HINT BASED OPTIMIZATION

FROM `PG_HINT_PLAN` EXTENSION TO MODERN ML SOLUTIONS

Sergey @zinchse Zinchenko

openGauss contributor, R&D engineer at Huawei RRI

AGENDA

1. WHAT CHALLENGES DOES THE OPTIMIZER FACE?
2. HOW WE CAN HELP TO COMBAT THEM?
3. OPTIMIZER & PG_HINT_PLAN IMPLEMENTATION
4. PRESENT AND FUTURE OF HINT BASED OPTIMIZATION

1. WHAT CHALLENGES DOES THE OPTIMIZER FACE?

DECLARATIVENESS

```
.sql

/* magic */

answer
```

The .sql describes *what data is needed*
The planner needs to figure out
how to get it (optimally, ofc)

QUERY EXECUTION FLOW

.sql

/* rewriting */
/* parsing */
/* planning / optimizing */

physical operations

/* executing */

answer

QUERY EXECUTION FLOW

.sql

/* rewriting */

/* parsing */

/* planning / optimizing */

physical operations

/* executing */

answer

Why do we need a plan stage?

We've already parsed all instructions from the .sql, right?

Q

How does the optimizer look for a best plan to execute?

a) randomly

b) based on heuristics and rules

c) based on statistics

d) we write the plan ourselves

e) none of this

```
SELECT * FROM T1, T2, T3;
```

What is the **best** plan?

```
SELECT * FROM T1, T2, T3;
```

What is the **best** plan?

We need to know the sizes

EX 1

```
SELECT *
FROM SmallT, MediumT, BigT;
```

What is the **best** plan now?

/* LET'S ASK PG */

EX 1

```
SELECT *
FROM SmallT, MediumT, BigT;
```

1: Cost[(SmallT ⋈ MediumT) ⋈ BigT] = $S * M + S * M * B$

2: Cost[SmallT ⋈ (MediumT ⋈ BigT)] = $M * B + S * M * B$

…

The order (SmallT ⋈ MediumT) ⋈ BigT is **cheapest**

EX 1

```
SELECT *
FROM SmallT, MediumT, BigT;
```

Hm...

We can find the best order by knowing the sizes and going through all the combinations.

So what's the Challenge?

```
SELECT *
FROM SmallT, MediumT, BigT
WHERE ...;
```

EX #2

Table Food(breakfast::text, lunch::text) contains 2 * N tuples;
50% of them have breakfast = '🥐☕';
50% of them have lunch = '🥘';

What is the result of the following query?

```
SELECT COUNT(*)
FROM Food F
WHERE F.breakfast='🥐☕'
    AND F.lunch='🥘';
```

/* LET'S ASK PG */

CHALLENGE 1 – CARDINALITY ESTIMATION

What is the result of the following query?

```
SELECT COUNT(*)
FROM Food F
WHERE F.breakfast='🥐☕'
    AND F.lunch='🥘';
```

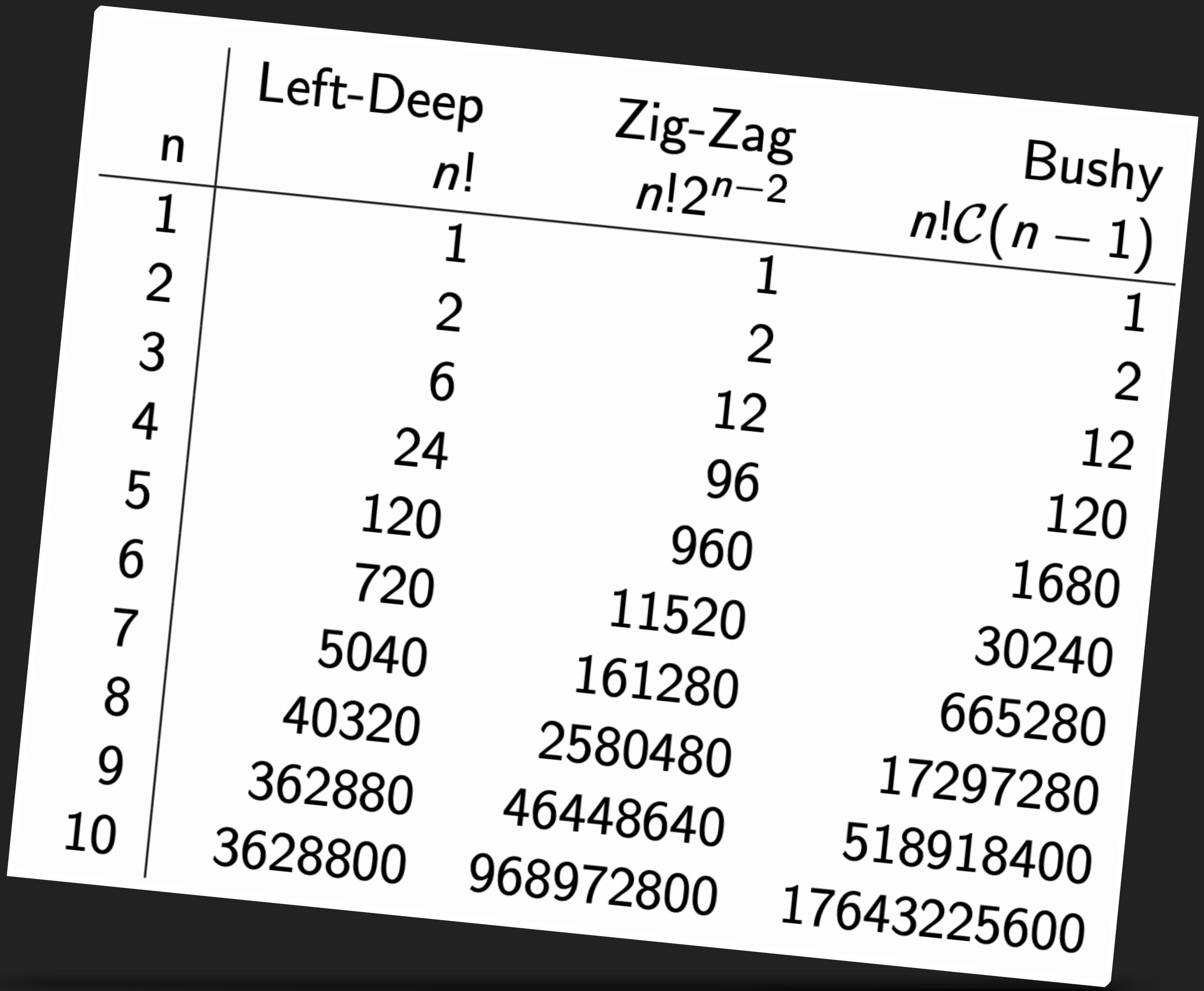
For a estimation, we need to known **correlations**

```
SELECT *
FROM T_1, T_2, ..., T_N;
```

How to **find** best order?

CHALLENGE 2 – JOIN ORDER ENUMERATION

Even if the sizes of all tables are known, finding the optimal order of joins remains a difficult problem. Since the search space is **huge.**

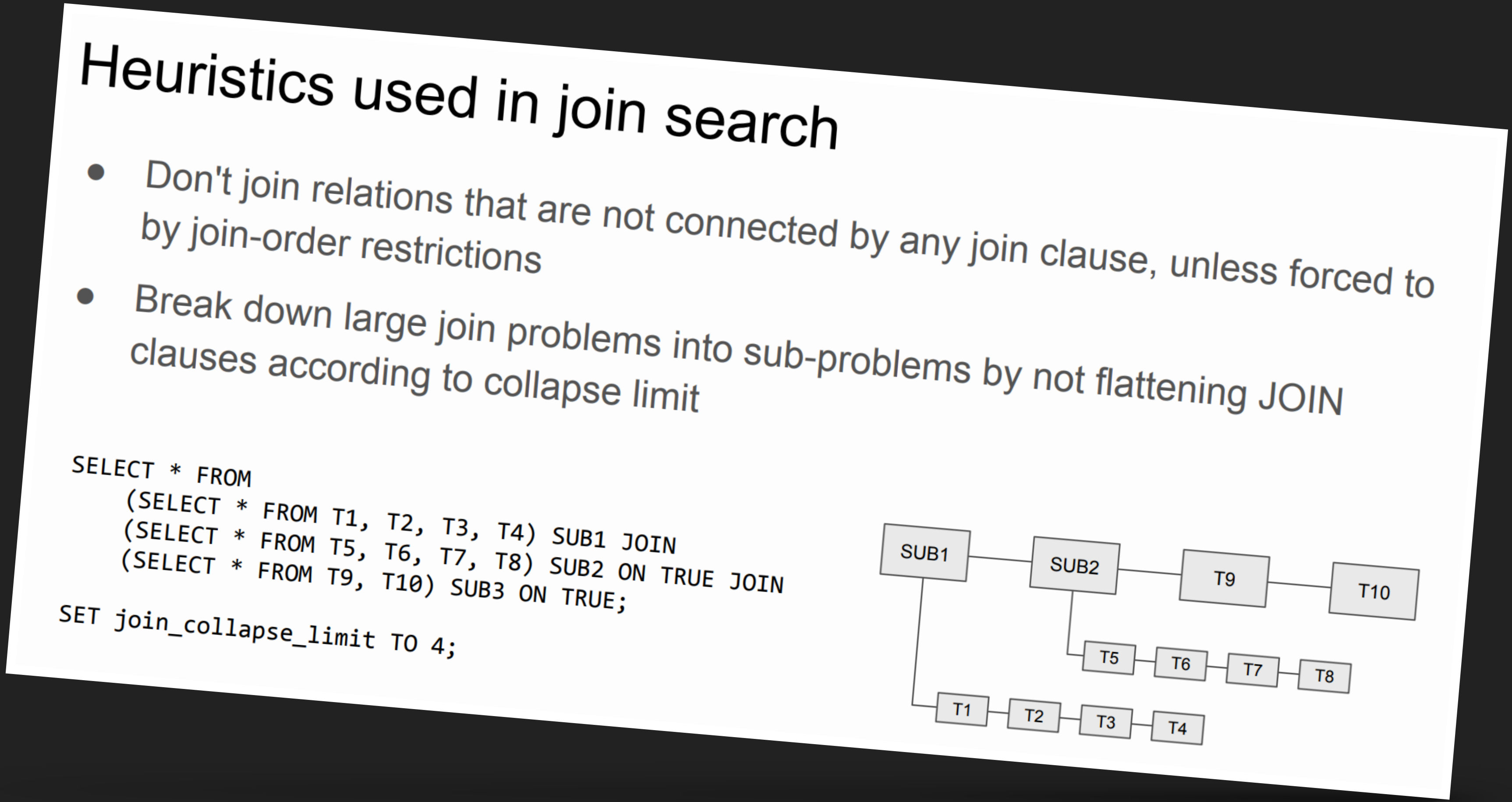


n	Left-Deep $n!$	Zig-Zag $n!2^{n-2}$	Bushy $n!\mathcal{C}(n-1)$
1	1	1	1
2	2	2	2
3	6	12	12
4	24	96	120
5	120	960	1680
6	720	11520	30240
7	5040	161280	665280
8	40320	2580480	17297280
9	362880	46448640	518918400
10	3628800	968972800	17643225600

/* JOIN AND FROM COLLAPSE LIMITS */

The "joinlist" is a list of items that are either RangeTblRef
jointree nodes or sub-joinlists.
All the items at the same level of joinlist must be joined in an order
to be detrmined by make_one_rel().
A sub-joinlist represents a subproblem to be planned separately.

/* JOIN AND FROM COLLAPSE LIMITS */



Heuristics used in join search

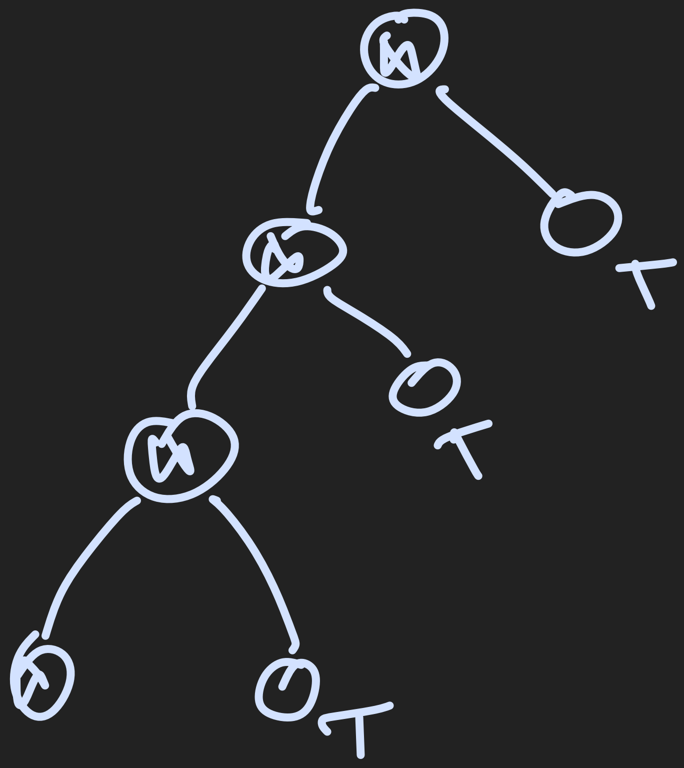
- Don't join relations that are not connected by any join clause, unless forced to by join-order restrictions
- Break down large join problems into sub-problems by not flattening JOIN clauses according to collapse limit

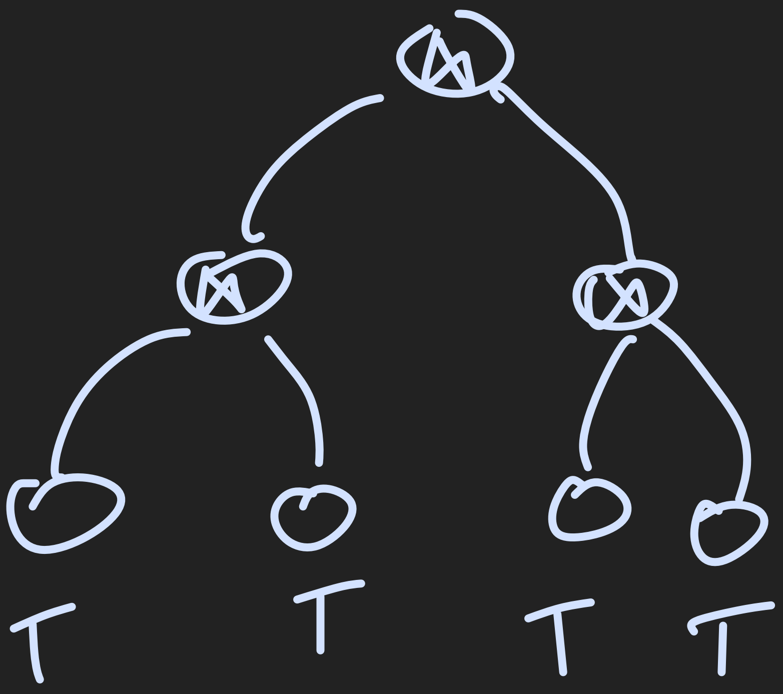
```
SELECT * FROM
    (SELECT * FROM T1, T2, T3, T4) SUB1 JOIN
    (SELECT * FROM T5, T6, T7, T8) SUB2 ON TRUE JOIN
    (SELECT * FROM T9, T10) SUB3 ON TRUE;

SET join_collapse_limit TO 4;
```

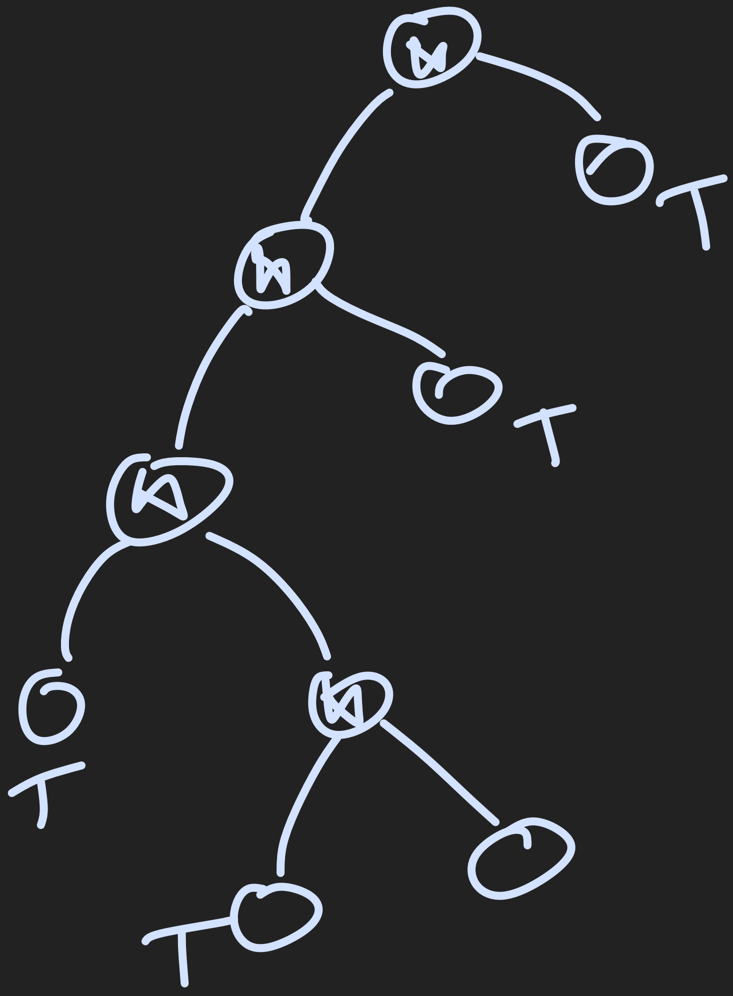
The items at the same level of joinlist must be joined by make_one_rel(). A sub-joinlist represents a subproblem to be planned separately.

QUESTION

Left/Right-deep vs Bushy vs ZigZag







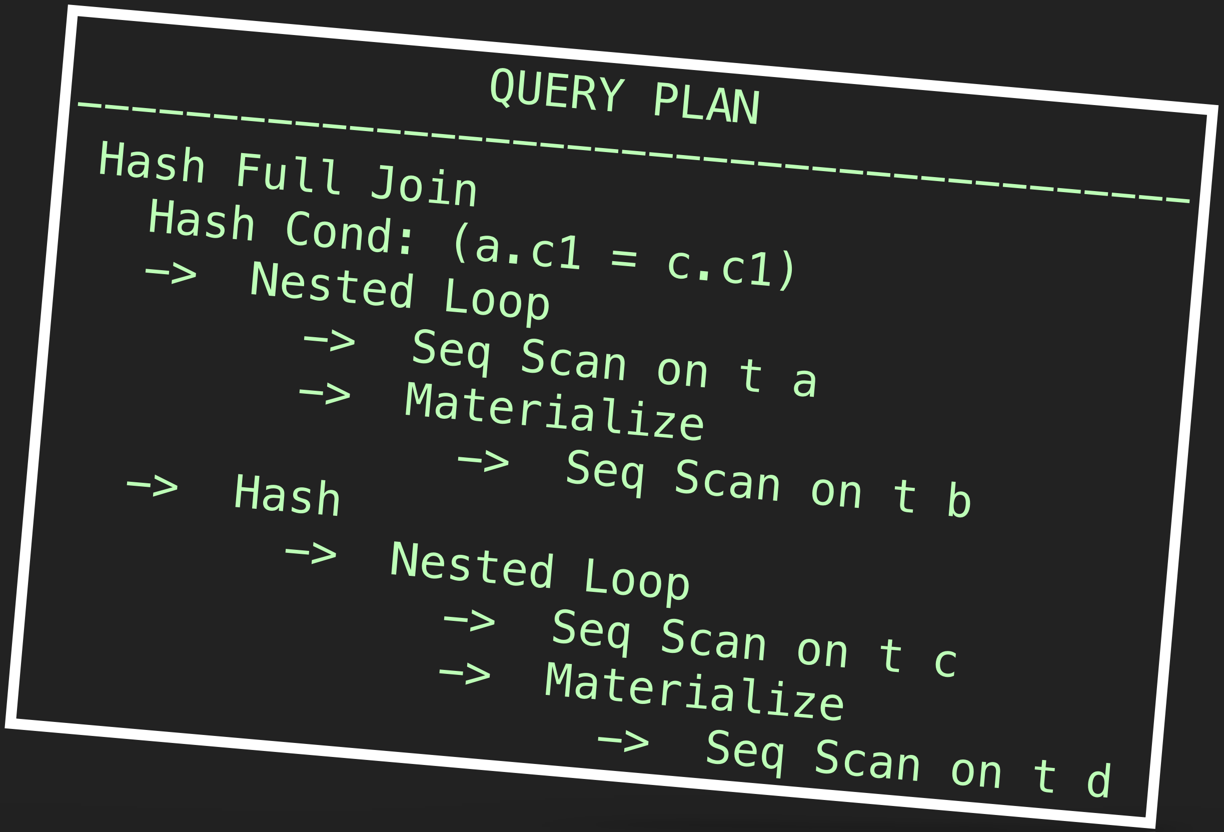
QUESTION

What kind of tree is this?

a) bushy

b) zig-zag

c) left-deep



```
                    QUERY PLAN
---------------------------------------------------
Hash Full Join
  Hash Cond: (a.c1 = c.c1)
  ->  Nested Loop
        ->  Seq Scan on t a
        ->  Materialize
              ->  Seq Scan on t b
  ->  Hash
        ->  Nested Loop
              ->  Seq Scan on t c
              ->  Materialize
                    ->  Seq Scan on t d
```

Hash vs NestLoop vs Merge

What is the **best** implementation?

/* LET'S ASK PG */

/* LET'S ASK PG */

but later

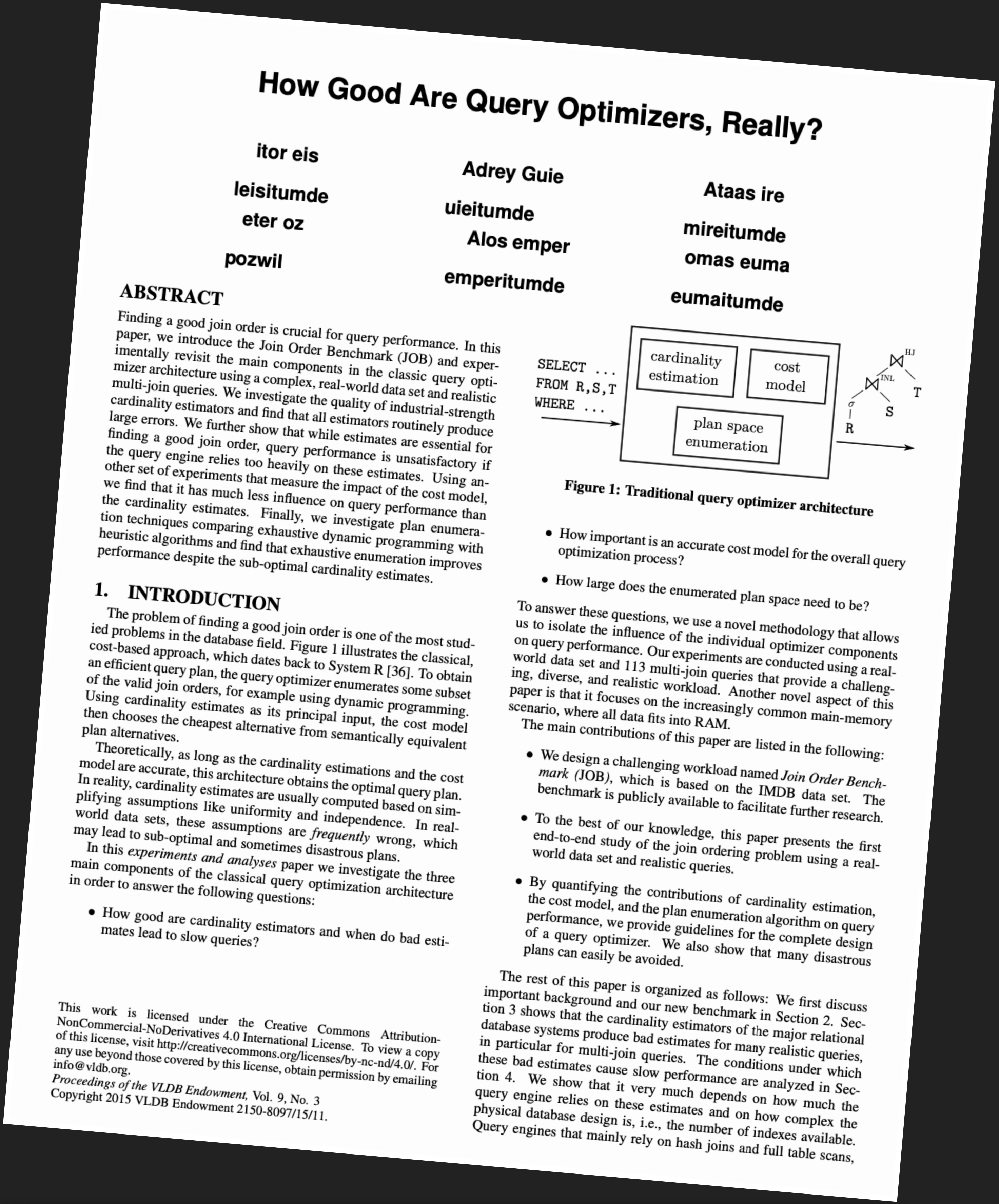
Hash vs NestLoop vs Merge

What is the **best** implementation?

Depending on the situation, either type of joins may be optimal.
Moreover, the physical join operator no longer has commutativity!

E2E PERFORMANCE

"The goal of this paper is to investigate the contribution of all relevant query optimizer components to end-to-end query performance …"



How Good Are Query Optimizers, Really?

itor eis
leisitumde
eter oz
pozwil

Adrey Guie
uieitumde
Alos emper
emperitumde

Ataas ire
mireitumde
omas euma
eumaitumde

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 of the valid join orders, for example using dynamic programming. Using cardinality estimates as its principal input, the cost model then chooses the cheapest alternative from semantically equivalent plan alternatives.

Theoretically, as long as the cardinality estimations and the cost model are accurate, this architecture obtains the optimal query plan. In reality, cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. In real-world data sets, these assumptions are *frequently* wrong, which may lead to sub-optimal and sometimes disastrous plans.

In this *experiments and analyses* paper we investigate the three main components of the classical query optimization architecture in order to answer the following questions:

- How good are cardinality estimators and when do bad estimates lead to slow queries?

Proceedings of the VLDB Endowment, Vol. 9, No. 3
Copyright 2015 VLDB Endowment 2150-8097/15/11.

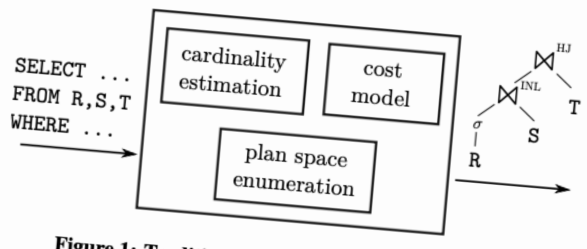


Figure 1: Traditional query optimizer architecture

- 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 scenario, where all data fits into RAM.

The main contributions of this paper are listed in the following:

- We design a challenging workload named *Join Order Benchmark (JOB)*, which is based on the IMDB data set. The benchmark is publicly available to facilitate further research.
- To the best of our knowledge, this paper presents the first end-to-end study of the join ordering problem using a real-world data set and realistic queries.
- By quantifying the contributions of cardinality estimation, the cost model, and the plan enumeration algorithm on query performance, we provide guidelines for the complete design of a query optimizer. We also show that many disastrous plans can easily be avoided.

The rest of this paper is organized as follows: We first discuss important background and our new benchmark in Section 2. Section 3 shows that the cardinality estimators of the major relational database systems produce bad estimates for many realistic queries, in particular for multi-join queries. The conditions under which these bad estimates cause slow performance are analyzed in Section 4. We show that it very much depends on how much the query engine relies on these estimates and on how complex the physical database design is, i.e., the number of indexes available. Query engines that mainly rely on hash joins and full table scans,

E2E PERFORMANCE

Statement 1: **complex** queries are **hard** to optimize

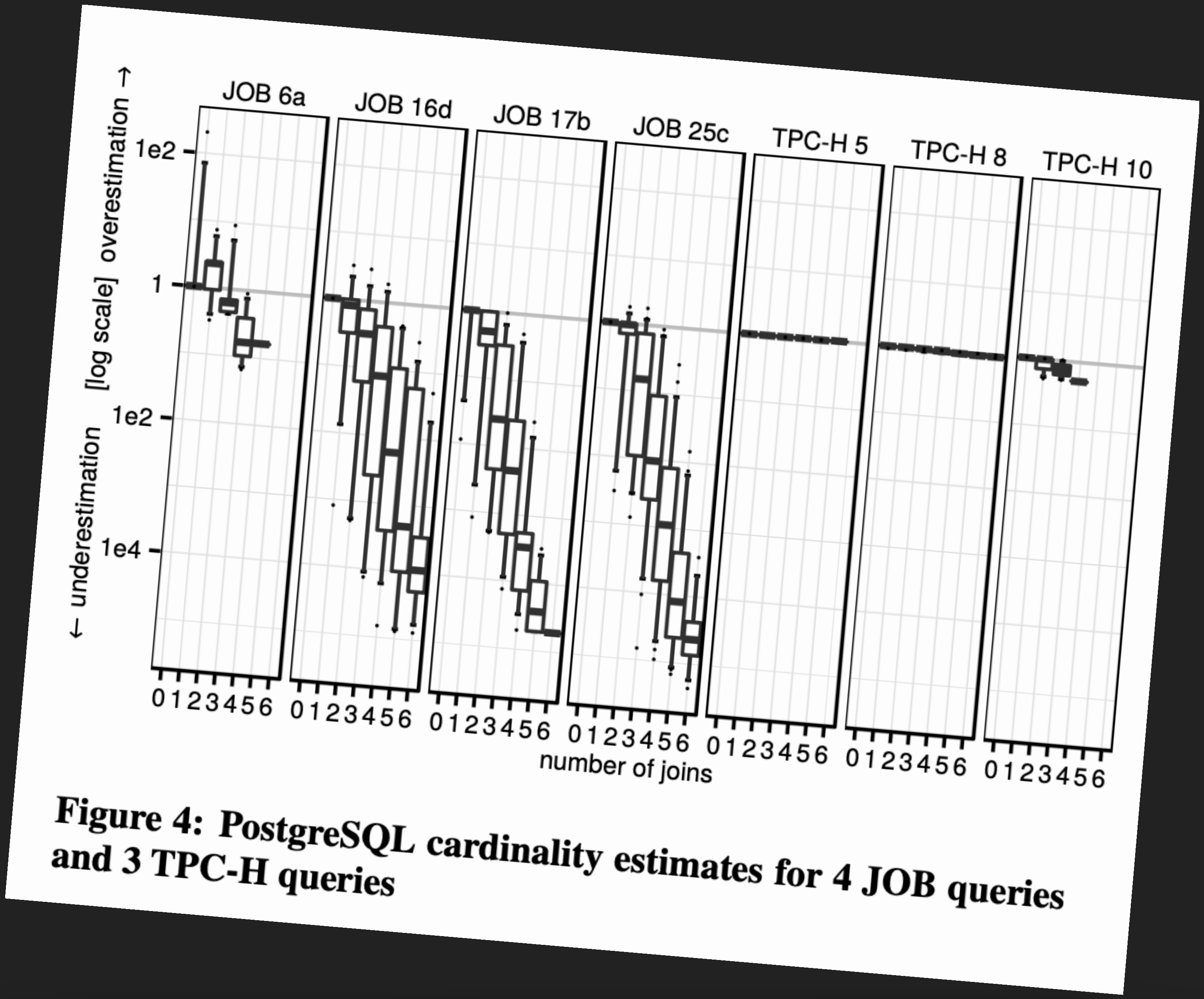


Figure 4: PostgreSQL cardinality estimates for 4 JOB queries and 3 TPC-H queries

E2E PERFORMANCE

Statement 2: cardinality is **important**

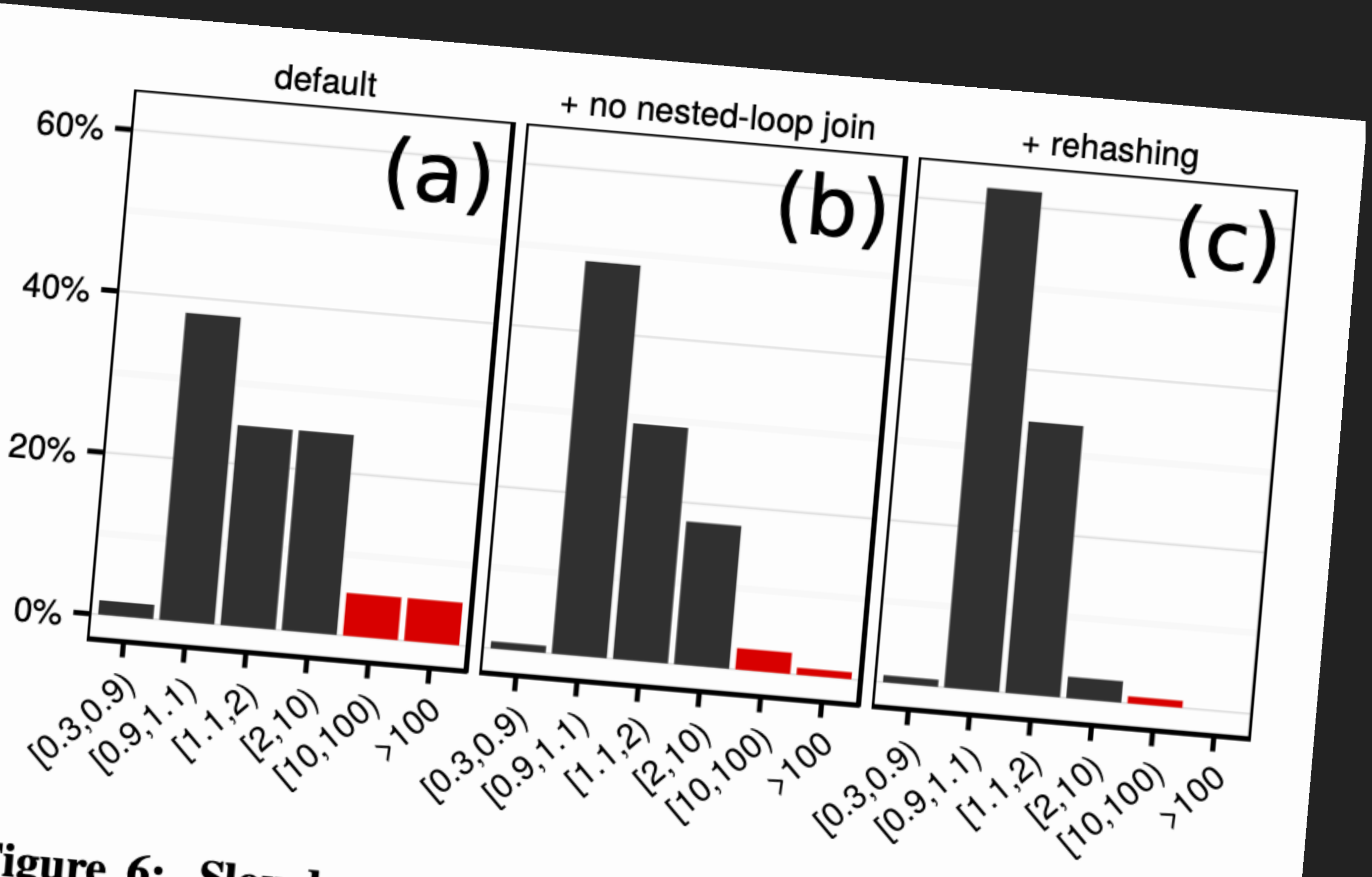


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

E2E PERFORMANCE

Statement 3:
join order is
super important

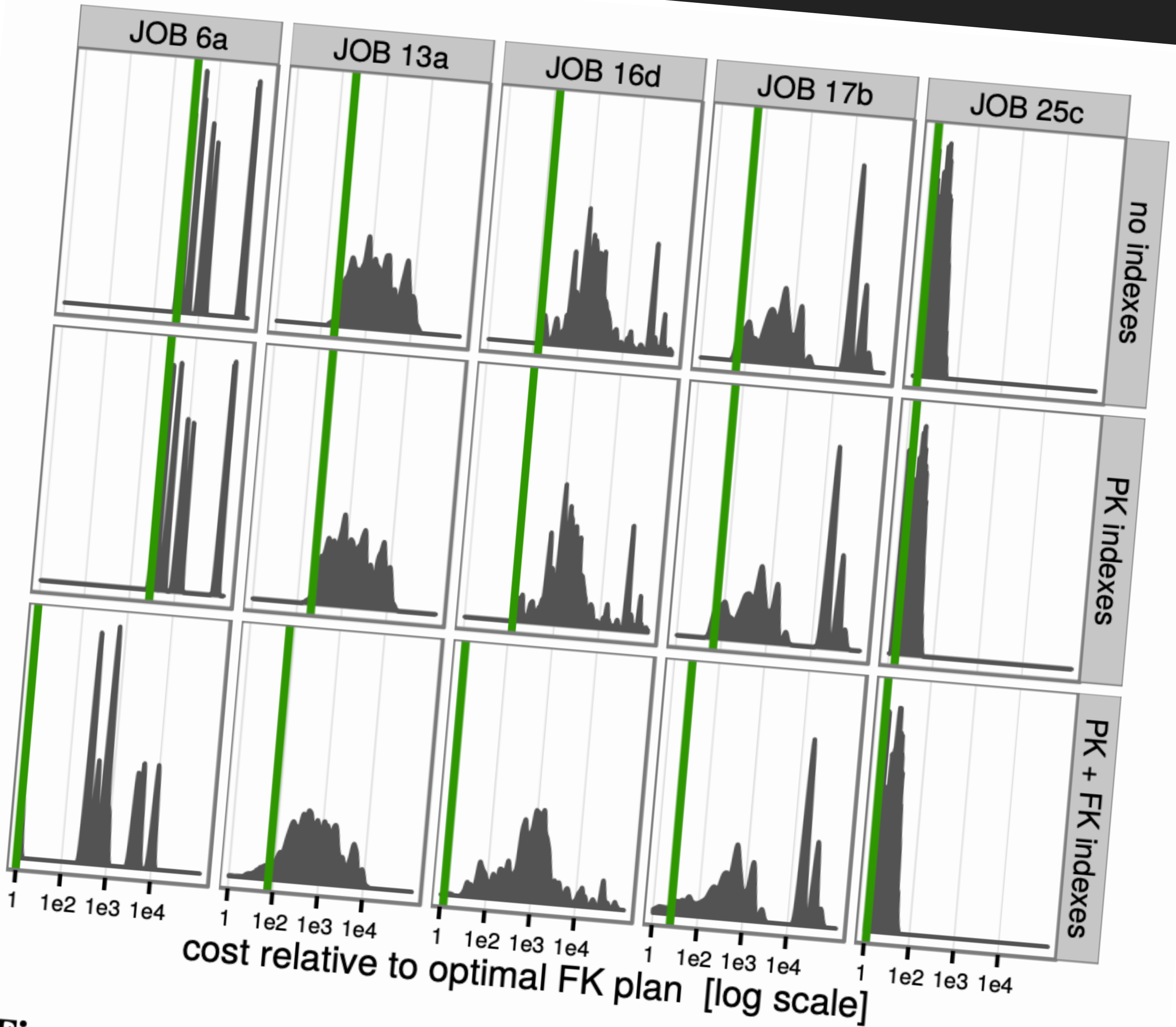


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

1. RESUME

Even full enumeration are **worthless** unless the cardinality estimates are correct

Effects of query optimization are always gated by the **available options** (in terms of access paths)

Join-crossing correlations remain an **open frontier** for database research

2. HOW WE CAN HELP TO COMBAT THIS CHALLENGES?

CARDINALITIES

We can build more accurate estimation models, or … get hint in the way

/* Hi, optimizer!

Please, [set / multiply] cardinalities of …

[to / by] X

*/

JOIN ORDER

We can get accurate estimations and explore full space, or … get hint in the way

/* Hi, optimizer!

Please, start join order with A JOIN B, …

*/

PHYSICAL OPERATOR

We can say DON'T USE NESTED LOOP JOIN, or … get hint in the way

/* Hi, optimizer!

Please, use HashJoin for A JOIN B

*/

PG_HINT_PLAN INTERFACE

CARDINALITY HINT

/* Hi, optimizer!

Please, [set / multiply] cardinalities of

A JOIN B [to / by] X

*/

$\downarrow$

/* Rows(A B [# / *]X) */

JOIN ORDER HINT

/* Hi, optimizer!
Please, start join order with A JOIN B ...
*/

$\downarrow$

/* Leading (A B ...) */

PHYSICAL OPERATOR HINT

/* Hi, optimizer!
Please, use HashJoin for A JOIN B
*/

↓

/* HashJoin (A B ...) */

/* TIME TO CODE */

2. RESUME

Using the hints, we can *optimize* the *optimizer*:
- change the cardinality estimations
- choose the join order
- select the physical operators

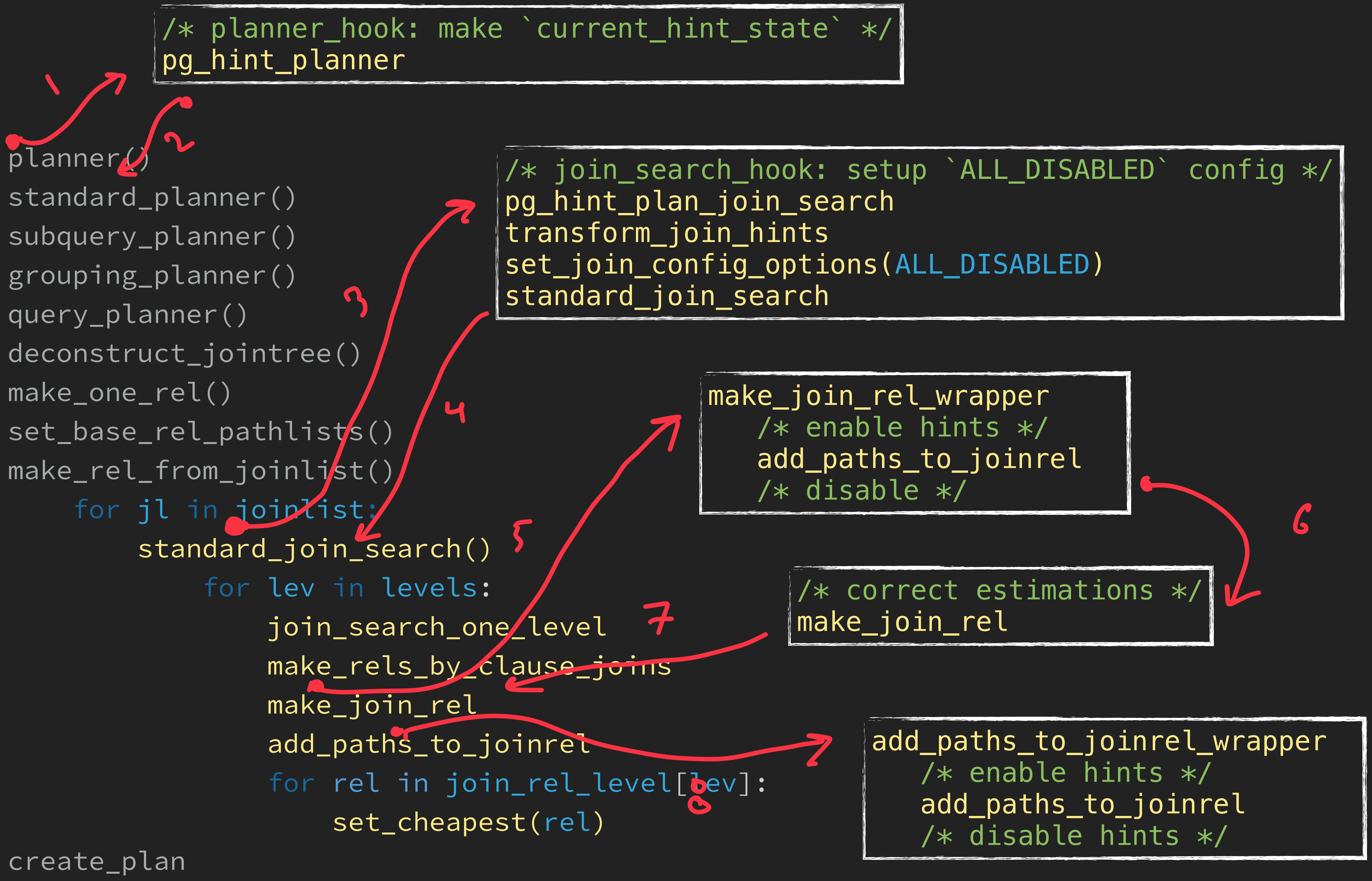
It's **Hint-based Optimization**

3. OPTIMIZER & PG_HINT_PLAN IMPLEMENTATION

DEFAULT FUNCTIONS FLOW

```
planner()
standard_planner() /* setup global state, preprocessing */
subquery_planner() /* recursive processing of sub-selects, pull-upping subqueries */
grouping_planner() /* top-level processing */
query_planner() /* basic query processing */
deconstruct_jointree() /* using heuristics to simplify the search process */
make_one_rel() /* start point for join orderS enumeration */
set_base_rel_pathlists() /* initial sizes and paths */
make_rel_from_joinlist() /* recursive join enumeration for sub-joinlist */
    for jl in joinlist:
        standard_join_search() /* DP entry point*/
            for lev in levels:
                join_search_one_level /* main DP routine */
                make_rels_by_clause_joins /* iterating over candidates */
                make_join_rel /* overlapping & legality check */
                add_paths_to_joinrel /* add paths according to join type */
                for rel in join_rel_level[lev]:
                    set_cheapest(rel) /* selecting the best way to obtain data */
create_plan
```

FUNCTIONS FLOW WITH PG_HINT_PLAN



```
/* planner_hook: make `current_hint_state` */
pg_hint_planner
```

```
planner()
standard_planner()
subquery_planner()
grouping_planner()
query_planner()
deconstruct_jointree()
make_one_rel()
set_base_rel_pathlists()
make_rel_from_joinlist()
    for jl in joinlist:
        standard_join_search()
            for lev in levels:
                join_search_one_level
                make_rels_by_clause_joins
                make_join_rel
                add_paths_to_joinrel
                for rel in join_rel_level[lev]:
                    set_cheapest(rel)
create_plan
```

```
/* join_search_hook: setup `ALL_DISABLED` config */
pg_hint_plan_join_search
transform_join_hints
set_join_config_options(ALL_DISABLED)
standard_join_search
```

```
make_join_rel_wrapper
   /* enable hints */
   add_paths_to_joinrel
   /* disable */
```

```
/* correct estimations */
make_join_rel
```

```
add_paths_to_joinrel_wrapper
   /* enable hints */
   add_paths_to_joinrel
   /* disable hints */
```

AUTOMATIZATION

We can pre-define a set of hints that will be applied to a class of queries (based on their template)

```
INSERT INTO
hint_plan.hints (norm_query_string, application_name, hints)
VALUES (
    'SELECT * FROM t t1, t t2, t t3, t t4 WHERE t1.col = ?;',
    '',
    'Leading(t4 t3 t2 t1)'
);
```

```
SELECT * FROM t t1, t t2, t t3, t t4 WHERE t1.col = ‘X’;

SELECT * FROM t t1, t t2, t t3, t t4 WHERE t1.col = ‘Y’;
```

3. SUMMARY

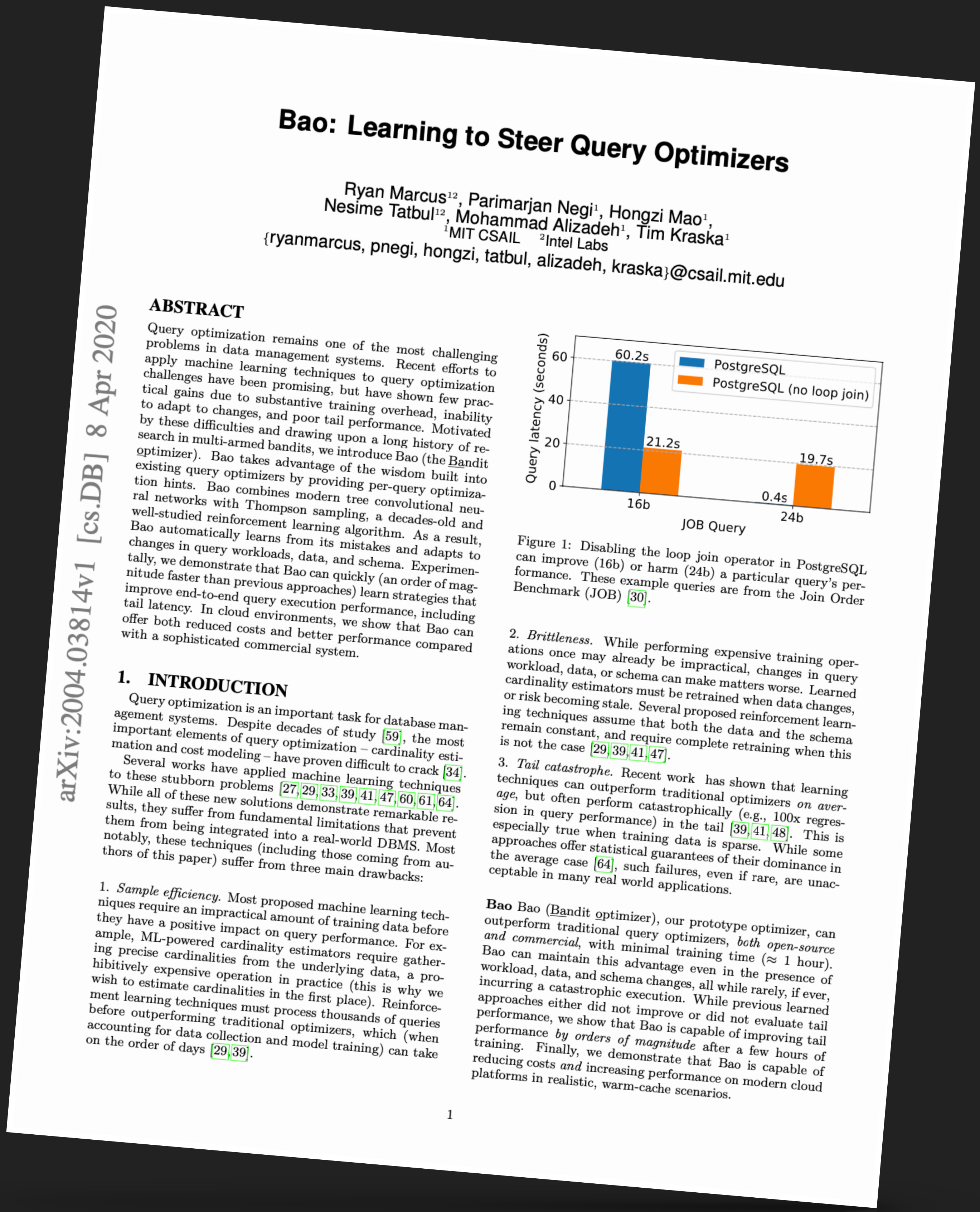
pg_hint_plan extension **allows us to** a) make hints **fine-grained,** b) change almost **anything** (cardinalities, join order, and physical operators) and c) **automate** part of the process by using hint_table.

But the implementation **has disadvantages**: we do it through over-costing alternatives and approach doesn't adapt to changes in the data

4. PRESENT AND FUTURE OF HINT BASED OPTIMIZATION

BAO (BANDIT OPTIMIZER)

We can set the best hintset **fully** automatically, **based** on the **experience**.



Bao: Learning to Steer Query Optimizers

Ryan Marcus[12], Parimarjan Negi[1], Hongzi Mao[1],
Nesime Tatbul[12], Mohammad Alizadeh[1], Tim Kraska[1]
[1]MIT CSAIL [2]Intel Labs
{ryanmarcus, pnegi, hongzi, tatbul, alizadeh, kraska}@csail.mit.edu

ABSTRACT

Query optimization remains one of the most challenging
problems in data management systems. Recent efforts to
apply machine learning techniques to query optimization
challenges have been promising, but have shown few prac-
tical gains due to substantive training overhead, inability
to adapt to changes, and poor tail performance. Motivated
by these difficulties and drawing upon a long history of re-
search in multi-armed bandits, we introduce Bao (the Bandit
optimizer). Bao takes advantage of the wisdom built into
existing query optimizers by providing per-query optimiza-
tion hints. Bao combines modern tree convolutional neu-
ral networks with Thompson sampling, a decades-old and
well-studied reinforcement learning algorithm. As a result,
Bao automatically learns from its mistakes and adapts to
changes in query workloads, data, and schema. Experimen-
tally, we demonstrate that Bao can quickly (an order of mag-
nitude faster than previous approaches) learn strategies that
improve end-to-end query execution performance, including
tail latency. In cloud environments, we show that Bao can
offer both reduced costs and better performance compared
with a sophisticated commercial system.

1. INTRODUCTION

Query optimization is an important task for database man-
agement systems. Despite decades of study [59], the most
important elements of query optimization – cardinality esti-
mation and cost modeling – have proven difficult to crack [34].
Several works have applied machine learning techniques
to these stubborn problems [27, 29, 33, 39, 41, 47, 60, 61, 64].
While all of these new solutions demonstrate remarkable re-
sults, they suffer from fundamental limitations that prevent
them from being integrated into a real-world DBMS. Most
notably, these techniques (including those coming from au-
thors of this paper) suffer from three main drawbacks:

1. *Sample efficiency.* Most proposed machine learning tech-
niques require an impractical amount of training data before
they have a positive impact on query performance. For ex-
ample, ML-powered cardinality estimators require gather-
ing precise cardinalities from the underlying data, a pro-
hibitively expensive operation in practice (this is why we
wish to estimate cardinalities in the first place). Reinforce-
ment learning techniques must process thousands of queries
before outperforming traditional optimizers, which (when
accounting for data collection and model training) can take
on the order of days [29, 39].

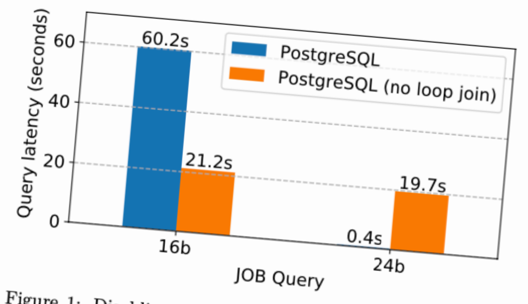


Figure 1: Disabling the loop join operator in PostgreSQL
can improve (16b) or harm (24b) a particular query's per-
formance. These example queries are from the Join Order
Benchmark (JOB) [30].

2. *Brittleness.* While performing expensive training oper-
ations once may already be impractical, changes in query
workload, data, or schema can make matters worse. Learned
cardinality estimators must be retrained when data changes,
or risk becoming stale. Several proposed reinforcement learn-
ing techniques assume that both the data and the schema
remain constant, and require complete retraining when this
is not the case [29, 39, 41, 47].

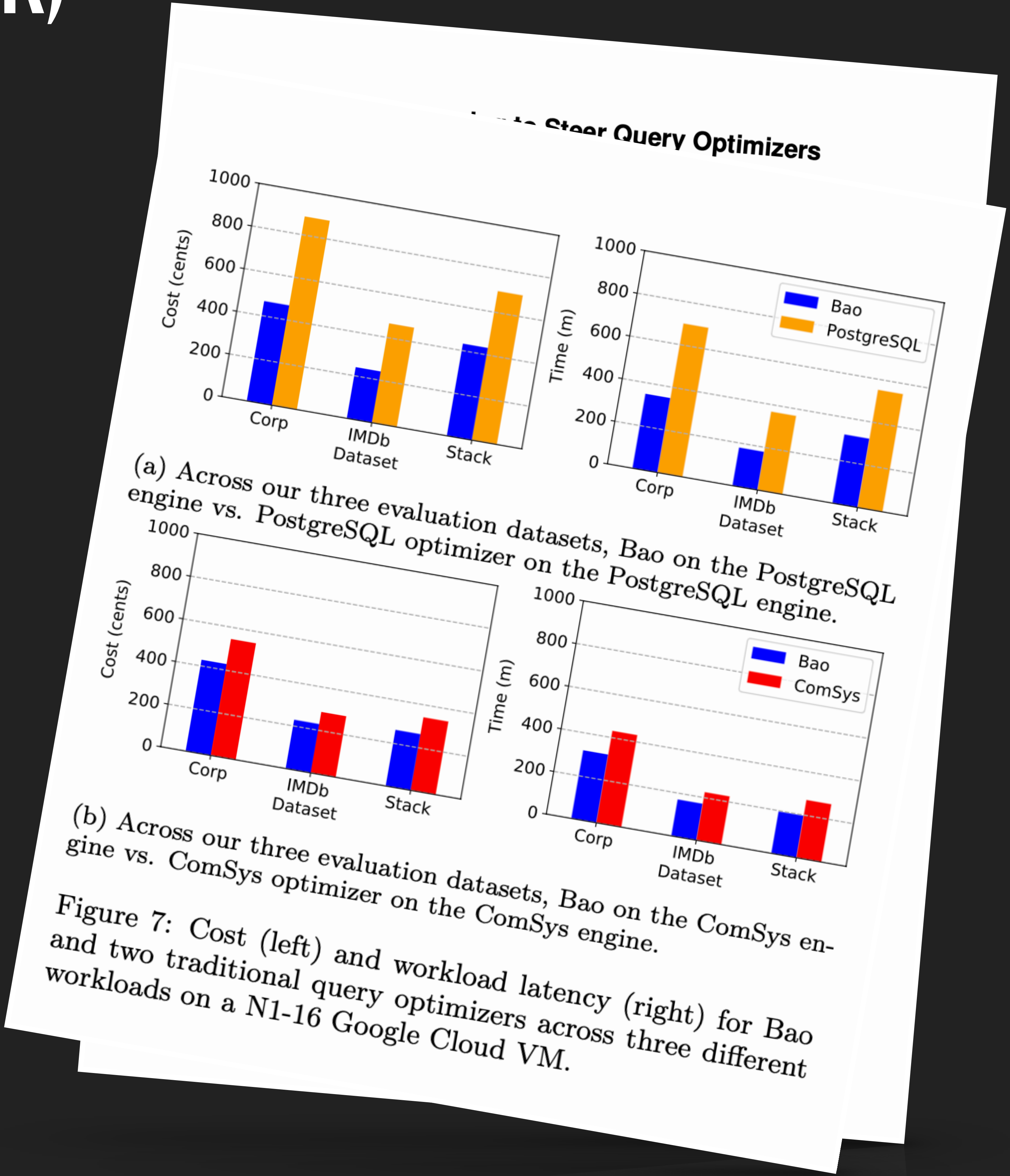
3. *Tail catastrophe.* Recent work has shown that learning
techniques can outperform traditional optimizers *on aver-
age*, but often perform catastrophically (e.g., 100x regres-
sion in query performance) in the tail [39, 41, 48]. This is
especially true when training data is sparse. While some
approaches offer statistical guarantees of their dominance in
the average case [64], such failures, even if rare, are unac-
ceptable in many real world applications.

Bao Bao (Bandit optimizer), our prototype optimizer, can
outperform traditional query optimizers, *both open-source
and commercial*, with minimal training time (≈ 1 hour).
Bao can maintain this advantage even in the presence of
workload, data, and schema changes, all while rarely, if ever,
incurring a catastrophic execution. While previous learned
approaches either did not improve or did not evaluate tail
performance, we show that Bao is capable of improving tail
performance *by orders of magnitude* after a few hours of
training. Finally, we demonstrate that Bao is capable of
reducing costs *and* increasing performance on modern cloud
platforms in realistic, warm-cache scenarios.

1

BAO (BANDIT OPTIMIZER)

Even **coarse-grained** hints can be used to improve performance



(a) Across our three evaluation datasets, Bao on the PostgreSQL engine vs. PostgreSQL optimizer on the PostgreSQL engine.

(b) Across our three evaluation datasets, Bao on the ComSys engine vs. ComSys optimizer on the ComSys engine.

Figure 7: Cost (left) and workload latency (right) for Bao and two traditional query optimizers across three different workloads on a N1-16 Google Cloud VM.

/* LET'S TRAIN IT */

BAO – HOW IT WORKS

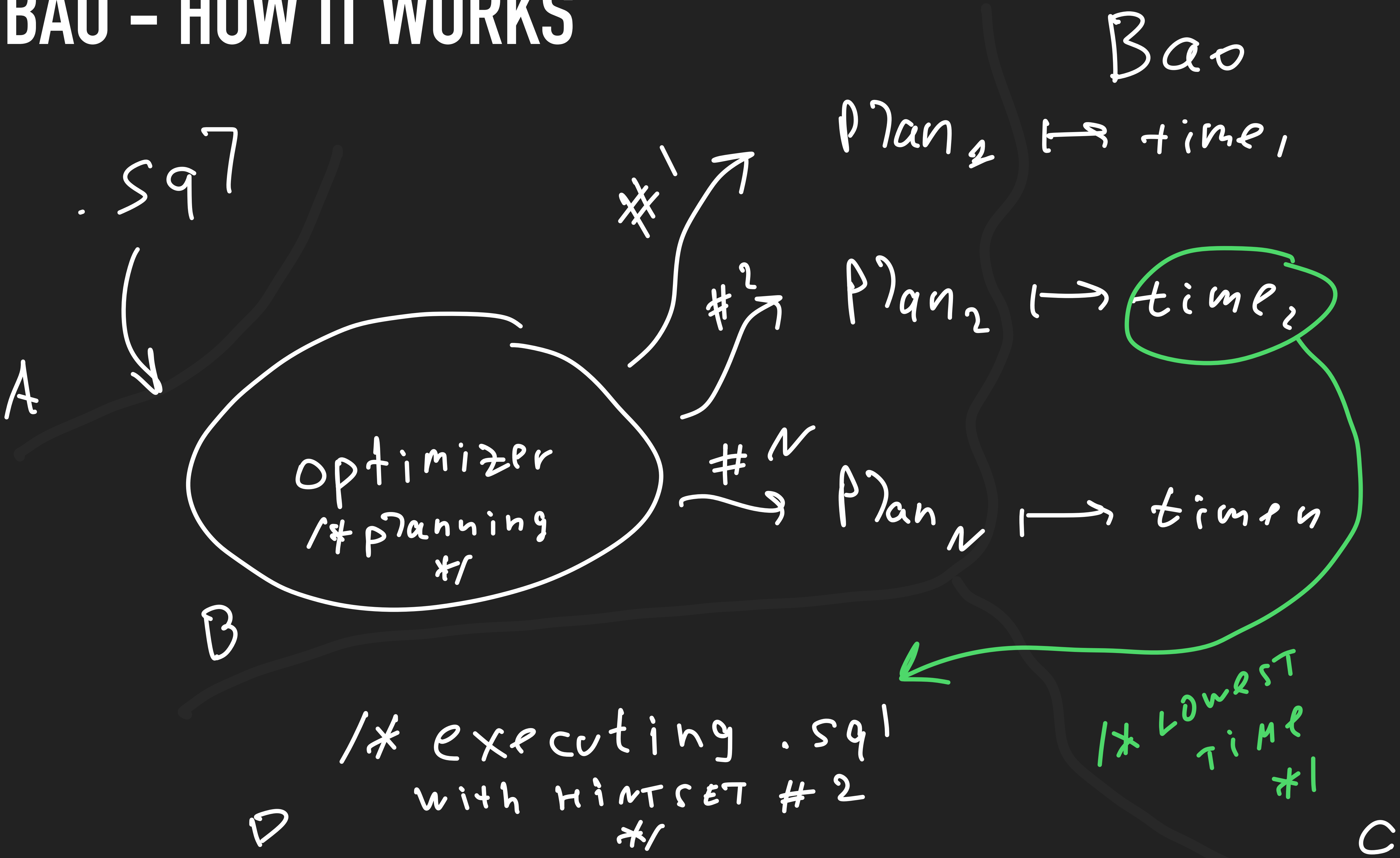
Experience

query	hintset	plan	time_ex
q17.sql	#24	...	1710 sec

$$\text{Bao}: \quad \text{plan} \longmapsto \text{time\_ex}$$

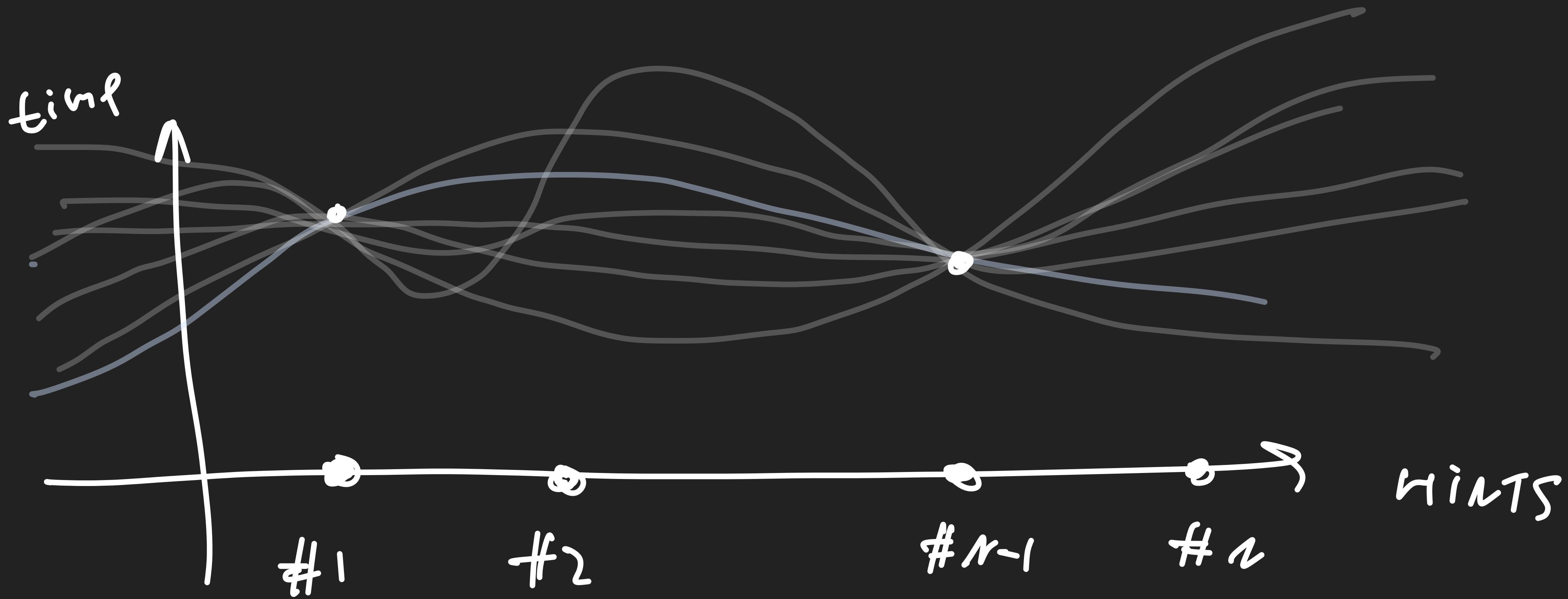
It uses the optimizer as a black box,
and just evaluates plans with its own eyes.

BAO - HOW IT WORKS



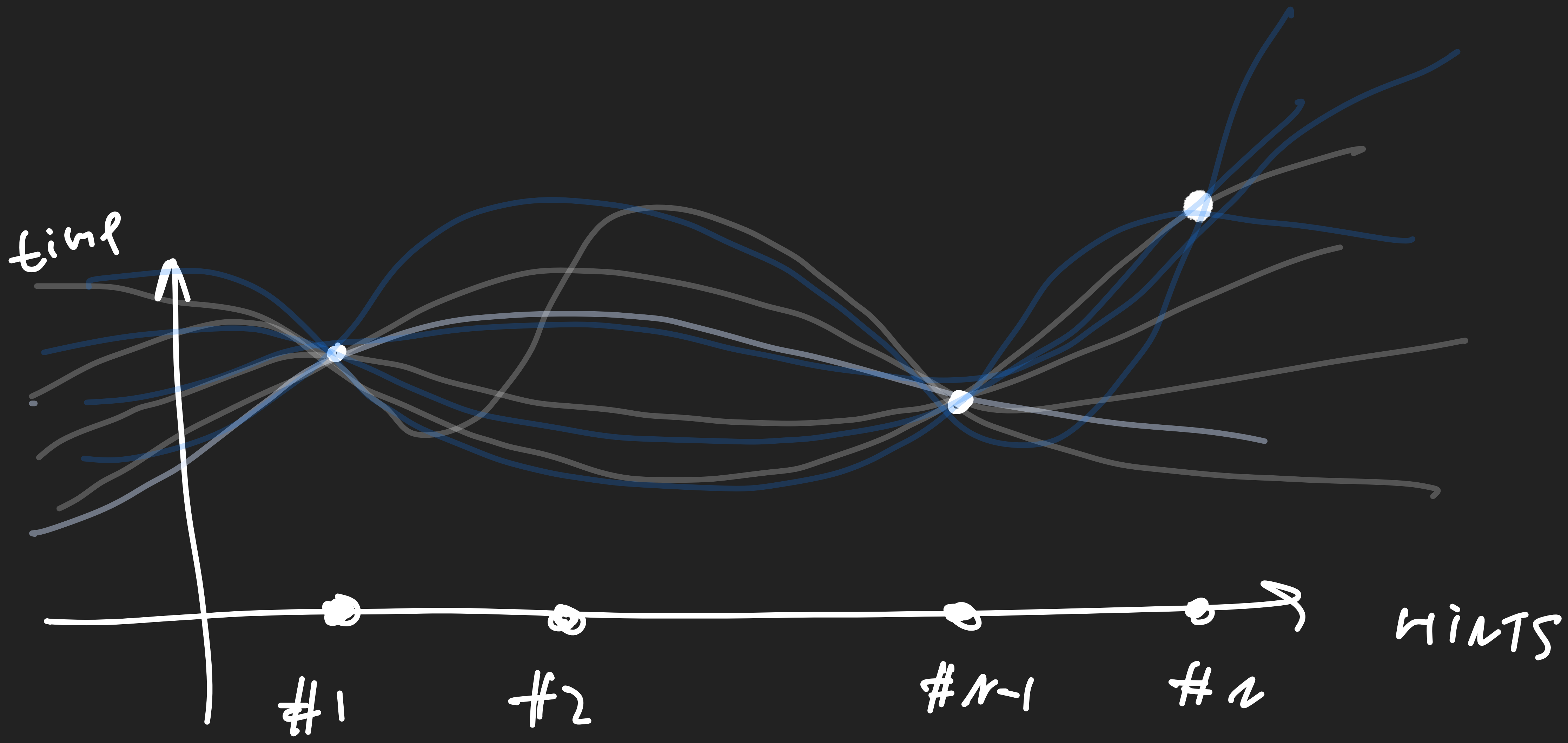
It uses the optimizer as a black box, and just evaluates plans with its own eyes.

4. SUMMARY



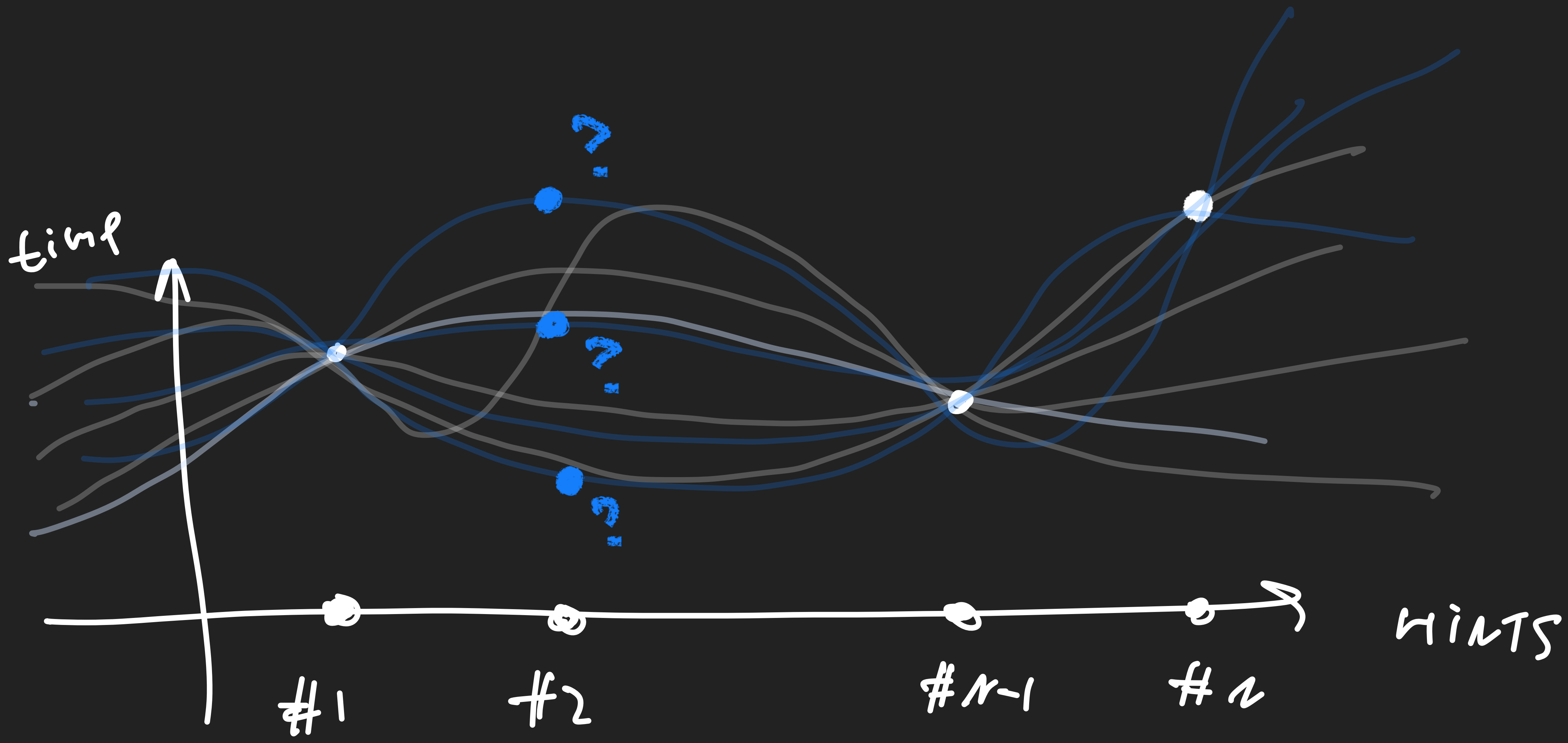
Bao needs to be trained in accurate manner.
It has to be able to say “this hintset is terrible”, otherwise the user will get a **regression**.

4. SUMMARY



Bao needs to be trained in accurate manner. It has to be able to say “this hintset is terrible”, otherwise the user will get a **regression**.

4. SUMMARY



Bao needs to be trained in accurate manner.
It has to be able to say "this hintset is terrible", otherwise the user will get a **regression**.

WHAT WE ARE DOING AT HUAWEI RRI

In my team, we are developing a fundamentally new model that
1) can guarantee the **absence of regressions**,
2) **learn automatically** and **faster**, and
3) **does not require** the implementation of a neural network in the kernel.

RESOURCES

Lecture Andy Pablo, CMU

Posts Franck Pachot, YugabyteDB

LectureMax Planck Institute for Informatucs

Bao: Learning to Steer Query Optimizers Marcus Ryan, MIT

How Good Are Query Optimizers, Really? Viktor Leis et al., TUM

Presentation Richard Guo, PGCON 2020

THANK YOU

Special thanks to Maksim Milyutin for help