

*Through the
Joining Glass*

Daniel Gustafsson • Pivotal

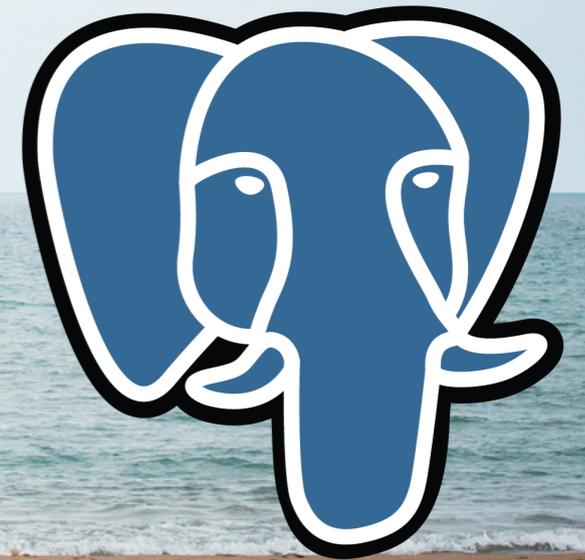
Hello, I'm Daniel Gustafsson

Pivotal

I work at ● with ● and ●



**GREENPLUM
DATABASE**



Hello, I'm Daniel Gustafsson

daniel@yesql.se • @d_gustafsson

Contributor

PGConf.EU, Nordic PGDay, FOSDEM PGDay
{Stockholm | Oslo} PostgreSQL Usergroup





unboxing

All

Images

Videos

News

About 102 000 000 results (0,50 seconds)



Postgres95



t



tt

tt



t

Joins are for lazy people?


164

I recently had a discussion with another developer who claimed to me that JOINS (SQL) are useless. This is technically true but he added that using joins is less efficient than making several requests and link tables in the code (C# or Java).



26

For him joins are for lazy people that don't care about performance. Is this true? Should we avoid using joins?

[c#](#) [java](#) [sql](#) [join](#)

[share](#) [improve this question](#)

OLD MAN YELLS AT CLOUD



Ἰδοὺ ἡμεῖς οὐρανὸν ἵκεν
καὶ ὁ οὐρανὸς ἵκεν ἡμᾶς
καὶ ἡμεῖς οὐρανὸν ἵκεν
καὶ ὁ οὐρανὸς ἵκεν ἡμᾶς
καὶ ἡμεῖς οὐρανὸν ἵκεν
καὶ ὁ οὐρανὸς ἵκεν ἡμᾶς

Ἰδοὺ ἡμεῖς οὐρανὸν ἵκεν
καὶ ὁ οὐρανὸς ἵκεν ἡμᾶς

Ἰδοὺ ἡμεῖς οὐρανὸν ἵκεν
καὶ ὁ οὐρανὸς ἵκεν ἡμᾶς

**Introduce a really elegant,
yet sort of taken-for-granted,
part of the query planner and
trace its origins**

```
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;
```

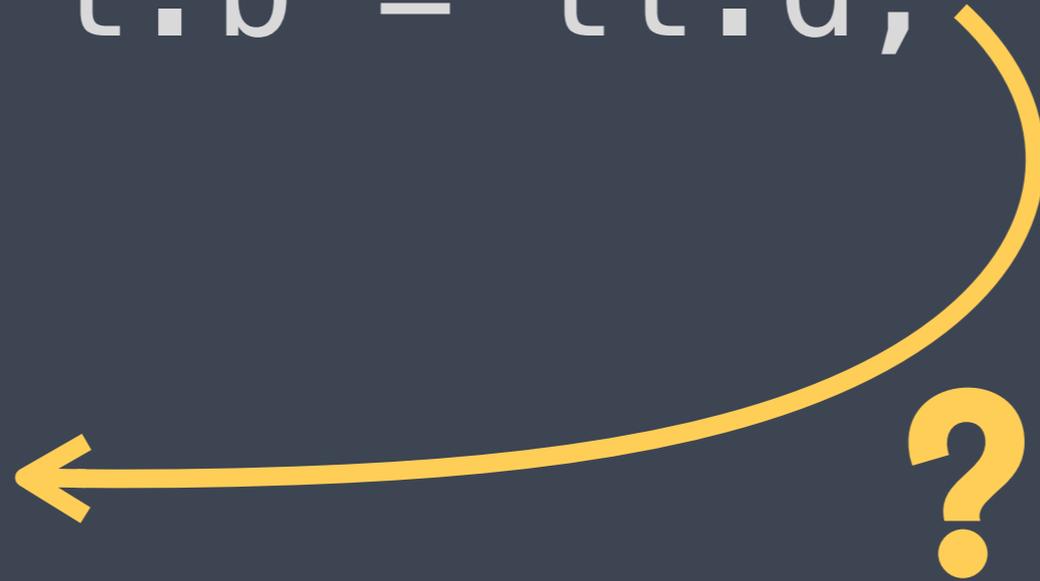
a	c
1	2

(1 row)

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

a	c
1	2

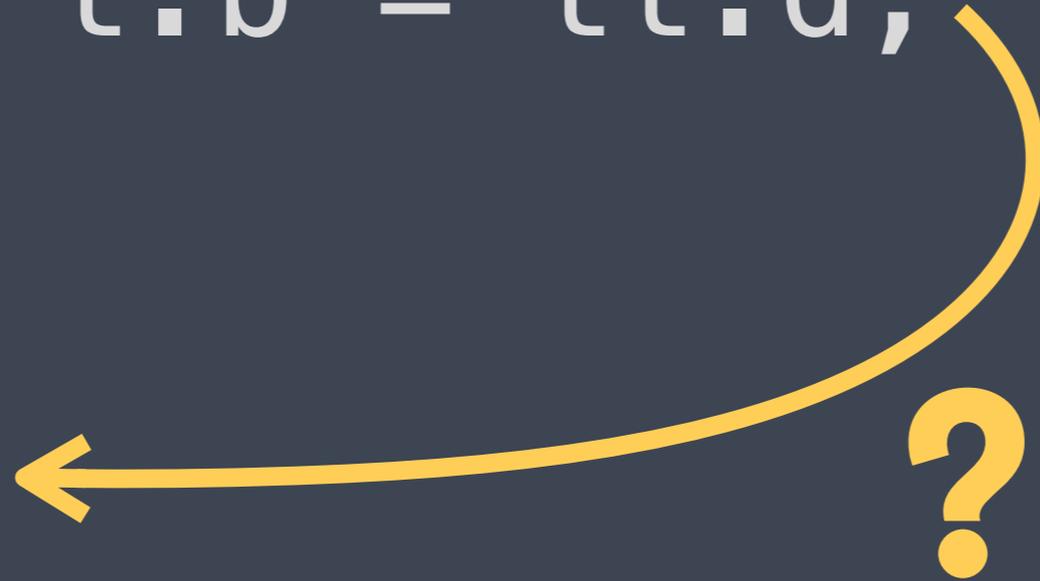
(1 row)



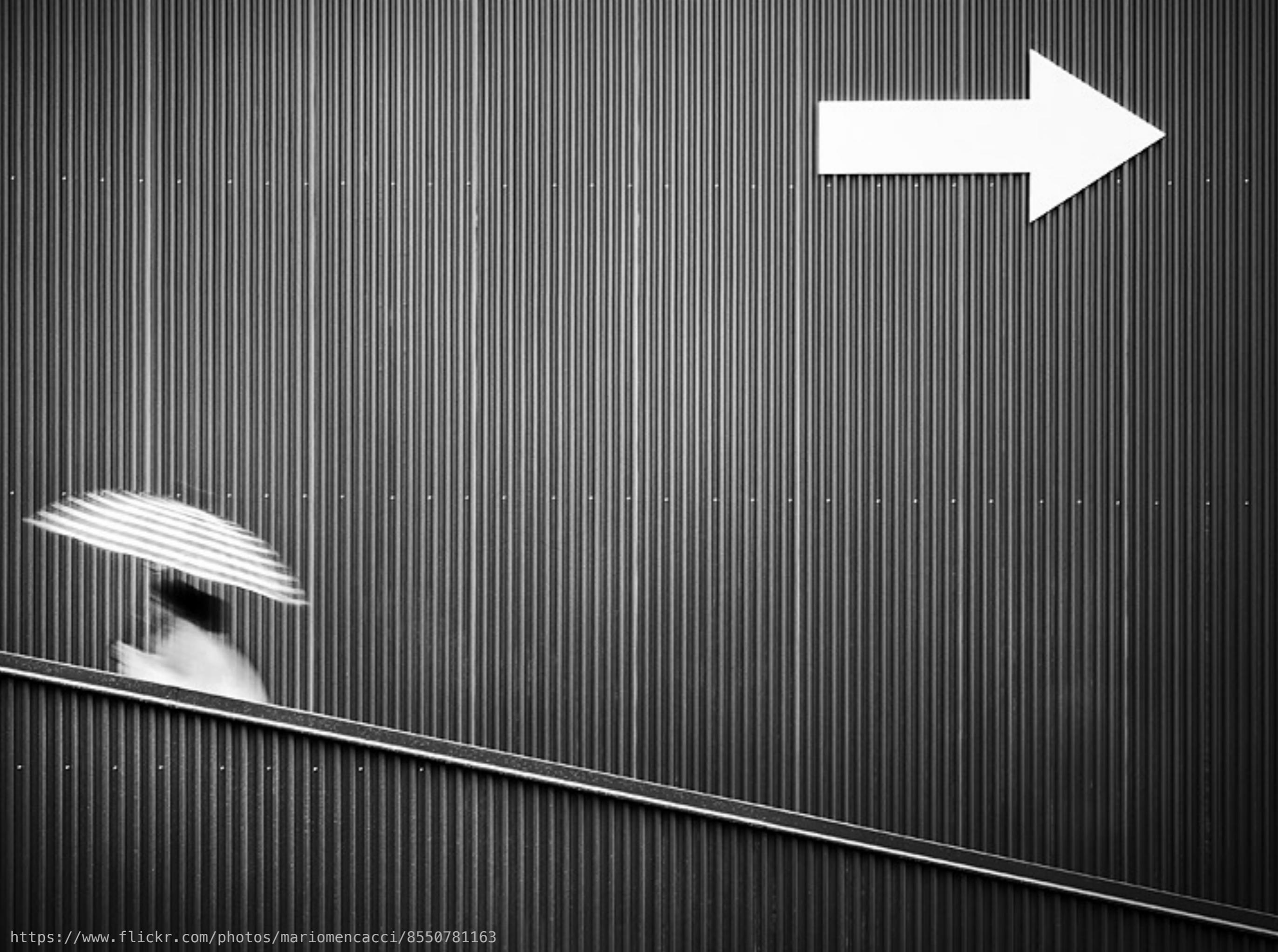
```
=# SELECT t.a, tt.c FROM t, tt
-# WHERE t.b = tt.d;
```

a	c
1	2

(1 row)



```
SELECT
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
FROM
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND l_suppkey = s_suppkey
    AND c_nationkey = s_nationkey
    AND s_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'ASIA'
    AND o_orderdate >= date '1994-01-01'
    AND o_orderdate < date '1994-01-01' + interval '1' year
GROUP BY
    n_name
ORDER BY
    revenue desc;
```



```
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;
```

Parser

Well defined

Planner

Executor

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

Parser

Well defined

Planner

Executor

Pretty well
defined

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



Parser

Well defined



Executor

Pretty well defined

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



Parser

Well defined



..rather complicated



Executor

Pretty well defined

238 MAGIC TRICKS

REVEALED

NOTHING EXTRA TO BUY!

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

ONLY
50¢
POSTPAID

"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)

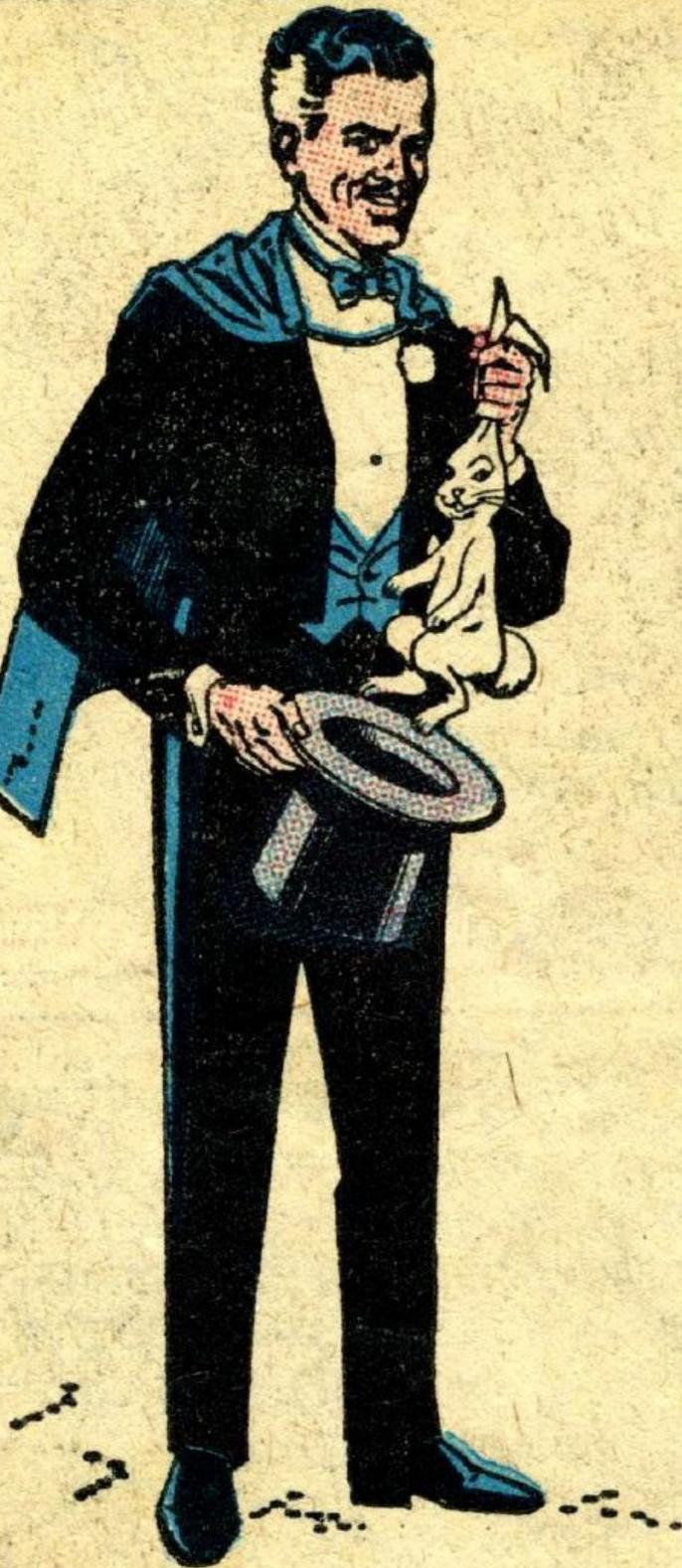
SATISFACTION GUARANTEED. NO C.O.D. (PLEASE PRINT INFORMATION BELOW)

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

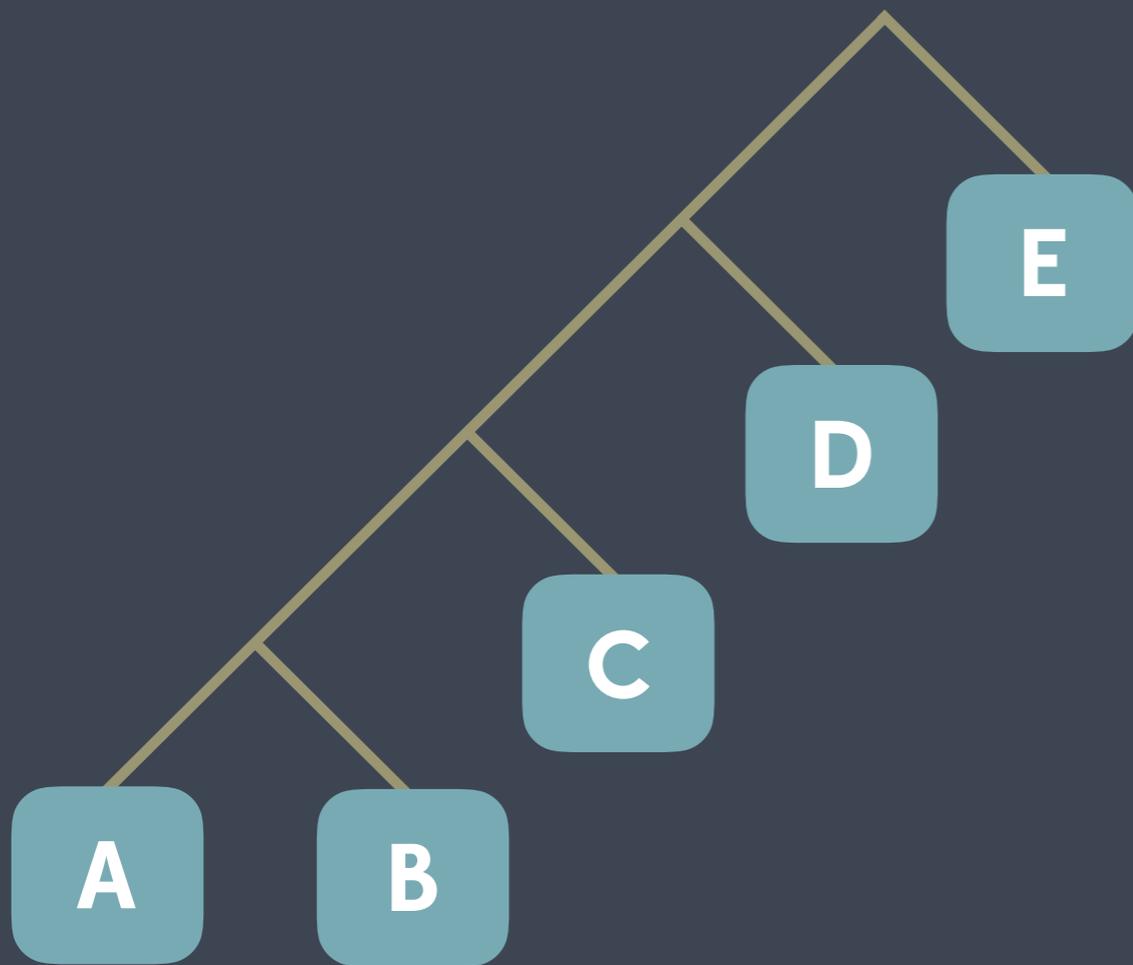
Join Order Enumeration

Join Tree Construction

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

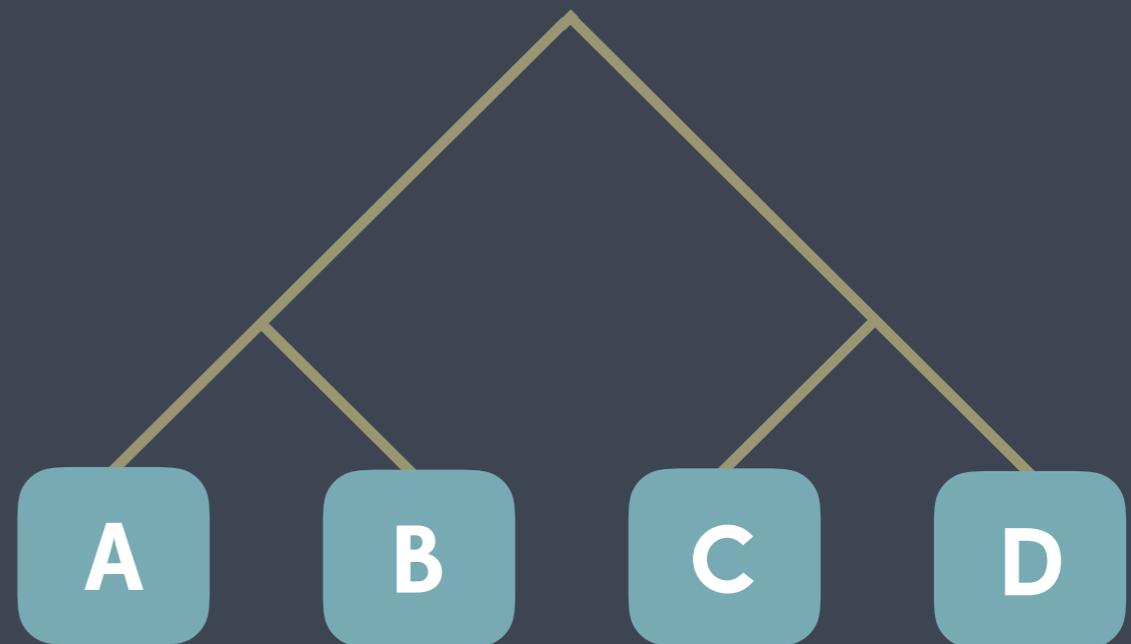
Join Trees

Left Sided



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

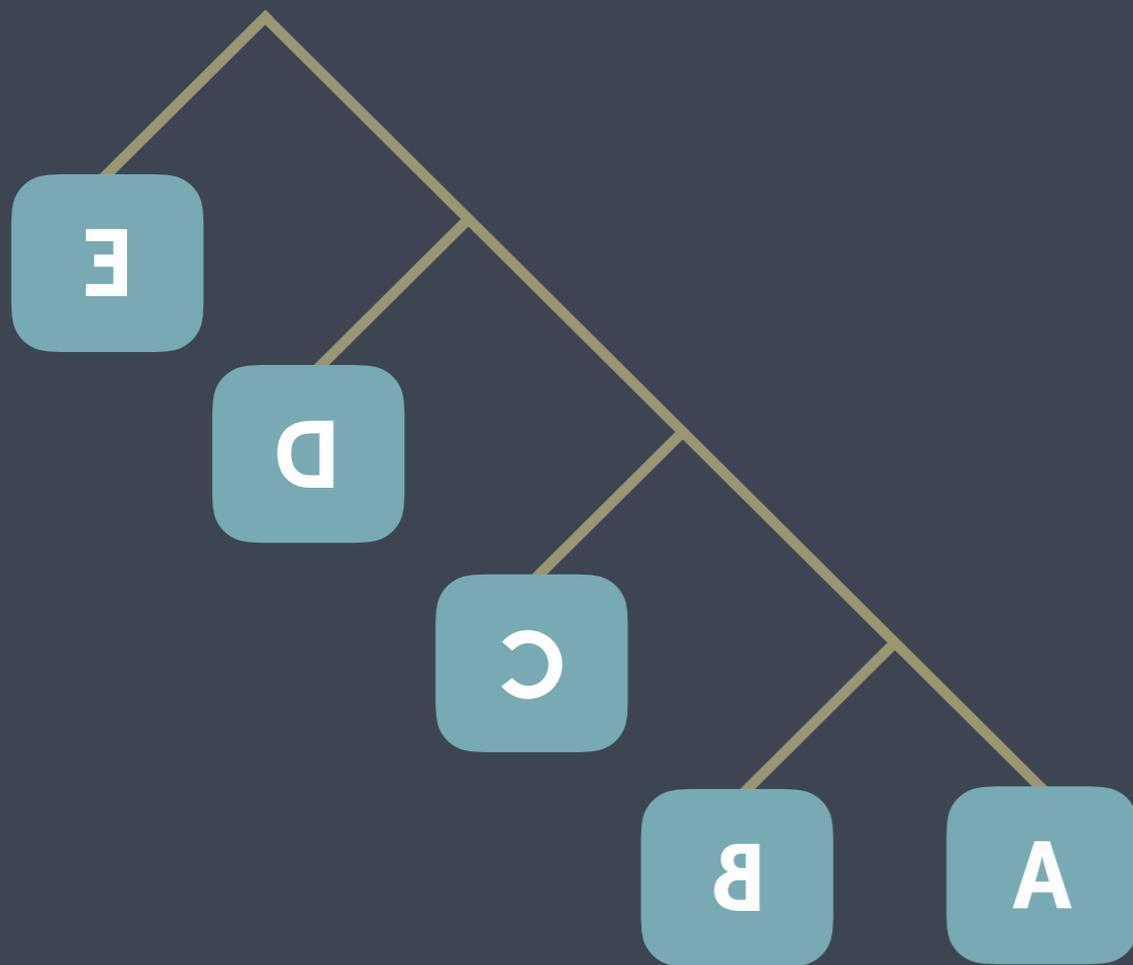
Bushy



$((AB)(CD))$

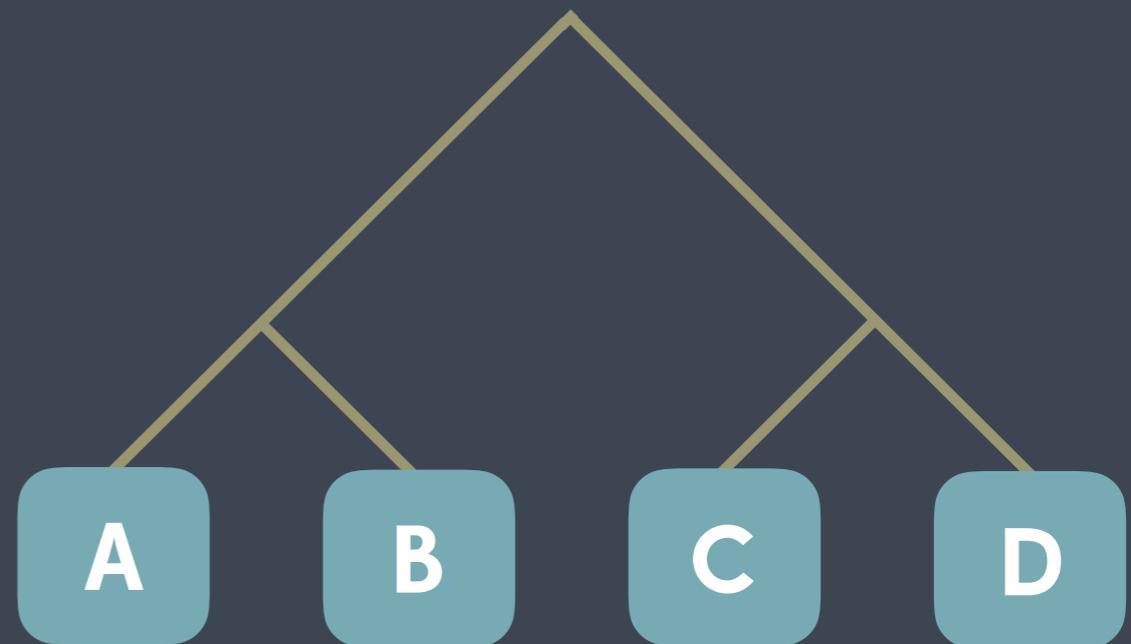
Join Trees

Right Sided



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

Bushy



$((AB)(CD))$

A



B



C



D

A ✕ B ✕ C ✕ D

N! join orderings:

ABCD, ABDC, ADCB, DABC ...

A ⋈ B ⋈ C ⋈ D

$N!$ join orderings:

ABCD, ABDC, ADCB, DABC ...

$(N-1)!$ plans per join order:

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

$N! \times (N-1)!$ possible plans

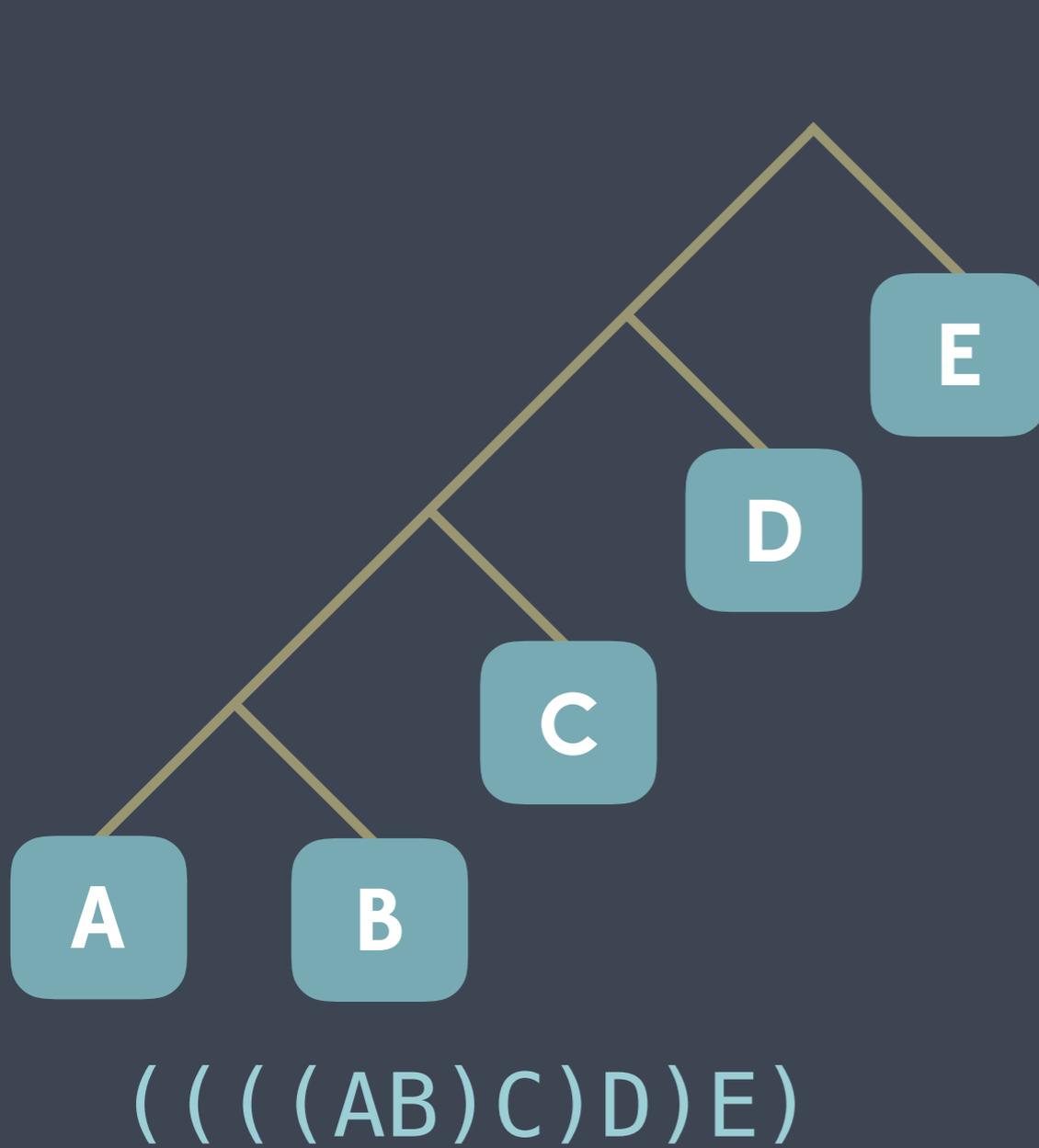
4 way join \rightarrow 144 plans

10 way join \rightarrow 1,316,818,944,000 plans

20 way ..  

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

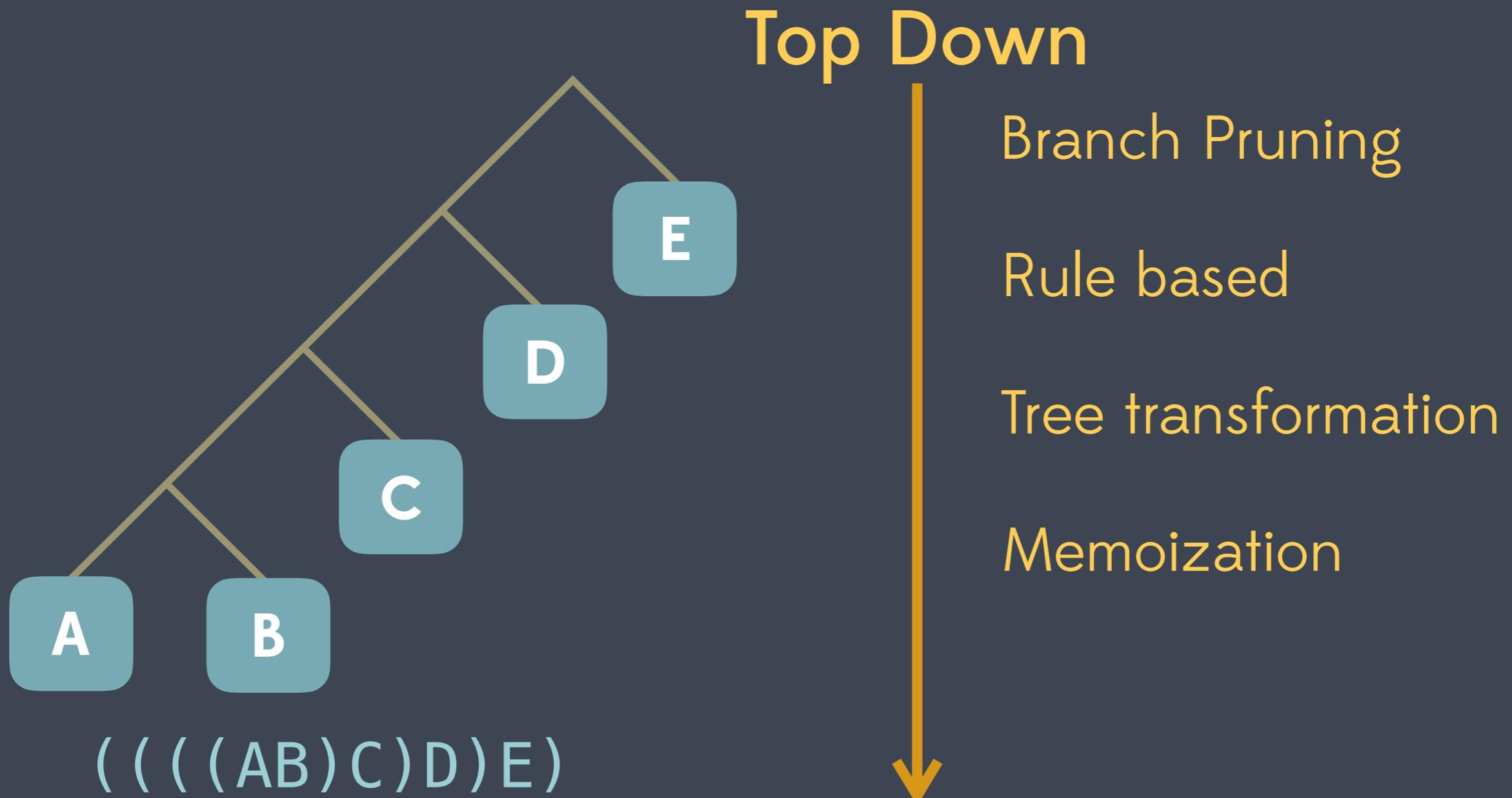
Strategies



Dynamic programming

Bottom Up

Strategies







ENDLICH ANGEKOMMEN
ATARI PERSONAL COMPUTER SYSTEM

ATARI 400 (16K) und ATARI 800 (bis 48K) 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.



A Warner Communications Company

Computers for people

Fordern Sie jetzt ausführliche Informationen an!

Name: _____

Beruf: _____

Straße: _____

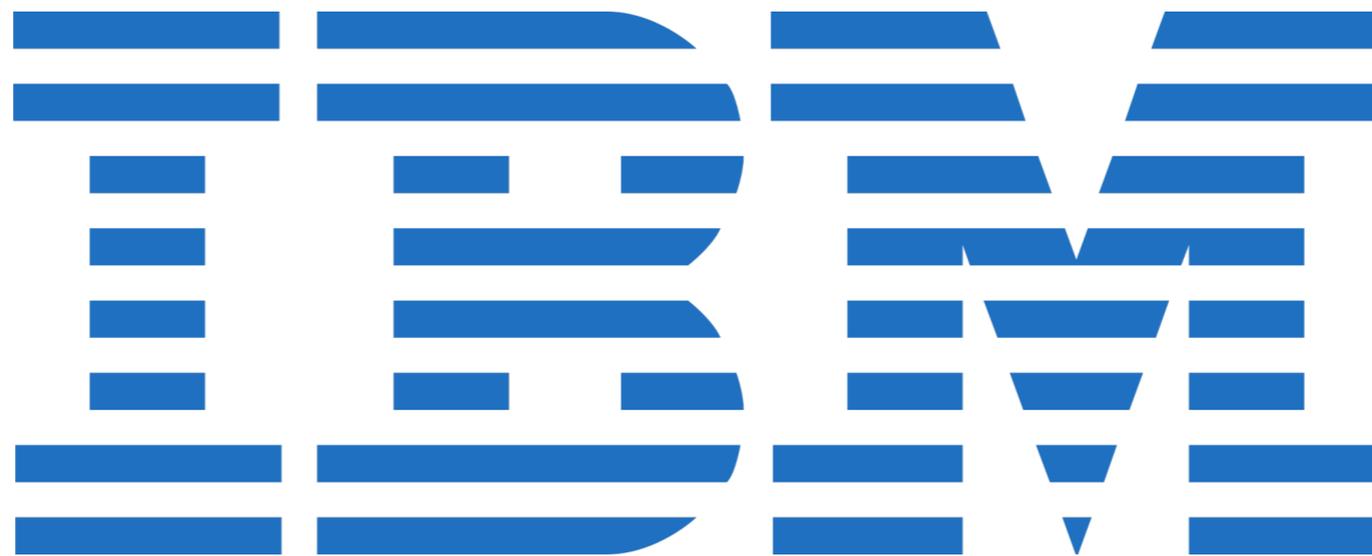
PLZ/Ort: _____

geplanter Einsatzbereich:
 Beruf Hobby Ausbildung Unterhaltung
Atari Elektronik Vertriebsgesellschaft mbH
Bebelallee 10 · 2000 Hamburg 60

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



1979



System/R

Patricia Selinger





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

Selinger Algorithm

Selinger Algorithm

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

Selinger Algorithm

- 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 Algorithm

- 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 Algorithm

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 Algorithm

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 Algorithm

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



B



C



D

Step 1 • Access Paths

$A = \text{OptimalAccess}(A_{\text{relation}});$

$B = \text{OptimalAccess}(B_{\text{relation}});$

■ ■ ■

Step 2 • 2-way Join

$\{A, B\} = \text{Cheapest}(AB, BA);$

$\{B, C\} = \text{Cheapest}(BC, CB);$

...

Step 3 • 3-way Join

$\{A, B, C\} = \text{Cheapest}(A\{B, C\}, \{B, C\}A, \\ B\{A, C\}, \{A, C\}B, \\ C\{A, B\}, \{A, B\}C);$

$\{A, B, D\} = \text{Cheapest}(A\{B, D\}, \{B, D\}A, \\ B\{A, D\}, \{A, D\}B, \\ D\{A, B\}, \{A, B\}D);$

...

Step 3 • 3-way Join

$\{A, B, C\} = \text{Cheapest}(A\{B, C\}, \{B, C\}A, \\ B\{A, C\}, \{A, C\}B, \\ C\{A, B\}, \{A, B\}C);$

$\{A, B, D\} = \text{Cheapest}(A\{B, D\}, \{B, D\}A, \\ B\{A, D\}, \{A, D\}B, \\ D\{A, B\}, \{A, B\}D);$

...

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

Selinger Extensions

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



Step 1 • Enumerate all access paths to individual relations, keep the cheapest **for all interesting orderings** around



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



GEQO - Genetic Query Optimizer

----- geqo_threshold -----

Selinger Algorithm

GEQO - Genetic Query Optimizer

geqo_threshold



Default: 12

Selinger Algorithm

GEQO

Heuristics required as the search space increase

Travelling salesman algorithm across the relations

..not terribly good, but better than waiting till the heat death of the universe

PostgreSQL Selinger

Keep interesting sort orders around

Use existing join clauses when possible, only attempt cartesian-product join when no clause

Bushy trees $\{AB\}\{CD\}$

PostgreSQL Selinger

```
SELECT *  
FROM   tab1, tab2, tab3, tab4  
WHERE  tab1.col = tab2.col AND  
       tab1.col = tab3.col AND  
       tab1.col = tab4.col
```

{1 2}, {1 3}, {1 4}

{1 2 3}, {1 3 4}, {1 2 4}

{1 2 3 4}

code

```

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);

```

```

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.
         */
        /* 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.
         */
        /* generate_gather_paths() for each list-item in rel. We
         * could not do this earlier because both regular and partial paths
         * can get added to a particular joinrel at multiple times within
         * one level. But now that we're done setting paths for
         * the joinrel, so run set_cheapest().
         */
        for (lc = list_head(root->join_rel_level[lev-1]); lc; lc = list_next(lc))
        {
            RelOptInfo *lfirst(lc);

            /* Gather paths for all useful partial rels for lev.
             * generate_gather_paths(root, rel);
             */
            /* find and save the cheapest paths for 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
         * a single level. After that we've done creating paths for
         * the joinrel, we can set cheapest.
         */
        for (lc = list_head(root->join_rel_level[lev]); lc; lc = list_next(lc, root->join_rel_level[lev]))
        {
            RelOptInfo *rel = (RelOptInfo *) lfirst(lc);

            /* Create Gather Paths 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 *));

    /* Find and save the cheapest paths for this rel */
    for (lev = 2; lev <= levels_needed; lev++)
    {
        /*
         * 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))
         *     continue;
         *
         * ListCell   *other_rels;
         * if (level == 2)
         *     /* consider remaining initial rels */
         *     other_rels = list(r);
         * else
         *     /* consider all initial rels */
         *     other_rels = list_head(joinrels[1]);
         *
         * make_rels_by_clauseless_joins(root,
         *                               old_rel,
         *                               other_rels);
         *
         * }
         *
         * for (k = 2; k < level; k++)
         *     int other_level = level - k;
         *     if (k > other_level)
         *         continue;
         *     foreach(r, joinrels[k])
         *     {
         *         RelOptInfo *new_rel = (RelOptInfo *) lfirst(r);
         *         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 = list(r); /* only consider remaining rels */
         *         else
         *             other_rels = list_head(joinrels[other_level]);
         *         foreach(r2, other_rels)
         *         {
         *             RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);
         *             if (have_relevant_joinclause(root, old_rel, new_rel) ||
         *                 have_join_order_restriction(root, old_rel, new_rel))
         *                 continue;
         *             (void) make_join_rel(root, old_rel, new_rel);
         *         }
         *     }
         *
         * }
         *
         * The result is returned in root->join_rel_level[level].
         */
        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)
{
    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
                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);
            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);
    }
}
```

```
/*
 * 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
                other_rels = list_head(joinrels[1]);

            /* 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(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 &&
        root->join_info_list == NIL &&
        !root->hasLateralRTEs)
        elog(ERROR, "failed to build any %d-way joins", 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 ..

```

    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)
{
    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]));
        }
    }
}

```

```

void
join_search_one_level(PlannerInfo *root,
                    int level)

```

```

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++)

```

```

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

```

```

void
join_search_one_level(PlannerInfo *root,
                      int level)

```

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

```

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

```

```

/* foreach(r, joinrels[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;
    /*
     * 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;

    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);
    }
}

```



Science





How Good Are Query Optimizers, Really?

Viktor Leis
TUM

leis@in.tum.de

Peter Boncz
CWI

p.boncz@cwi.nl

Andrey Gubichev
TUM

gubichev@in.tum.de

Alfons Kemper
TUM

kemper@in.tum.de

Atanas Mirchev
TUM

mirchev@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

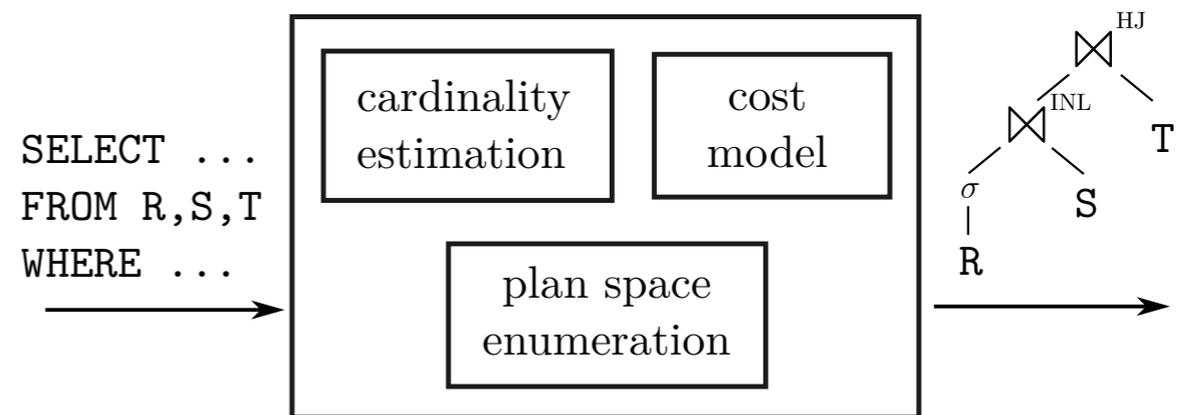


Figure 1: Traditional query optimizer architecture

- 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

How Good Are Query Optimizers, Really?

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.



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

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

Contributions

Challenging conventional wisdom is unglamorous but important research

Introduce a Join Order Benchmark using non-synthetic data (IMDB dataset)

A New, Highly Efficient, and Easy To Implement Top-Down Join Enumeration Algorithm

Pit Fender ^{#1}, Guido Moerkotte ^{#2}

[#]*Database Research Group, University of Mannheim
68131 Mannheim, Germany*

¹fender@informatik.uni-mannheim.de

³moerkotte@informatik.uni-mannheim.de

Abstract—Finding an optimal execution order of join operations is a crucial task in every cost-based query optimizer. Since there are many possible join trees for a given query, the overhead of the join (tree) enumeration algorithm per valid join tree should be minimal. In the case of a clique-shaped query graph, the best known top-down algorithm has a complexity of $\Theta(n^2)$ per join tree, where n is the number of relations. In this paper, we present an algorithm that has an according $O(1)$ complexity in this case.

We show experimentally that this more theoretical result has indeed a high impact on the performance in other non-clique settings. This is especially true for cyclic query graphs. Further, we evaluate the performance of our new algorithm and compare it with the best top-down and bottom-up algorithms described in the literature.

I. INTRODUCTION

For a DBMS that provides support for a declarative query language like SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows it to be translated into many equivalent evaluation plans. The process of choosing a suitable plan from all alternatives is known as query optimization. The basis of this choice are a cost model

Furthermore, since we exclude cross products, S_1 and S_2 must induce connected subgraphs of our query graph, and there must be two relations $R_1 \in S_1$ and $R_2 \in S_2$ such that they are connected by an edge, i.e., there must exist a join predicate involving attributes in R_1 and R_2 . Let us call such a partition (S_1, S_2) a *csg-cmp-pair* (or *ccp* for short). Denote by T_i the best plan for S_i . Then the query optimizer has to consider the plans $T_1 \bowtie T_2$ for all *csg-cmp-pairs* (S_1, S_2) .

One possibility to generate all *csg-cmp-pairs* for a set S of relations is to consider all subsets $S_1 \subset S$, define $S_2 = S \setminus S_1$, and then check the above conditions. Let us call such a procedure *naive generate and test* or *ngt* for short.

Table I gives for $n = 5, 10, 15, 20$ relations the number of connected subgraphs (*#csg*), the number of *csg-cmp-pairs* (*#ccp*), and the number of generated subsets S_1 for the naive generate and test algorithm (*#ngt*). These numbers were determined analytically ([2], [3]), but the formulas are not very intuitive. Therefore, we decided to illustrate our points with some explicit numbers¹.

Challenge. The number of subsets considered by naive



A New, Highly Efficient, and Easy To Implement Top-Down Join Enumeration Algorithm

We show experimentally that this more
theoretical result has indeed a high impact on
the performance in other non-clique settings.

Abstract—Finding an optimal execution order of join operations is a crucial task in every cost-based query optimizer. Since there are many possible join trees for a given query, the overhead of the join (tree) enumeration algorithm per valid join tree should be minimal. In the case of a clique-shaped query graph, the best known top-down algorithm has a complexity of $\Theta(n^2)$ per join tree, where n is the number of relations. In this paper, we present an algorithm that has an according $O(1)$ complexity in this case.

We show experimentally that this more theoretical result has indeed a high impact on the performance in other non-clique settings. This is especially true for cyclic query graphs. Further, we evaluate the performance of our new algorithm and compare it with the best top-down and bottom-up algorithms described in the literature.

I. INTRODUCTION

For a DBMS that provides support for a declarative query language like SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows it to be translated into many equivalent evaluation plans. The process of choosing a suitable plan from all alternatives is known as

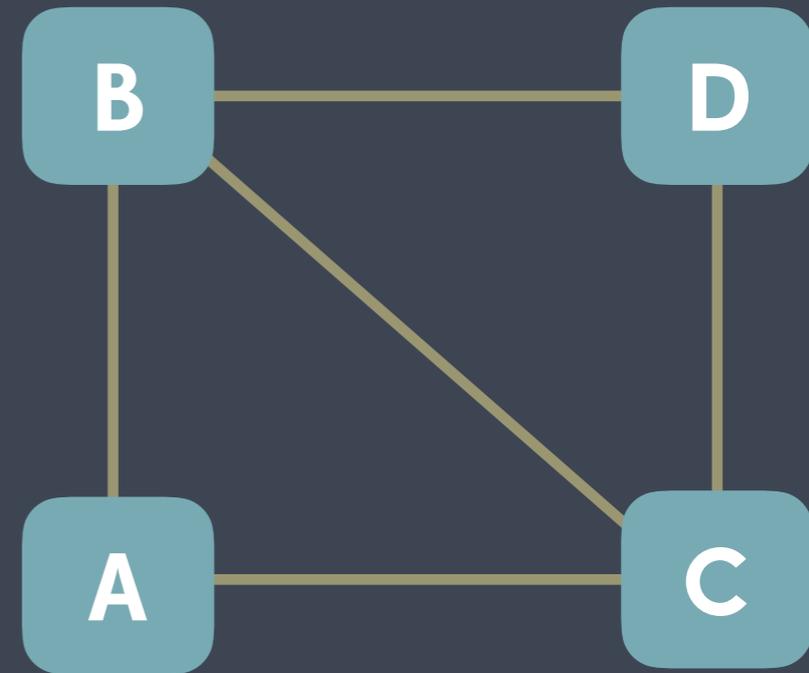
Furthermore, since we exclude cross products, S_1 and S_2 must induce connected subgraphs of our query graph, and there must be two relations $R_1 \in S_1$ and $R_2 \in S_2$ such that they are connected by an edge, i.e., there must exist a join predicate involving attributes in R_1 and R_2 . Let us call such a partition (S_1, S_2) a *csg-cmp-pair* (or *ccp* for short). Denote by T_i the best plan for S_i . Then the query optimizer has to consider the plans $T_1 \bowtie T_2$ for all *csg-cmp-pairs* (S_1, S_2) .

One possibility to generate all *csg-cmp-pairs* for a set S of relations is to consider all subsets $S_1 \subset S$, define $S_2 = S \setminus S_1$, and then check the above conditions. Let us call such a procedure *naive generate and test* or *ngt* for short.

Table I gives for $n = 5, 10, 15, 20$ relations the number of connected subgraphs (*#csg*), the number of *csg-cmp-pairs* (*#ccp*), and the number of generated subsets S_1 for the naive generate and test algorithm (*#ngt*). These numbers were determined analytically ([2], [3]), but the formulas are not very intuitive. Therefore, we decided to illustrate our points with some explicit numbers!

Contributions

$O(1)$ only for cliques



Cross-products not handled

Focus on implementation is refreshing

Dynamic Programming Strikes Back

Guido Moerkotte
University of Mannheim
Mannheim, Germany

moerkotte@informatik.uni-mannheim.de

Thomas Neumann
Max-Planck Institute for Informatics
Saarbrücken, Germany
neumann@mpi-inf.mpg.de

ABSTRACT

Two highly efficient algorithms are known for optimally ordering joins while avoiding cross products: **DPccp**, which is based on dynamic programming, and Top-Down Partition Search, based on memoization. Both have two severe limitations: They handle only (1) simple (binary) join predicates and (2) inner joins. However, real queries may contain complex join predicates, involving more than two relations, and outer joins as well as other non-inner joins.

Taking the most efficient known join-ordering algorithm, **DPccp**, as a starting point, we first develop a new algorithm, **DPhyp**, which is capable to handle complex join predicates efficiently. We do so by modeling the query graph as a (variant of a) hypergraph and then reason about its connected subgraphs. Then, we present a technique to exploit this capability to efficiently handle the widest class of non-inner joins dealt with so far. Our experimental results show that this reformulation of non-inner joins as complex predicates can improve optimization time by orders of magnitude, compared to known algorithms dealing with complex join predicates and non-inner joins. Once again, this gives dynamic programming a distinct advantage over current memoization techniques.

a dynamic programming algorithm to find the optimal join order for a given conjunctive query [21]. More precisely, they proposed to generate plans in the order of increasing size. Although they restricted the search space to left-deep trees, the general idea of their algorithm can be extended to the algorithm **DPsize**, which explores the space of bushy trees (see Fig. 1). The algorithm still forms the core of state-of-the-art commercial query optimizers like the one of DB2 [12].

Recently, we gave a thorough complexity analysis of **DPsize** [17]. We proved that **DPsize** has a runtime complexity which is much worse than the lower bound. This is mainly due to the tests (marked by '*' in Fig. 1), which fail far more often than they succeed. Furthermore, we proposed the algorithm **DPccp**, which exactly meets the lower bound. Experiments showed that **DPccp** is highly superior to **DPsize**. The core of their algorithm generates connected subgraphs in a bottom-up fashion.

The main competitor for dynamic programming is memoization, which generates plans in a top-down fashion. All known approaches needed tests similar to those shown for **DPsize**. Thus, with the advent of **DPccp**, dynamic programming became superior to memoization when it comes to generating optimal bushy join trees, which do not contain cross

Contributions

Handles all types of joins

Hypergraph approach

Important building block

Shows promise (I think..)

TOWN

THE ATOMIC BID
MILK FOUNTAIN SANDS & SODA

HOLT'S TIRE SERVICE



타 : 드 로

Summary

The Selinger Algorithm has stood the test of time and serve us really well

Work is probably needed to keep up with increased complexity and **BIG DATA**

The PostgreSQL codebase is of unrivalled quality

We're hiring, email me!

Thank you!

daniel@yesql.se • @d_gustafsson