Daniel Gustafsson • Pivotal

Hello, I'm **Daniel Gustafsson**

Pivotal

I work at

with

and

GREENPLUM DATABASE

PostgreSQL Conference Europ ×

Secure https://2017.pgconf.eu

PGCONF.EU 2017

ABOUT CFP REGISTRATION VENUE &

THE 9TH ANNUAL

POSTGRESQL CONFERENCE EUROPE

```
CREATE TABLE t (
    a integer,
    b integer
);
CREATE TABLE tt (
    c integer,
    d integer
);

INSERT INTO t VALUES (1, 2);
INSERT INTO tt VALUES (2, 2);
```

```
=# SELECT t.a, tt.c FROM t, tt
-# WHERE t.b = tt.d;
 a | c
---+---
 1 | 2
(1 row)
```

```
=# SELECT t.a, tt.c FROM t, tt
-# WHERE t.b = tt.d;
 a | c
---+---
 1 | 2
(1 row)
```

```
=# SELECT t.a, tt.c FROM t, tt
-# WHERE t.b = tt.d;
 a | c
---+---
 1 | 2
(1 row)
```

?

Purpose

Introduce a few select parts of the PostgreSQL query planner and hopefully demystify a little

Purpose

Introduce a few select parts of the PostgreSQL query planner and hopefully demystify a little

..hopefully

```sql
SELECT a, c FROM t, tt WHERE b = d;
```

```
SELECT a, c FROM t, tt WHERE b = d;
```

Parser

```
SELECT a, c FROM t, tt WHERE b = d;
```

Parser

Planner

```
SELECT a, c FROM t, tt WHERE b = d;
```

Parser

Planner

Executor

```
SELECT a, c FROM t, tt WHERE b = d;
```

Parser

Planner

Executor

$

```
SELECT a, c FROM t, tt WHERE b = d;
```

Well defined

```
SELECT a, c FROM t, tt WHERE b = d;
```

Parser

Well defined

Planner

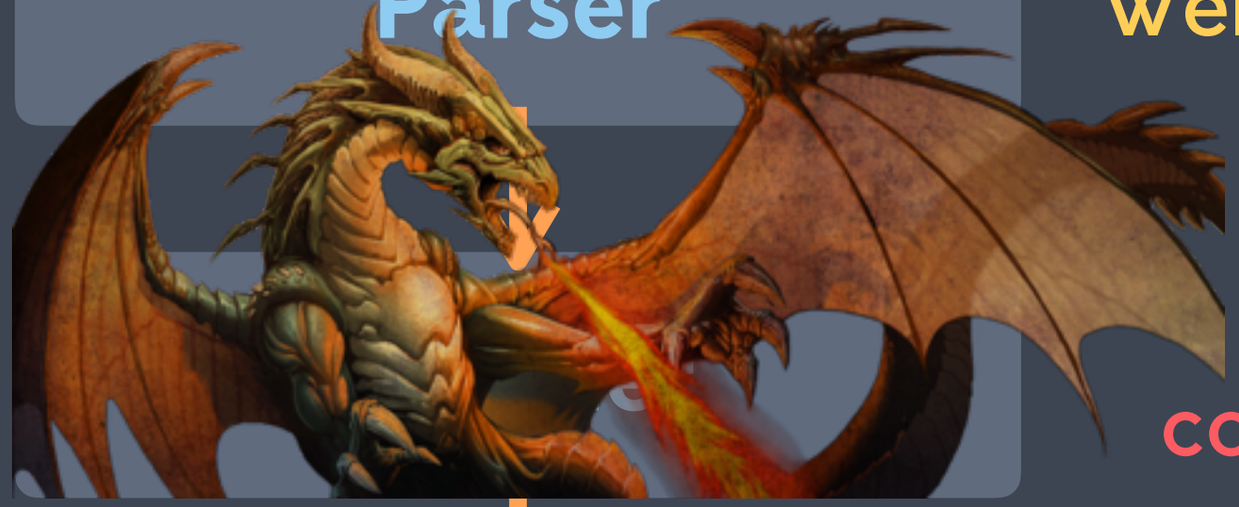
Executor

Well defined

```
SELECT a, c FROM t, tt WHERE b = d;
```

↓

Parser



Well defined

..bloody complicated

↓

Executor

Well defined

↓

$

http://www.freeiconspng.com/img/20230

238 MAGIC TRICKS

REVEALED

NOTHING EXTRA TO BUY!

ONLY 50¢ POSTPAID

EVERY SINGLE TRICK IS PERFORMED WITH EVERYDAY THINGS YOU HAVE AROUND THE HOUSE... COINS, CARDS, BALLS, HANDKERCHIEFS, ROPES, etc.

"POSITIVELY ASTONISHING"...

SAY PEOPLE WHO HAVE SEEN THIS COLLECTION. YOU'LL BE PLUCKING COINS FROM THIN AIR! YOU'LL CAUSE CARDS TO CHANGE THEIR SPOTS AT YOUR COMMAND! YOU'LL HEAR THE GASPS OF WONDER AS YOU DO THE WORLD-FAMOUS "INDIAN ROPE TRICK." YOU'LL ACTUALLY DO OVER **238** BAFFLING TRICKS, INCLUDING:

- THE VANISHING BALL
- THE MIND READING TRICK
- THE SECRET of NUMBER 9
- PHANTOM WRITING
- GROWING MONEY TRICK
- THE COIN LEAPING TRICK
- DISAPPEARING HANDKERCHIEF
- THE KNOT THAT UNTIES ITSELF
- THE DISAPPEARING COIN
- MAKING A BALL ROLL BY ITSELF
- MIRACLE CARD JUMPING TRICK
- THE PHANTOM MONEY TRICK, etc.

ANYONE... 6 TO 60... CAN PERFORM THESE FEATS OF MAGIC... ONCE YOU KNOW THEIR SECRETS!

COMPLETE SECRETS REVEALED!

EVERY SINGLE TRICK FULLY EXPLAINED! YOU SAW SOME OF THEM ON T.V. MANY WERE PERFORMED BY SUCH MASTER MAGICIANS AS HOUDINI, THURSTON, etc. AND NOW... YOU CAN DO ALL OF THESE FAMOUS MAGIC TRICKS. THEY'RE FUN!

MAIL TODAY

MAGIC TRICKS

P.O. BOX 397 - ROCKVILLE CENTRE,

RUSH ME MY MAGIC TRICKS FOR WHICH I HAVE ENCLOSED 50¢.

Send me 3 for only $1.25 (check here) ☐

SATISFACTION GUARANTEED. NO C.O.D.

(PLEASE PRINT INFORMATION BELOW)

Name _____

Address _____

City _____ State _____ Zip _____

CANADIAN & FOREIGN 70¢ EACH INT'L MONEY ORDER

Query Planning

Preprocessing — Simplification, constant folding

↓

Scan/Join Planning — WHERE clause

↓

Special Features — `GROUP BY`, window functions ..

↓

Postprocessing — Convert plan to execution

Query Planning

Preprocessing — Simplification, constant folding

↓

Scan/Join Planning — WHERE clause

↓

Special Features — `GROUP BY`, window functions ..

↓

Postprocessing — Convert plan to execution

Join Order Selection

Join Order Enumeration

Given the set of relations in a query, find the optimal order in which to access the relations in order to satisfy the query

```
A.a = B.b AND
A.d = D.d AND
B.c = C.c
```

```
A.a = B.b AND
A.d = D.d AND
B.c = C.c
```

N! join orderings:

```
ABCD, ABDC, ADBC, DABC ...
```

```
A.a = B.b AND
A.d = D.d AND
B.c = C.c
```

N! join orderings:

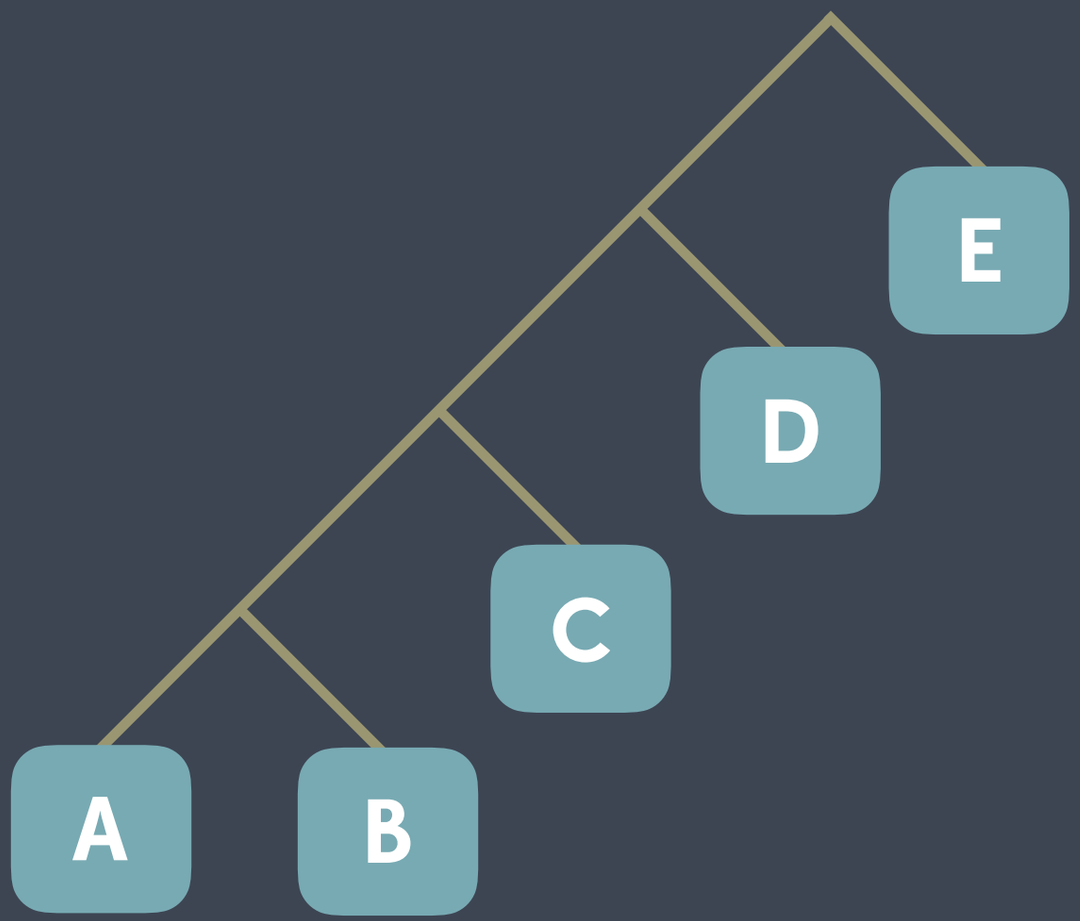
```
ABCD, ABDC, ADBC, DABC ...
```

(N-1)! plans per join order:

```
(((AB)C)D), ((AB)(CD)) ...
```

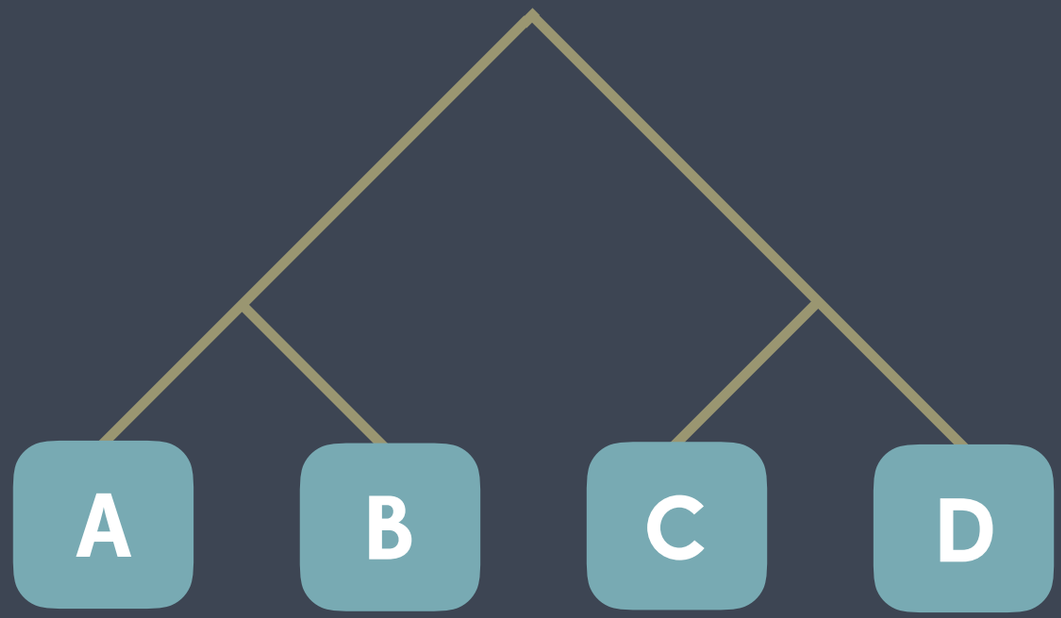
Join Trees

Left Sided



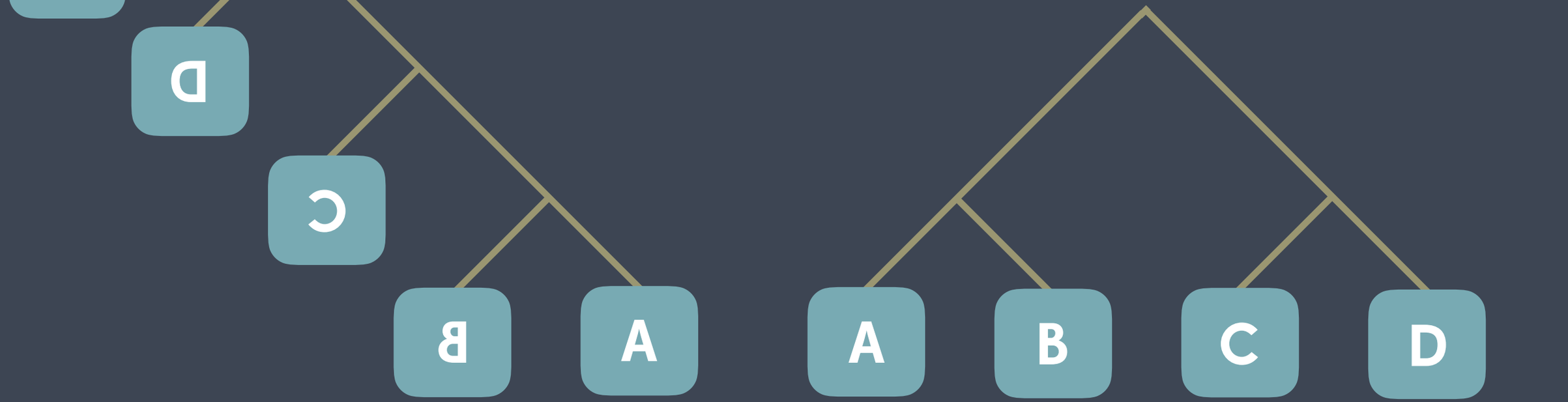
(((((AB)C)D)E)

Bushy



((AB)(CD))

Join Trees



Right Sided

(E(D(C(BA))))

Bushy

((AB)(CD))

N! × (N-1)! possible plans

4 way join → 144 plans

10 way join → 1,316,818,944,000 plans

20 way .. ☠️💣

Naive, and/or, exhaustive approaches doesn't scale

79

https://en.wikipedia.org/wiki/Katson_sineen_taivaan

ENDLICH ANGEKOMMEN

ATARI PERSONAL COMPUTER SYSTEM

ATARI 400 (16 K) und ATARI 800 (bis 48 K) sind das Herz des kompletten Personal Computer Systems. Color-PAL-Signal für jeden Fernseher. 6502 Mikroprozessor. Grafik, Sound und über 160 Farben. Programmiersprachen BASIC, PASCAL, PILOT und ASSEMBLER-EDITOR ROM-Programm-Module. Disk-Drives, Drucker, Programm-Recorder, Interface, Light Pen, Joysticks usw. als geprüftes ATARI-Zubehör. Umfangreiche ATARI-Software-Bibliothek. Lieferung nur über den qualifizierten Fachhandel.

ATARI®

A Warner Communications Company

Fordern Sie jetzt ausführliche Informationen an!

Name: _____

Beruf: _____

Straße: _____

PLZ/Ort: _____

geplanter Einsatzbereich:

☐ Beruf ☐ Hobby ☐ Ausbildung ☐ Unterhaltung

Atari Elektronik Vertriebsgesellschaft mbH
Bebelallee 10 · 2000 Hamburg 60

Computers for people

Machen Sie Ihr Hobby zum Beruf.
Atari sucht „Computer-Freaks" als Mitarbeiter.
Für viele Bereiche.
Schreiben Sie an
Herrn Ollmann.

System/R

https://www.youtube.com/watch?v=bvnWpTJPPUg

Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G. (1979), *"Access Path Selection in a Relational Database Management System"*, Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, pp. 23–34

https://www.youtube.com/watch?v=bvnWpTJPPUg

Selinger Algoritm

Step 1 • Enumerate all access paths to individual relations, keep the cheapest around

Selinger Algoritm

Step 1 • Enumerate all access paths to individual relations, keep the cheapest around

Step 2 • Consider all ways to join two relations, using best access path as computed in step one

Selinger Algoritm

Step 1 • Enumerate all access paths to individual relations, keep the cheapest around

Step 2 • Consider all ways to join two relations, using best access path as computed in step one

Step 3 • Consider all ways to join 3 relations, reusing cached calculations from step 2

Selinger Algoritm

Step 1 • Enumerate all access paths to individual relations, keep the cheapest around

Step 2 • Consider all ways to join two relations, using best access path as computed in step one

Step 3 • Consider all ways to join 3 relations, reusing cached calculations from step 2

• • •

Step n • Consider all ways to join n relations, reusing cached calculations from step n-1

Selinger Algoritm

Step 1 • Enumerate all access paths to individual relations, keep the cheapest around

Cost based

Step 2 • Consider all ways to join two relations, using best access path as computed in step one

Step 3 • Consider all ways to join 3 relations, reusing cached calculations from step 2

• • •

Step n • Consider all ways to join n relations, reusing cached calculations from step n-1

Selinger Algoritm

Step 1 • Enumerate all access paths to individual relations, keep the cheapest around

Cost based

Step 2 • Consider all ways to join two relations, using best access path as computed in step one

Step 3 • Consider all ways to join 3 relations, reusing cached calculations from step 2

• • •

Step n • Consider all ways to join n relations, reusing cached calculations from step n-1

Dynamic Programming

```
A.a = B.b AND
A.d = D.d AND
B.c = C.c
```

Step 1 • Access Paths

```
A = OptimalAccess(A_relation);
B = OptimalAccess(B_relation);
...
```

Step 2 • 2-way Join

```
{A,B} = Cheapest(AB,BA);
{B,C} = Cheapest(BC,CB);
...
```

Step 3 • 3-way Join

```
{A,B,C} = Cheapest(A{B,C}, {B,C}A,
                   B{A,C}, {A,C}B,
                   C{A,B}, {A,B}C);
{A,B,D} = Cheapest(A{B,D}, {B,D}A,
                   B{A,D}, {A,D}B,
                   D{A,B}, {A,B}D);

                ...
```

Step 3 • 3-way Join

```
{A,B,C} = Cheapest(A{B,C}, {B,C}A,
                   B{A,C}, {A,C}B,
                   C{A,B}, {A,B}C);

{A,B,D} = Cheapest(A{B,D}, {B,D}A,
                   B{A,D}, {A,D}B,
                   D{A,B}, {A,B}D);

...
```

Precomputed in step 2

Step n • n-way Join

```
{A,B,C,D} = ...
```

Step n • n-way Join

`{A,B,C,D} = ...`

Cheapest join order for query reached

or

Cheapest join order with the correct ordering iff cheaper than cheapest overall + final sort-step

PostgreSQL

GEQO - Genetic Query Optimizer

geqo_threshold

Relations in query

Selinger Algorithm

GEQO

Heuristics required as the search space increase

Travelling salesman algorithm across the relations

..not terribly good, but currently our best alternative

How Good Are Query Optimizers, Really?

Viktor Leis
TUM
leis@in.tum.de

Andrey Gubichev
TUM
gubichev@in.tum.de

Atanas Mirchev
TUM
mirchev@in.tum.de

Peter Boncz
CWI
p.boncz@cwi.nl

Alfons Kemper
TUM
kemper@in.tum.de

Thomas Neumann
TUM
neumann@in.tum.de

ABSTRACT

Finding a good join order is crucial for query performance. In this paper, we introduce the Join Order Benchmark (JOB) and experimentally revisit the main components in the classic query optimizer architecture using a complex, real-world data set and realistic multi-join queries. We investigate the quality of industrial-strength cardinality estimators and find that all estimators routinely produce large errors. We further show that while estimates are essential for finding a good join order, query performance is unsatisfactory if the query engine relies too heavily on these estimates. Using another set of experiments that measure the impact of the cost model, we find that it has much less influence on query performance than the cardinality estimates. Finally, we investigate plan enumeration techniques comparing exhaustive dynamic programming with heuristic algorithms and find that exhaustive enumeration improves performance despite the sub-optimal cardinality estimates.

1. INTRODUCTION

The problem of finding a good join order is one of the most studied problems in the database field. Figure 1 illustrates the classical, cost-based approach, which dates back to System R [36]. To obtain an efficient query plan, the query optimizer enumerates some subset

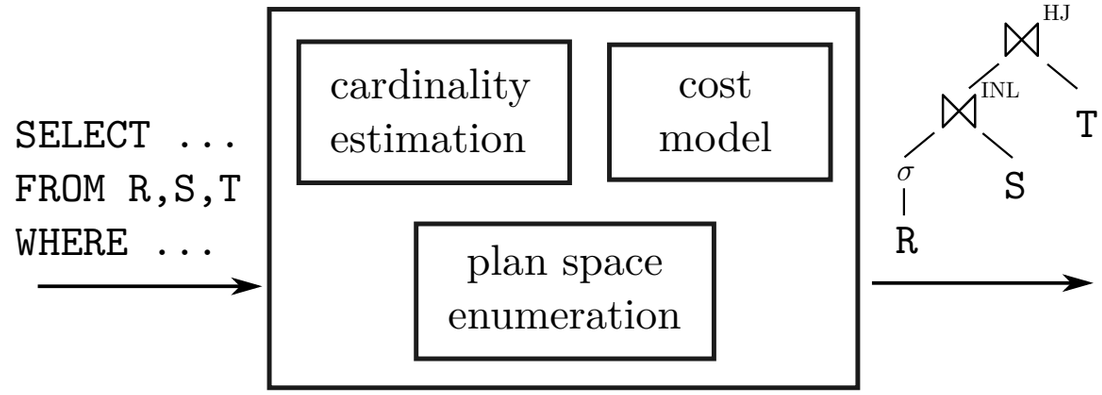


Figure 1: Traditional query optimizer architecture

- How important is an accurate cost model for the overall query optimization process?
- How large does the enumerated plan space need to be?

To answer these questions, we use a novel methodology that allows us to isolate the influence of the individual optimizer components on query performance. Our experiments are conducted using a real-world data set and 113 multi-join queries that provide a challenging, diverse, and realistic workload. Another novel aspect of this paper is that it focuses on the increasingly common main-memory

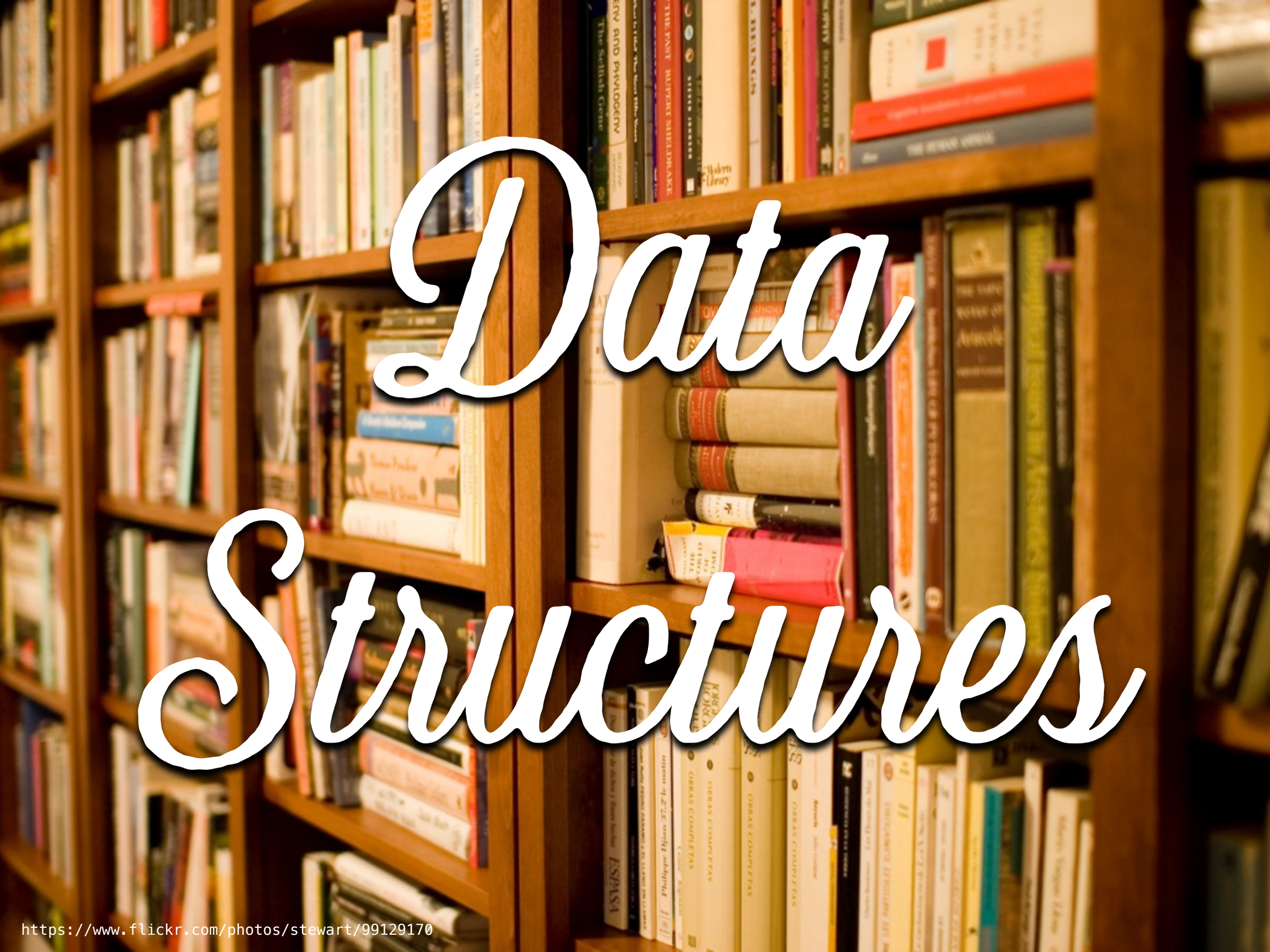
“

In this paper we have **provided quantitative evidence for conventional wisdom** that has been accumulated in three decades of practical experience with query optimizers. We have shown that query optimization is essential for efficient query processing and that **exhaustive enumeration algorithms find better plans than heuristics**.

Implementation

https://www.flickr.com/photos/stewart/99129170

Data Structures



https://www.flickr.com/photos/stewart/99129170

Plan Node

A plan is a tree of Plan nodes

How and What of query execution

Data Source (relation, join)

Targetlist (`SELECT ...`)

Selection Conditions (`WHERE ...`)

Path

A Path is a Plan node omitting the details irrelevant to the planner

RelOptInfo

Created for each base relation and join relation

List of Paths representing how to scan/join the relation (`RelOptInfo.pathlist`)

List of all join clauses involving the relation (`RelOptInfo.joininfo`)

RANGE TABLE lists the relations in the query

```
SELECT a, c FROM t, tt AS t2 WHERE b = d;
```

range table entries

```
SELECT a, c FROM t JOIN tt ON (b = d);
```

```
:rtable (
   {RTE
   :alias <>
   :eref
      {ALIAS
      :aliasname t
      :colnames ("a" "b")
      }
   :rtekind 0
   :relid 16385
   :relkind r
   ...
   }
   {RTE
   :alias
      {ALIAS
      :aliasname t2
      :colnames <>
      }
   :eref
      {ALIAS
      :aliasname t2
      :colnames ("c" "d")
      }
   :rtekind 0
   :relid 16388
   :relkind r
   ...
   }
)
```

```
SELECT a, c
  FROM t, tt AS t2
 WHERE b = d;
```

Controlled by logging GUCS:

```
debug_print_parse
debug_print_rewritten
debug_print_plan
```

```
:rtable (
   {RTE
   :alias <>
   :eref
      {ALIAS
      :aliasname t
      :colnames ("a" "b")
      }
   :rtekind 0
   :relid 16385
   :relkind r
   ...
   }
   {RTE
   :alias
      {ALIAS
      :aliasname t2
      :colnames <>
      }
   :eref
      {ALIAS
      :aliasname t2
      :colnames ("c" "d")
      }
   :rtekind 0
   :relid 16388
   :relkind r
   ...
   }
)
```

```
SELECT a, c
  FROM t, tt AS t2
 WHERE b = d;
```

```
:rtable (
   {RTE
   :alias <>
   :eref
      {ALIAS
      :aliasname t
      :colnames ("a" "b")
      }
   :rtekind 0
   :relid 16385
   :relkind r
   ...
   }
   {RTE
   :alias
      {ALIAS
      :aliasname t2
      :colnames <>
      }
   :eref
      {ALIAS
      :aliasname t2
      :colnames ("c" "d")
      }
   :rtekind 0
   :relid 16388
   :relkind r
   ...
   }
)
```

```
=# SELECT relname FROM pg_class
-# WHERE oid IN (16385, 16388);
 relname
---------
 t
 tt
(2 rows)
```

Code

```c
RelOptInfo *
standard_join_search(PlannerInfo *root, int levels_needed, List *initial_rels)
{
	int			lev;
	RelOptInfo *rel;

	/*
	 * This function cannot be invoked recursively within any one planning
	 * problem, so join_rel_level[] can't be in use already.
	 */
	Assert(root->join_rel_level == NULL);

	/*
	 * We employ a simple "dynamic programming" algorithm: we first find all
	 * ways to build joins of two jointree items, then all ways to build joins
	 * of three items (from two-item joins and single items), then four-item
	 * joins, and so on until we have considered all ways to join all the
	 * items into one rel.
	 *
	 * root->join_rel_level[j] is a list of all the j-item rels.  Initially we
	 * set root->join_rel_level[1] to represent all the single-jointree-item
	 * relations.
	 */
	root->join_rel_level = (List **) palloc0((levels_needed + 1) * sizeof(List *));

	root->join_rel_level[1] = initial_rels;

	for (lev = 2; lev <= levels_needed; lev++)
	{
		ListCell   *lc;

		/*
		 * Determine all possible pairs of relations to be joined at this
		 * level, and build paths for making each one from every available
		 * pair of lower-level relations.
		 */
		join_search_one_level(root, lev);

		/*
		 * Run generate_gather_paths() for each just-processed joinrel.  We
		 * could not do this earlier because both regular and partial paths
		 * can get added to a particular joinrel at multiple times within
		 * join_search_one_level.  After that, we're done creating paths for
		 * the joinrel, so run set_cheapest().
		 */
		foreach(lc, root->join_rel_level[lev])
		{
			rel = (RelOptInfo *) lfirst(lc);

			/* Create GatherPaths for any useful partial paths for rel */
			generate_gather_paths(root, rel);

			/* Find and save the cheapest paths for this rel */
			set_cheapest(rel);

#ifdef OPTIMIZER_DEBUG
			debug_print_rel(root, rel);
#endif
		}
	}

	/*
	 * We should have a single rel at the final level.
	 */
	if (root->join_rel_level[levels_needed] == NIL)
		elog(ERROR, "failed to build any %d-way joins", levels_needed);
	Assert(list_length(root->join_rel_level[levels_needed]) == 1);

	rel = (RelOptInfo *) linitial(root->join_rel_level[levels_needed]);

	root->join_rel_level = NULL;

	return rel;
}
```

```c
RelOptInfo *
standard_join_search(
    PlannerInfo *root,
    int levels_needed,
    List *initial_rels);
```

```
RelOptInfo *
standard_join_search(PlannerInfo *root, int levels_needed, List *initial_rels)
{
	int			lev;
	RelOptInfo *rel;

	/*
	 * This function cannot be invoked recursively within any one planning
	 * problem, so join_rel_level[] can't be in use already.
	 */
	Assert(root->join_rel_level == NULL);

	/*
	 * We employ a simple "dynamic programming" algorithm: we first find all
	 * ways to build joins of two jointree items, then all ways to build joins
	 * of three items (from two-item joins and single items), then four-item
	 * joins, and so on until we have considered all ways to join all the
	 * items into one rel.
	 *
	 * root->join_rel_level[j] is a list of all the j-item rels.  Initially we
	 * set root->join_rel_level[1] to represent all the single-jointree-item
	 * relations.
	 */
	root->join_rel_level = (List **) palloc0((levels_needed + 1) * sizeof(List *));

	root->join_rel_level[1] = initial_rels;

	for (lev = 2; lev <= levels_needed; lev++)
	{
		ListCell   *lc;

		/*
		 * Determine all possible pairs of relations to be joined at this
		 * level, and build paths for making each one from every available
		 * pair of lower-level relations.
		 */
		join_search_one_level(root, lev);

		/*
		 * Run generate_gather_paths() for each just-processed joinrel.  We
		 * could not do this earlier because both regular and partial paths
		 * can get added to a particular joinrel at multiple times within
		 * join_search_one_level.
		 *
		 * After that, we're done creating paths for the joinrel, so run
		 * set_cheapest().
		 */
		foreach(lc, root->join_rel_level[lev])
		{
			rel = (RelOptInfo *) lfirst(lc);

			/* Create GatherPaths for any useful partial paths for rel */
			generate_gather_paths(root, rel);

			/* Find and save the cheapest paths for this rel */
			set_cheapest(rel);

#ifdef OPTIMIZER_DEBUG
			debug_print_rel(root, rel);
#endif
		}
	}

	/*
	 * We should have a single rel at the final level.
	 */
	if (root->join_rel_level[levels_needed] == NIL)
		elog(ERROR, "failed to build any %d-way joins", levels_needed);
	Assert(list_length(root->join_rel_level[levels_needed]) == 1);

	rel = (RelOptInfo *) linitial(root->join_rel_level[levels_needed]);

	root->join_rel_level = NULL;

	return rel;
}
```


```
RelOptInfo *
standard_join_search(
    PlannerInfo *root,
    int levels_needed,
    List *initial_rels);
```

```
/*
 * We employ a simple "dynamic programming" algorithm: we
 * first find all ways to build joins of two jointree
 * items, then all ways to build joins of three items
 * (from two-item joins and single items), then four-item
 * joins, and so on until we have considered all ways to
 * join all the items into one rel.
 */
for (lev = 2; lev <= levels_needed; lev++)
```

```
RelOptInfo *
standard_join_search(PlannerInfo *root, int levels_needed, List *initial_rels)
{
	int			lev;
	RelOptInfo *rel;

	/*
	 * This function cannot be invoked recursively within any one planning
	 * problem, so join_rel_level[] can't be in use already.
	 */
	Assert(root->join_rel_level == NULL);

	/*
	 * We employ a simple "dynamic programming" algorithm: we first find all
	 * ways to build joins of two jointree items, then all ways to build joins
	 * of three items (from two-item joins and single items), then four-item
	 * joins, and so on until we have considered all ways to join all the
	 * items into one rel.
	 *
	 * root->join_rel_level[j] is a list of all the j-item rels.  Initially we
	 * set root->join_rel_level[1] to represent all the single-jointree-item
	 * relations.
	 */
	root->join_rel_level = (List **) palloc0((levels_needed + 1) * sizeof(List *));

	root->join_rel_level[1] = initial_rels;

	for (lev = 2; lev <= levels_needed; lev++)
	{
		ListCell   *lc;

		/*
		 * Determine all possible pairs of relations to be joined at this
		 * level, and build paths for making each one from every available
		 * pair of lower-level relations.
		 */
		join_search_one_level(root, lev);

		/*
		 * Run generate_gather_paths() for each just-processed joinrel.  We
		 * could not do this earlier because both regular and partial paths
		 * can get added to a particular joinrel at multiple times within
		 * join_search_one_level.
		 *
		 * After that, we're done creating paths for the joinrel, so run
		 * set_cheapest().
		 */
		foreach(lc, root->join_rel_level[lev])
		{
			rel = (RelOptInfo *) lfirst(lc);

			/* Create GatherPaths for any useful partial paths for rel */
			generate_gather_paths(root, rel);

			/* Find and save the cheapest paths for this rel */
			set_cheapest(rel);

#ifdef OPTIMIZER_DEBUG
			debug_print_rel(root, rel);
#endif
		}
	}

	/*
	 * We should have a single rel at the final level.
	 */
	if (root->join_rel_level[levels_needed] == NIL)
		elog(ERROR, "failed to build any %d-way joins", levels_needed);
	Assert(list_length(root->join_rel_level[levels_needed]) == 1);

	rel = (RelOptInfo *) linitial(root->join_rel_level[levels_needed]);

	root->join_rel_level = NULL;

	return rel;
}
```


```
RelOptInfo *
standard_join_search(
    PlannerInfo *root,
    int levels_needed,
    List *initial_rels);
```

```
/*
 * Determine all possible pairs of relations to be joined at this
 * level, and build paths for making each one from every available
 * pair of lower-level relations.
 */
join_search_one_level(root, lev);
```

```
RelOptInfo *
standard_join_search(PlannerInfo *root, int levels_needed, List *initial_rels)
{
	int			lev;
	RelOptInfo *rel;

	/*
	 * This function cannot be invoked recursively within any one planning
	 * problem, so join_rel_level[] can't be in use already.
	 */
	Assert(root->join_rel_level == NULL);

	/*
	 * We employ a simple "dynamic programming" algorithm: we first find all
	 * ways to build joins of two jointree items, then all ways to build joins
	 * of three items (from two-item joins and single items), then four-item
	 * joins, and so on until we have considered all ways to join all the
	 * items into one rel.
	 *
	 * root->join_rel_level[j] is a list of all the j-item rels.  Initially we
	 * set root->join_rel_level[1] to represent all the single-jointree-item
	 * relations.
	 */
	root->join_rel_level = (List **) palloc0((levels_needed + 1) * sizeof(List *));

	root->join_rel_level[1] = initial_rels;

	for (lev = 2; lev <= levels_needed; lev++)
	{
		ListCell   *lc;

		/*
		 * Determine all possible pairs of relations to be joined at this
		 * level, and build paths for making each one from every available
		 * pair of lower-level relations.
		 */
		join_search_one_level(root, lev);

		/*
		 * Run generate_gather_paths() for each just-processed joinrel.  We
		 * could not do this earlier because both regular and partial paths
		 * can get added to a particular joinrel at multiple times within
		 * join_search_one_level.  After that, we're done creating paths for
		 * the joinrel, so run set_cheapest().
		 */
		foreach(lc, root->join_rel_level[lev])
		{
			rel = (RelOptInfo *) lfirst(lc);

			/* Create GatherPaths for any useful partial paths for rel */
			generate_gather_paths(root, rel);

			/* Find and save the cheapest paths for this rel */
			set_cheapest(rel);

#ifdef OPTIMIZER_DEBUG
			debug_print_rel(root, rel);
#endif
		}
	}

	/*
	 * We should have a single rel at the final level.
	 */
	if (root->join_rel_level[levels_needed] == NIL)
		elog(ERROR, "failed to build any %d-way joins", levels_needed);
	Assert(list_length(root->join_rel_level[levels_needed]) == 1);

	rel = (RelOptInfo *) linitial(root->join_rel_level[levels_needed]);

	root->join_rel_level = NULL;

	return rel;
}
```


```
RelOptInfo *
standard_join_search(
    PlannerInfo *root,
    int levels_needed,
    List *initial_rels);
```

```
/* Find and save the cheapest paths for this rel */
set_cheapest(rel);
```

```
void
join_search_one_level(PlannerInfo *root, int level)
{
        List      **joinrels = root->join_rel_level;
        ListCell   *r;
        int                     k;

        Assert(joinrels[level] == NIL);

        /* Set join_cur_level so that new joinrels are added to proper list */
        root->join_cur_level = level;

        foreach(r, joinrels[level - 1])
        {
                RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                if (old_rel->joininfo != NIL || old_rel->has_eclass_joins ||
                        has_join_restriction(root, old_rel))
                {
                        ListCell   *other_rels;

                        if (level == 2)          /* consider remaining initial rels */
                                other_rels = lnext(r);
                        else                                    /* consider all initial rels */
                                other_rels = list_head(joinrels[1]);

                        make_rels_by_clause_joins(root,
                                                  old_rel,
                                                  other_rels);
                }
                else
                {
                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }
        }

        for (k = 2;; k++)
        {
                int                     other_level = level - k;

                if (k > other_level)
                        break;

                foreach(r, joinrels[k])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
                        ListCell   *other_rels;
                        ListCell   *r2;

                        if (old_rel->joininfo == NIL && !old_rel->has_eclass_joins &&
                                !has_join_restriction(root, old_rel))
                                continue;

                        if (k == other_level)
                                other_rels = lnext(r);    /* only consider remaining rels */
                        else
                                other_rels = list_head(joinrels[other_level]);

                        for_each_cell(r2, other_rels)
                        {
                                RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);

                                if (!bms_overlap(old_rel->relids, new_rel->relids))
                                {
                                        if (have_relevant_joinclause(root, old_rel, new_rel) ||
                                                have_join_order_restriction(root, old_rel, new_rel))
                                        {
                                                (void) make_join_rel(root, old_rel, new_rel);
                                        }
                                }
                        }
                }
        }
        if (joinrels[level] == NIL)
        {
                foreach(r, joinrels[level - 1])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }

                if (joinrels[level] == NIL &&
                        root->join_info_list == NIL &&
                        !root->hasLateralRTEs)
                        elog(ERROR, "failed to build any %d-way joins", level);
        }
}
```


void
join_search_one_level(PlannerInfo *root,
                      int level)

```
void
join_search_one_level(PlannerInfo *root, int level)
{
	List	  **joinrels = root->join_rel_level;
	ListCell   *r;
	int			k;

	Assert(joinrels[level] == NIL);

	/* Set join_cur_level so that new joinrels are added to proper list */
	root->join_cur_level = level;

	foreach(r, joinrels[level - 1])
	{
		RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

		if (old_rel->joininfo != NIL || old_rel->has_eclass_joins ||
			has_join_restriction(root, old_rel))
		{
			ListCell   *r2;

			if (level == 2)		/* consider remaining initial rels */
			{
				other_rels = lnext(r);
				other_level = 1;
			}
			else
			{
				other_rels = list_head(joinrels[1]);
				other_level = level - 1;
			}

			for_each_cell(r2, other_rels)
			{
				RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);

				if (!bms_overlap(old_rel->relids, new_rel->relids))
				{
					if (have_relevant_joinclause(root, old_rel, new_rel) ||
						have_join_order_restriction(root, old_rel, new_rel))
					{
						(void) make_join_rel(root, old_rel, new_rel);
					}
				}
			}
		}
	}

	if (joinrels[level] == NIL)
	{
		foreach(r, joinrels[level - 1])
		{
			RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

			make_rels_by_clauseless_joins(root,
										  old_rel,
										  list_head(joinrels[1]));
		}

		if (joinrels[level] == NIL &&
			root->join_info_list == NIL &&
			!root->hasLateralRTEs)
			elog(ERROR, "failed to build any %d-way joins", level);
	}
}
```

```
void
join_search_one_level(PlannerInfo *root,
                      int level)
```

```
/*
 * join_search_one_level
 *	  Consider ways to produce join relations containing
 *	  exactly 'level' jointree items.  (This is one step of
 *	  the dynamic-programming method embodied in
 *	  standard_join_search.)  Join rel nodes for each
 *	  feasible combination of lower-level rels are created
 *	  and returned in a list. Implementation paths are
 * created for each such joinrel, too.
 *
 * level: level of rels we want to make this time
 * root->join_rel_level[j], 1 <= j < level, is a list of
 * rels containing j items
 *
 * The result is returned in root->join_rel_level[level].
 */
```

```
void
join_search_one_level(PlannerInfo *root, int level)
{
        List      **joinrels = root->join_rel_level;
        ListCell   *r;
        int                     k;

        Assert(joinrels[level] == NIL);

        /* Set join_cur_level so that new joinrels are added to proper list */
        root->join_cur_level = level;

        foreach(r, joinrels[level - 1])
        {
                RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                if (old_rel->joininfo != NIL || old_rel->has_eclass_joins ||
                        has_join_restriction(root, old_rel))
                {
                        ListCell   *other_rels;

                        if (level == 2)                 /* consider remaining initial rels */
                                other_rels = lnext(r);
                        else                            /* consider all initial rels */
                                other_rels = list_head(joinrels[1]);

                        make_rels_by_clause_joins(root,
                                                  old_rel,
                                                  other_rels);
                }
                else
                {
                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }
        }

        for (k = 2;; k++)
        {
                int                     other_level = level - k;

                if (k > other_level)
                        break;                  /* processed all combinations */

                foreach(r, joinrels[k])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
                        ListCell   *other_rels;
                        ListCell   *r2;

                        if (old_rel->joininfo == NIL && !old_rel->has_eclass_joins &&
                                !has_join_restriction(root, old_rel))
                                continue;

                        if (k == other_level)
                                other_rels = lnext(r);  /* only consider remaining rels */
                        else
                                other_rels = list_head(joinrels[other_level]);

                        for_each_cell(r2, other_rels)
                        {
                                RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);

                                if (!bms_overlap(old_rel->relids, new_rel->relids))
                                {
                                        if (have_relevant_joinclause(root, old_rel, new_rel) ||
                                                have_join_order_restriction(root, old_rel, new_rel))
                                        {
                                                (void) make_join_rel(root, old_rel, new_rel);
                                        }
                                }
                        }
                }
        }
        if (joinrels[level] == NIL)
        {
                foreach(r, joinrels[level - 1])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }

                if (joinrels[level] == NIL &&
                        root->join_info_list == NIL &&
                        !root->hasLateralRTEs)
                        elog(ERROR, "failed to build any %d-way joins", level);
        }
}
```

```
void
join_search_one_level(PlannerInfo *root,
                      int level)
```

```
/*
 * First, consider left-sided and right-sided plans,
 * in which rels of exactly level-1 member relations
 * are joined against initial relations. We prefer to
 * join using join clauses, but if we find a rel of
 * level-1 members that has no join clauses, we will
 * generate Cartesian-product joins against all initial
 * rels not already contained in it.
 */
foreach(r, joinrels[level - 1])
```

```
void
join_search_one_level(PlannerInfo *root, int level)
{
        List      **joinrels = root->join_rel_level;
        ListCell   *r;
        int                     k;

        Assert(joinrels[level] == NIL);

        /* Set join_cur_level so that new joinrels are added to proper list */
        root->join_cur_level = level;

        foreach(r, joinrels[level - 1])
        {
                RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                if (old_rel->joininfo != NIL || old_rel->has_eclass_joins ||
                        has_join_restriction(root, old_rel))
                {
                        ListCell   *other_rels;

                        if (level == 2)                 /* consider remaining initial rels */
                                other_rels = lnext(r);
                        else                            /* consider all initial rels */
                                other_rels = list_head(joinrels[1]);

                        make_rels_by_clause_joins(root,
                                                  old_rel,
                                                  other_rels);
                }
                else
                {
                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }
        }

        for (k = 2;; k++)
        {
                int                     other_level = level - k;

                if (k > other_level)
                        break;                  /* done */

                foreach(r, joinrels[k])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
                        ListCell   *other_rels;
                        ListCell   *r2;

                        if (old_rel->joininfo == NIL && !old_rel->has_eclass_joins &&
                                !has_join_restriction(root, old_rel))
                                continue;

                        if (k == other_level)
                                other_rels = lnext(r);  /* only consider remaining rels */
                        else
                                other_rels = list_head(joinrels[other_level]);

                        for_each_cell(r2, other_rels)
                        {
                                RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);

                                if (!bms_overlap(old_rel->relids, new_rel->relids))
                                {
                                        if (have_relevant_joinclause(root, old_rel, new_rel) ||
                                                have_join_order_restriction(root, old_rel, new_rel))
                                        {
                                                (void) make_join_rel(root, old_rel, new_rel);
                                        }
                                }
                        }
                }
        }

        if (joinrels[level] == NIL)
        {
                foreach(r, joinrels[level - 1])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }

                if (joinrels[level] == NIL &&
                        root->join_info_list == NIL &&
                        !root->hasLateralRTEs)
                        elog(ERROR, "failed to build any %d-way joins", level);
        }
}
```

```
void
join_search_one_level(PlannerInfo *root,
                      int level)
```

```
/*
 * First, consider left-sided and right-sided plans,
 * in which rels of exactly level-1 member relations
 * are joined against initial relations. We prefer to
 * join using join clauses, but if we find a rel of
 * level-1 members that has no join clauses, we will
 * generate Cartesian-product joins against all initial
 * rels not already contained in it.
 */
foreach(r, joinrels[level - 1])
```

A{B,C}, {B,C}A, B{A,C}, {A,C}B ..

```c
void
join_search_one_level(PlannerInfo *root,
                      int level)
```

```c
/*
 * Now, consider "bushy plans" in which relations of k
 * initial rels are joined to relations of level-k
 * initial rels, for 2 <= k <= level-2.
 *
 * We only consider bushy-plan joins for pairs of rels
 * where there is a suitable join clause (or join order
 * restriction), in order to avoid unreasonable growth
 * of planning time.
 */
for (k = 2;; k++)
```

```
void
join_search_one_level(PlannerInfo *root, int level)
{
        List      **joinrels = root->join_rel_level;
        ListCell   *r;
        int                     k;

        Assert(joinrels[level] == NIL);

        /* Set join_cur_level so that new joinrels are added to proper list */
        root->join_cur_level = level;

        foreach(r, joinrels[level - 1])
        {
                RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                if (old_rel->joininfo != NIL || old_rel->has_eclass_joins ||
                        has_join_restriction(root, old_rel))
                {
                        ListCell   *other_rels;

                        if (level == 2)          /* consider remaining initial rels */
                                other_rels = lnext(r);
                        else                                    /* consider all initial rels */
                                other_rels = list_head(joinrels[1]);

                        make_rels_by_clause_joins(root,
                                                  old_rel,
                                                  other_rels);
                }
                else
                {
                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }
        }

        for (k = 2;; k++)
        {
                int                     other_level = level - k;

                if (k > other_level)
                        break;

                foreach(r, joinrels[k])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
                        ListCell   *other_rels;
                        ListCell   *r2;

                        if (old_rel->joininfo == NIL && !old_rel->has_eclass_joins &&
                                !has_join_restriction(root, old_rel))
                                continue;

                        if (k == other_level)
                                other_rels = lnext(r);  /* only consider remaining rels */
                        else
                                other_rels = list_head(joinrels[other_level]);

                        for_each_cell(r2, other_rels)
                        {
                                RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);

                                if (!bms_overlap(old_rel->relids, new_rel->relids))
                                {
                                        if (have_relevant_joinclause(root, old_rel, new_rel) ||
                                                have_join_restriction(root, old_rel, new_rel))
                                        {
                                                (void) make_join_rel(root, old_rel, new_rel);
                                        }
                                }
                        }
                }

                /*
                 * Now, consider "bushy plans" in which relations of k
                 * initial rels are joined to relations of level-k
                 * initial rels, for 2 <= k <= level-2.
                 *
                 * We only consider bushy-plan joins for pairs of rels
                 * where there is a suitable join clause (or join order
                 * restriction), in order to avoid unreasonable growth
                 * of planning time.
                 */
                if (joinrels[level] == NIL &&
                        root->join_info_list == NIL &&
                        !root->hasLateralRTEs)
                        elog(ERROR, "failed to build any %d-way joins", level);
        }
}
```


void
join_search_one_level(PlannerInfo *root,
int level)

{A,B}{B,C}, {A,C}{B,C}

for (k = 2;; k++)

/*
 * Now, consider "bushy plans" in which relations of k
 * initial rels are joined to relations of level-k
 * initial rels, for 2 <= k <= level-2.
 *
 * We only consider bushy-plan joins for pairs of rels
 * where there is a suitable join clause (or join order
 * restriction), in order to avoid unreasonable growth
 * of planning time.
 */
for (k = 2;; k++)

```
void
join_search_one_level(PlannerInfo *root, int level)
{
        List      **joinrels = root->join_rel_level;
        ListCell   *r;
        int                     k;

        Assert(joinrels[level] == NIL);

        /* Set join_cur_level so that new joinrels are added to proper list */
        root->join_cur_level = level;

        foreach(r, joinrels[level - 1])
        {
                RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
                if (old_rel->joininfo != NIL || old_rel->has_eclass_joins ||
                    has_join_restriction(root, old_rel))
                {
                        ListCell   *other_rels;

                        if (level == 2)
                                other_rels = lnext(r);
                        else
                                other_rels = list_head(joinrels[1]);

                        make_rels_by_clause_joins(root,
                                                  old_rel,
                                                  other_rels);
                }
                else
                {
                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }
        }

        for (k = 2;; k++)
        {
                int                     other_level = level - k;

                if (k > other_level)
                        break;

                foreach(r, joinrels[k])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
                        ListCell   *other_rels;
                        ListCell   *r2;

                        if (old_rel->joininfo == NIL && !old_rel->has_eclass_joins &&
                                !has_join_restriction(root, old_rel))
                                continue;

                        if (k == other_level)
                                other_rels = lnext(r);    /* only consider remaining rels */
                        else
                                other_rels = list_head(joinrels[other_level]);

                        for_each_cell(r2, other_rels)
                        {
                                RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);

                                if (!bms_overlap(old_rel->relids, new_rel->relids))
                                {
                                        if (have_relevant_joinclause(root, old_rel, new_rel) ||
                                                have_join_order_restriction(root, old_rel, new_rel))
                                        {
                                                (void) make_join_rel(root, old_rel, new_rel);
                                        }
                                }
                        }
                }
        }
        if (joinrels[level] == NIL)
        {
                foreach(r, joinrels[level - 1])
                {
                        RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

                        make_rels_by_clauseless_joins(root,
                                                      old_rel,
                                                      list_head(joinrels[1]));
                }

                if (joinrels[level] == NIL &&
                        root->join_info_list == NIL &&
                        !root->hasLateralRTEs)
                        elog(ERROR, "failed to build any %d-way joins", level);
        }
}
```

```
/*
 * Since make_join_rel(x, y) handles both x,y and y,x
 * cases, we only need to go as far as the halfway point.
 */
if (k > other_level)
    break;
```

Hedge

https://www.flickr.com/photos/chris77621/8080453815

Summary

The Selinger Algorithm has stood the test of time pretty well

Future work is needed to keep up with increased complexity and BIG DATA

The PostgreSQL codebase is of unrivalled quality

Thanks

daniel@yesql.se • @d_gustafsson

https://github.com/danielgustafsson/presentations/tree/master/joins