searching as a Traveling Salesman Problem, i.e., minimize the length of a "tour" visiting all "cities" (base relations)
- Does a partial search of the tour space using heuristics found useful for TSP
- Problem: join costs don't behave very much like inter-city distances; they interact. This makes the TSP heuristics not so effective
- This area desperately needs improvement

Query special feature handling

- Deal with `GROUP BY`, `DISTINCT`, aggregate functions, window functions
- Deal with `UNION`/`INTERSECT`/`EXCEPT`
- Apply final sort if needed for `ORDER BY`
- This code is very ad-hoc, not very pretty, not terribly bright either
- Maybe someday we will rewrite into generate-and-compare-paths style

Postprocessing

- Convert to representation used by executor
- Expand Paths to Plans
- Example task: renumber Var nodes to meet executor's requirements (Vars in join nodes must be labeled “OUTER” or “INNER”, not with original base relation’s number)
- Mostly boring, except when it breaks

I lied again ...

- Actually, Path-to-Plan conversion happens after scan/join planning, and before query special feature handling
- Other postprocessing does happen at the end
- This is because the query special feature code doesn't use Paths to represent what it's thinking about; it works on actual Plan trees
- If we were to switch over to doing special features with Paths, presumably this would change

A map of backend/optimizer/

Subdirectory	Contents
geqo	GEQO join searching
path	Path generation and cost estimation
plan	Main planning driver code
prep	Preprocessing
util	Miscellaneous

... and don't forget

backend/utils/adt/selfuncs.c	operator-specific selectivity functions

geqo/geqo_copy.c	boring support code
geqo/geqo_cx.c	unused method for generating a mutated tour
geqo/geqo_erx.c	**active** method for generating a mutated tour
geqo/geqo_eval.c	evaluate cost of tour
geqo/geqo_main.c	glue code
geqo/geqo_misc.c	debug printout code
geqo/geqo_mutation.c	randomly mutate a tour (by swapping cities)
geqo/geqo_ox1.c	unused method for generating a mutated tour
geqo/geqo_ox2.c	unused method for generating a mutated tour
geqo/geqo_pmx.c	unused method for generating a mutated tour
geqo/geqo_pool.c	boring support code (manage "pool" of tours)
geqo/geqo_px.c	unused method for generating a mutated tour
geqo/geqo_random.c	boring support code
geqo/geqo_recombination.c	boring support code
geqo/geqo_selection.c	randomly select two "parent" tours from pool
1200 lines	

path/allpaths.c	core scan/join search code (mostly about base rels)
path/clausesel.c	clause selectivity (driver routines mostly)
path/costsize.c	estimate path costs and relation sizes
path/equivclass.c	support code for managing equivalence classes
path/indxpath.c	core path generation for indexscan paths
path/joinpath.c	core path generation for joins
path/joinrels.c	core scan/join search code (mostly about join rels)
path/orindxpath.c	path generation for "OR" indexscans
path/pathkeys.c	support code for managing PathKey data structures
path/tidpath.c	core path generation for TID-scan paths (`WHERE` ctid = constant)

8000 lines

plan/analyzejoins.c	late-stage join preprocessing
plan/createplan.c	build Plan tree from selected Path tree
plan/initsplan.c	scan/join preprocessing (driven by planmain.c)
plan/planagg.c	special hack for planning min/max aggregates
plan/planmain.c	driver for scan/join planning
plan/planner.c	driver for all "extra" query features
plan/setrefs.c	Plan tree postprocessing
plan/subselect.c	handle sub-selects (that aren't in `FROM`)

8500 lines

prep/prepjointree.c	early-stage join preprocessing
prep/prepqual.c	`WHERE` clause (qual) preprocessing
prep/preptlist.c	target list preprocessing (mostly just for `INSERT`/`UPDATE`/`DELETE`)
prep/prepunion.c	plan set operations (`UNION`/`INTERSECT`/`EXCEPT`, but not simple `UNION ALL`); also has some support code for inheritance cases ("appendrels")

3000 lines

util/clauses.c	assorted code for manipulating expression trees, includes constant folding and SQL function inlining
util/joininfo.c	support code for managing lists of join clauses
util/pathnode.c	code for creating various sorts of Path nodes, and for comparing the costs of different Path nodes ⇒ add_path() can be seen as the very heart of the planner
util/placeholder.c	code for managing PlaceHolderVars
util/plancat.c	code for extracting basic info about tables and indexes from the system catalogs (sets up RelOptInfo and IndexOptInfo)
util/predtest.c	code for proving that a `WHERE` clause implies or contradicts another one; used for constraint exclusion and seeing if partial indexes can be used
util/relnode.c	support code for managing RelOptInfo nodes
util/restrictinfo.c	support code for managing RestrictInfo nodes
util/tlist.c	support code for managing target lists
util/var.c	support code for managing Vars

7000 lines

Agenda

- What's the problem we need to solve?
- Structure of the planner
- **Some key data structures**
- Cost estimation
- Future work

PathKeys

- PathKeys are a List structure representing the sort ordering of the output tuples of a Path; for example `ORDER BY x, y` is represented by a list of a PathKey for `x` and a PathKey for `y`
- They can also represent a desired ordering
- “Canonical” pathkeys are used to make pathkey comparison cheap (we can use pointer equality)
- For more info, read `src/backend/optimizer/README`

EquivalenceClasses

- An EquivalenceClass represents a set of values that are known equal as a consequence of clauses like `WHERE x = y AND y = z`
- By transitivity, we can deduce that any two members of an EquivalenceClass are equal
- EquivalenceClasses also represent the value that a PathKey orders by (since if `x` = `y`, then `ORDER BY x` must be the same as `ORDER BY y`)
- Again, see `src/backend/optimizer/README`

Agenda

- What's the problem we need to solve?
- Structure of the planner
- Some key data structures
- **Cost estimation**
- Future work

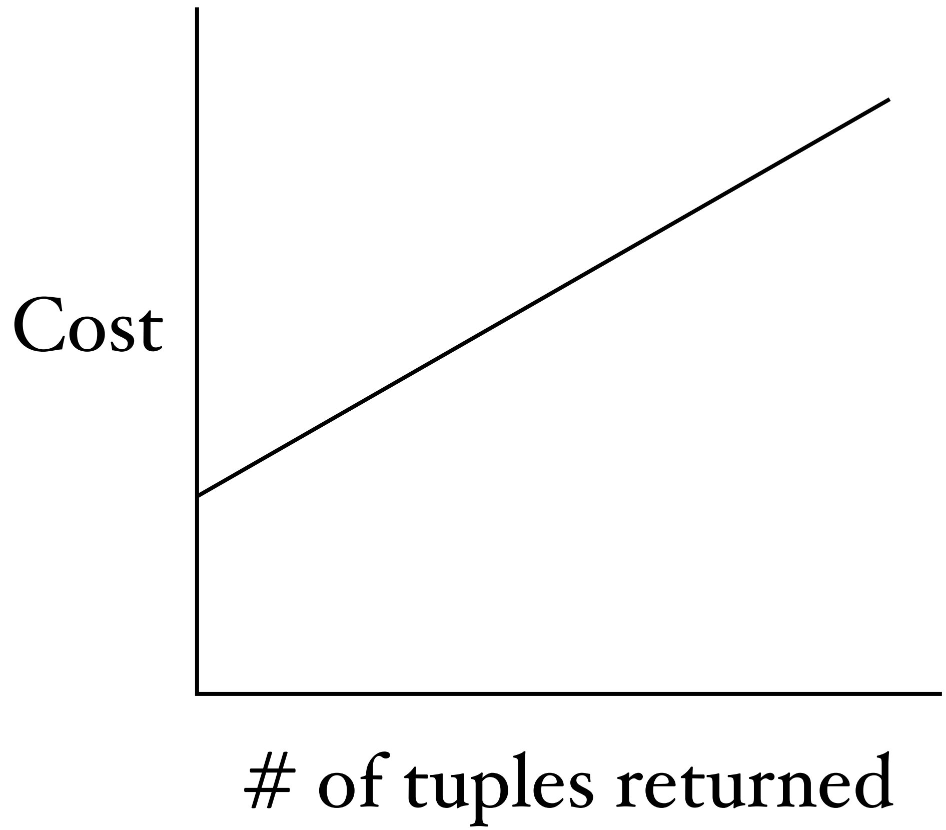
Cost estimation

- Everything I've talked about so far is just mechanism for generating alternatives to consider
- Cost estimation is what really drives the planner's behavior
- If the planner can't generate the plan you want, you need to fix the mechanism
- If it generates and rejects the plan you want, you need to fix the cost estimation

Cost estimation: basic theory

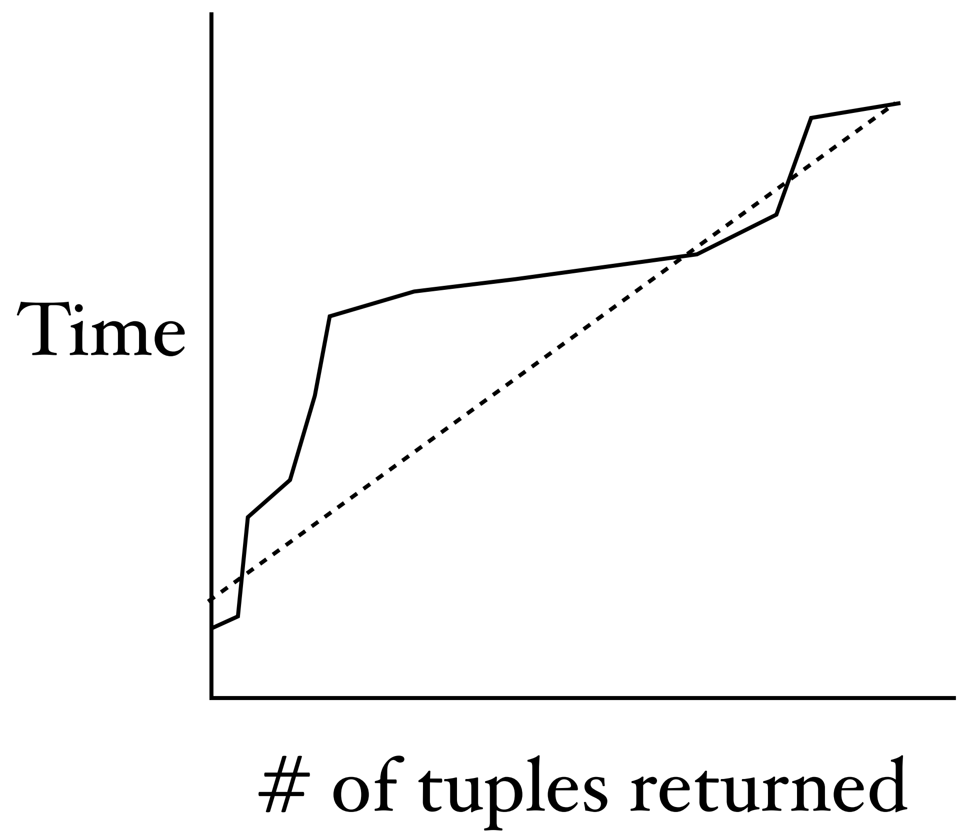
- Cost of a plan node is assumed to increase linearly from startup cost to total cost as more tuples are returned

- With a `LIMIT` we will stop short of paying the total cost, if we fetch fewer than the total number of tuples



Cost estimation: ugly reality

- Sometimes the real world is not so linear
- If we're selecting a subset of scanned tuples, we will skip over some tuples
- Non-uniform distribution of desired tuples results in non-linear runtime
- This can result in seriously bad estimates for small `LIMIT`s



Plan cost models

- For each plan node type, `costsize.c` has a function to estimate its cost in terms of the primitive cost variables (page reads, operator evaluations, etc), given estimates about the numbers of rows, total data size, etc

- These models are a bit simplistic, but for the most part when the planner falls down, it's not a modeling failure but a statistical failure

- “Garbage in, garbage out” applies here!

Result sizes for relation scans

- Number of rows returned by any given table scan is estimated as the raw relation size multiplied by the "selectivity" of relevant `WHERE` conditions

- Raw relation size is taken to be tuple density found by last `ANALYZE` (that is, # live tuples / # blocks) times relation's current size in blocks

- Not perfect, but seems to work pretty well

- Width is just the sum of the per-column average widths estimated by `ANALYZE` for all the variables needed in the query

Result sizes for joins

- Number of rows returned is the Cartesian product size (product of estimated rowcounts of input relations) multiplied by the selectivity of relevant join conditions

- … with some special twiddling for outer joins, for instance a `LEFT JOIN` cannot produce fewer rows than its left input has

- Width is sum of the per-column average widths for variables that are needed above the join

Selectivity is the hard part

- Applicable `WHERE`/`JOIN ON` conditions are broken into "clauses" that are combined with `AND`/`OR`/`NOT`
- Estimate selectivity of each clause separately, then combine results
- We have per-operator and per-special-clause selectivity estimation functions (these are mostly in `utils/adt/selfuncs.c`)
- Combination of per-clause estimates is done in `clausesel.c`
- `AND`/`OR`/`NOT` combinations are easy, **if** the clauses are independent ... but often they are not, and we get a bad result
- `clausesel.c` has some smarts for range restrictions, that is `X >= C1 AND X <= C2`

Operator selectivity functions

- “Restriction” estimators are used for clauses containing Vars of just one relation
- These generally don't try to handle anything more complex than “Var op Constant” (but we have a liberal definition of “Constant”)
- “Join” estimators are used for clauses containing Vars from more than one relation
- These generally don't try to handle anything more complex than “Var op Var”
- Lots of unfinished work here

Aggressive constant-folding

- When trying to reduce a clause sub-expression to a constant for selectivity estimation, we will substitute any available values for parameter symbols, and will evaluate stable as well as immutable functions

- For example,

```
ts_col >= now() - interval ‘1 day’
```

  will be estimated using current time minus one day as the comparison constant

Cost estimation API issues

- The “per operator” selectivity functions mostly are not truly specific to a single operator; rather we use functions like `eqsel` and `scalarltsel`, which embody knowledge about a class of operators
- I have a feeling that this is not the best API, mainly because it doesn’t seem extensible to cover estimation of interrelated clauses. But changing it would break a lot of extension modules ...
- Also, nobody's ever fixed the Berkeley-era omission of selectivity estimators for functions that are accessed directly rather than via operators

Agenda

- What's the problem we need to solve?
- Structure of the planner
- Some key data structures
- Cost estimation
- **Future work**

Parameterized scans

- A nestloop inner indexscan is basically a scan parameterized by values from the current row of the outer relation
- Sometimes it'd be useful to parameterize a larger chunk of the plan than a single base-relation scan (another way to say that is we'd like an indexscan to be able to use a parameter from more than one join level up)
- We need this in situations involving join order restrictions
- Problem #1: avoid explosion in number of paths to consider
- Problem #2: size estimates for inner scans are not independent of what the outer relation is

Foreign data wrappers

- As of 9.1, wrappers are on their own to produce plans and cost estimates for scans of foreign tables
- This obviously isn't good for the long run
- Particularly bad: no support for inner indexscan on a foreign table
- Need to figure out what sorts of functionality FDWs need, then refactor existing planner code to provide that in a reasonably clean fashion

Conclusion

- We really need more people working on the planner, the selectivity functions, etc
- I hope this talk has demystified the planner a bit, and given you some idea of where things can be improved