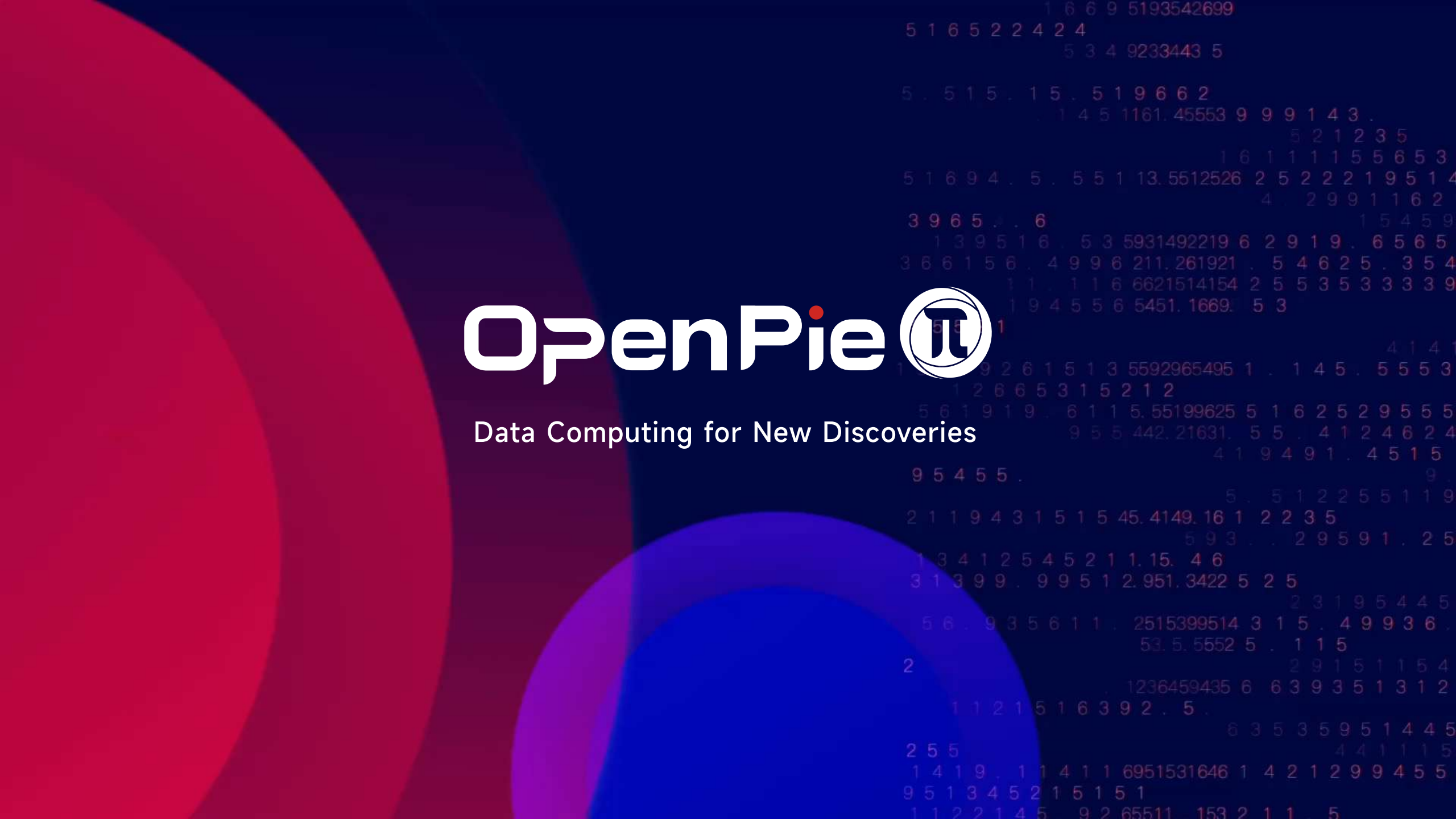
OpenPie
Data Computing for New Discoveries

A high-level introduction to the query planner in PostgreSQL

Richard Guo / OpenPie

Agenda

- About OpenPie and Me
- What Does Planner Do?
- Phases of Planning

OpenPie

Part One: About OpenPie and Me

OpenPie | PieCloudDB

- Dedicated to "Data Computing for New Discoveries"
- Aims to become a world-class high-tech innovation-driven institution in the field of data computing
- PieCloudDB, OpenPie’s flagship product, is a cutting-edge cloud-native data warehouse. It delivers elastic, highly available, and fully adaptable distributed HTAP database capabilities

About Me

Richard Guo

- PostgreSQL Contributor
- OpenPie/Pivotal

Overall Backend Structure

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

OpenPie

Part Two: What Does Planner Do?

What Does Planner Do?

- For a given query, find a correct execution plan that has the lowest cost
  - A given query can be actually executed in a wide variety of different ways
  - If it is computationally feasible, examine each of these possible ways, represented by data structures called Path
  - Select the cheapest Path and convert it to a full-fledged Plan

Part Three: Phases of Planning

Phases of Planning

- Preprocessing
  - Simplify the query if possible; collect information
- Scan/join planning
  - Decide how to implement FROM/WHERE
- Post scan/join planning
  - Deal with plan steps that are not scans or joins
- Postprocessing
  - Convert results into form the executor wants

Early Preprocessing

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

Simplify Scalar Expressions

- Simplify function calls
  - The function is strict and has any constant-null inputs

    int4eq(1,NULL) => NULL

  - The function is immutable and has all constant inputs

    2 + 2 => 4

Simplify Scalar Expressions

- Simplify boolean expressions

  "x OR true" => "true"

  "x AND false" => "false"

- Simplify CASE expressions

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

  ⇒ x+1

  ... not “ERROR: division by zero”

Expand Simple SQL Functions in-line

```
CREATE FUNCTION incr4(int) RETURNS int
AS 'SELECT $1 + (2 + 2)' LANGUAGE SQL;

SELECT incr4(a) FROM foo;

=>

SELECT a + 4 FROM foo;
```

Simplify Join Tree

- Convert IN, EXISTS sub-selects to semi-joins
- Flatten sub-selects if possible
- Reduce outer joins to inner joins
- Reduce outer joins to anti joins
- …

Convert IN, EXISTS Sub-selects to Semi-joins

```
SELECT * FROM foo WHERE EXISTS (SELECT 1 FROM bar WHERE foo.a = bar.c);

=>

SELECT * FROM foo SEMI JOIN bar ON foo.a = bar.c;
```

Flatten Sub-selects If Possible

```
SELECT * FROM foo JOIN (SELECT bar.c FROM bar JOIN baz ON TRUE) AS sub ON
foo.a = sub.c;
```

=>

```
SELECT * FROM foo JOIN (bar JOIN baz ON TRUE) ON foo.a = bar.c;
```

Reduce Outer Joins to Inner Joins

- If there is a strict qual above the outer join that constrains a Var from the nullable side of the join to be non-null

```
SELECT ... FROM foo LEFT JOIN bar ON (...) WHERE bar.d = 42;

=>

SELECT ... FROM foo INNER JOIN bar ON (...) WHERE bar.d = 42;
```

Reduce Outer Joins to Anti Joins

- If the outer join's own quals are strict for any nullable Var that was forced null by higher qual levels

```
SELECT * FROM foo LEFT JOIN bar ON foo.a = bar.c WHERE bar.c IS NULL;

=>

SELECT * FROM foo ANTI JOIN bar on foo.a = bar.c;
```

Later Preprocessing

- Distribute WHERE and JOIN/ON qual clauses
- Build equivalence classes for provably-equal expressions
- Gather information about join ordering restrictions
- Remove useless joins
- …

OpenPie

Distribute WHERE and JOIN/ON Qual Clauses

- A qual clause can be enforced at the lowest scan or join level that includes all relations used in the clause.
- Outer joins that may null some Vars in the clause are considered to be used in the clause.
- An outer join's own JOIN/ON quals mentioning nonnullable side rels cannot be pushed down below that outer join.

```
EXPLAIN (COSTS OFF)
SELECT * FROM foo LEFT JOIN bar ON foo.a = 42;
        QUERY PLAN
---------------------------
 Nested Loop Left Join
   Join Filter: (foo.a = 42)
   ->  Seq Scan on foo
   ->  Materialize
         ->  Seq Scan on bar
(5 rows)
```

```
EXPLAIN (COSTS OFF)
SELECT * FROM foo LEFT JOIN bar ON foo.a = bar.c WHERE COALESCE(bar.c,
1) = 42;
            QUERY PLAN
----------------------------------
 Hash Left Join
   Hash Cond: (foo.a = bar.c)
   Filter: (COALESCE(bar.c, 1) = 42)
   ->  Seq Scan on foo
   ->  Hash
         ->  Seq Scan on bar
(6 rows)
```

Equivalence Classes

- An Equivalence Class represents a set of values that are known all transitively equal to each other
- Equivalence clauses are removed from the standard qual distribution process. Instead, eclass-based qual clauses are generated dynamically when needed
- Equivalence Classes also represent the value that a PathKey orders by (since if x = y, then ORDER BY x must be the same as ORDER BY y)

Gather Information About Join Ordering Restrictions

- One-sided outer joins constrain the order of joining partially but not completely
  - ➢ non-FULL joins can be freely associated into the lefthand side of an OJ, but in some cases they can't be associated into the righthand side

(A leftjoin B on (Pab)) innerjoin C on (Pac)
= (A innerjoin C on (Pac)) leftjoin B on (Pab)

(A leftjoin B on (Pab)) innerjoin C on (Pbc)
!= A leftjoin (B innerjoin C on (Pbc)) on (Pab)

Gather Information About Join Ordering Restrictions

- One-sided outer joins constrain the order of joining partially but not completely
- We flatten non-FULL joins to top-level "joinlist" so that they participate fully in the join order search
- We record information about each outer join, in order to avoid generating illegal join orders

Scan/Join Planning

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

Scan/Join Planning

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

Join Searching

- Multi-way joins have to be built up from pairwise joins, because that is all the executor knows how to do
- For any given pairwise join step, we can identify the best input Paths and join methods via straightforward cost comparisons, resulting in a list of Paths much as for a base relation
- Finding the best ordering of pairwise joins is the hard part

Join Searching

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

Standard Join Search Method

- Generate paths for each base relation
- Generate paths for each possible two-way join
- Generate paths for each possible three-way join
- Generate paths for each possible four-way join
- …
- Continue until all base relations are joined into a single join relation; then use that relation's best path

Join Searching is Expensive

- An n-way join problem can potentially be implemented in n! (n factorial) different join orders
- It is not feasible to consider all possibilities
- We use a few heuristics, like not considering clause-less joins
- With too many relations (12 by default), fall back to GEQO (genetic query optimizer) search

Heuristics Used in Join Search

- Don't join relations that are not connected by any join clause, unless forced to by join-order restrictions
- Break down large join problems into sub-problems by not flattening JOIN clauses according to collapse limit

Post Scan/Join Planning

- Deal with GROUP BY, aggregation, window functions, DISTINCT
- Deal with UNION/INTERSECT/EXCEPT
- Apply final sort if needed for ORDER BY
- Produce one or more “Path” data structures for each step
- Add LockRows, Limit, ModifyTable steps to each surviving Path

Postprocessing

- Expand best Path to Plan
- Adjust some representational details of Plan
  - Flatten subquery rangetables into a single list
  - Label Vars in upper plan nodes as OUTER_VAR or INNER_VAR, to refer to the outputs of their subplans
  - Remove unnecessary SubqueryScan, Append, and MergeAppend plan nodes
  - etc

OpenPie

Thanks

Data Computing for New Discoveries

OpenPie_TSP

OpenPie

OpenPie

https://en.openpie.com/