TECHNISCHE UNIVERSITÄT DRESDEN

Dresden Database Research Group

Hands-On Material:
github.com/db-tu-dresden/SIGMOD25-OptimizerTutorial

Reproducible Prototyping of Query Optimizer Components

Rico Bergmann and Dirk Habich

Technische Universität Dresden, Germany

SIGMOD'25 Tutorial, June 27, 2025, Berlin, Germany

Presenters

Dresden Database Research Group

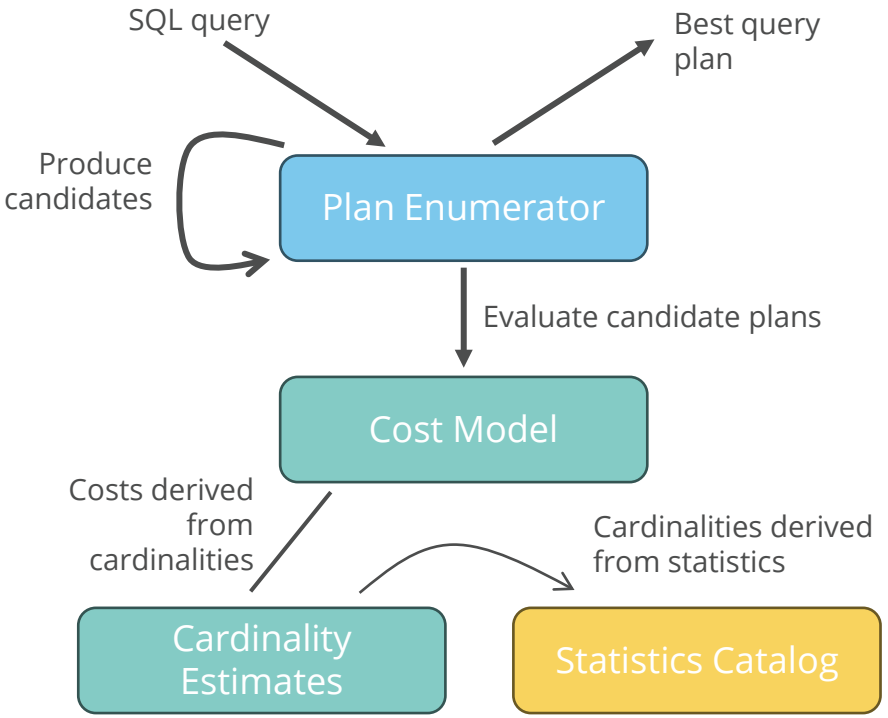
Rico Bergmann

- PhD student (2nd year)
- Technische Universität Dresden, Germany
- Research focus
  - Combining “traditional” and learned query optimization
  - Research tools for query optimization

Dirk Habich

- Associate Professor
- Technische Universität Dresden, Germany
- Research focus
  - Database systems on modern hardware
  - Query optimization
  - Habilitation in 2018 on „In-Memory Database Query Processing on Large-Scale Multiprocessor Systems“

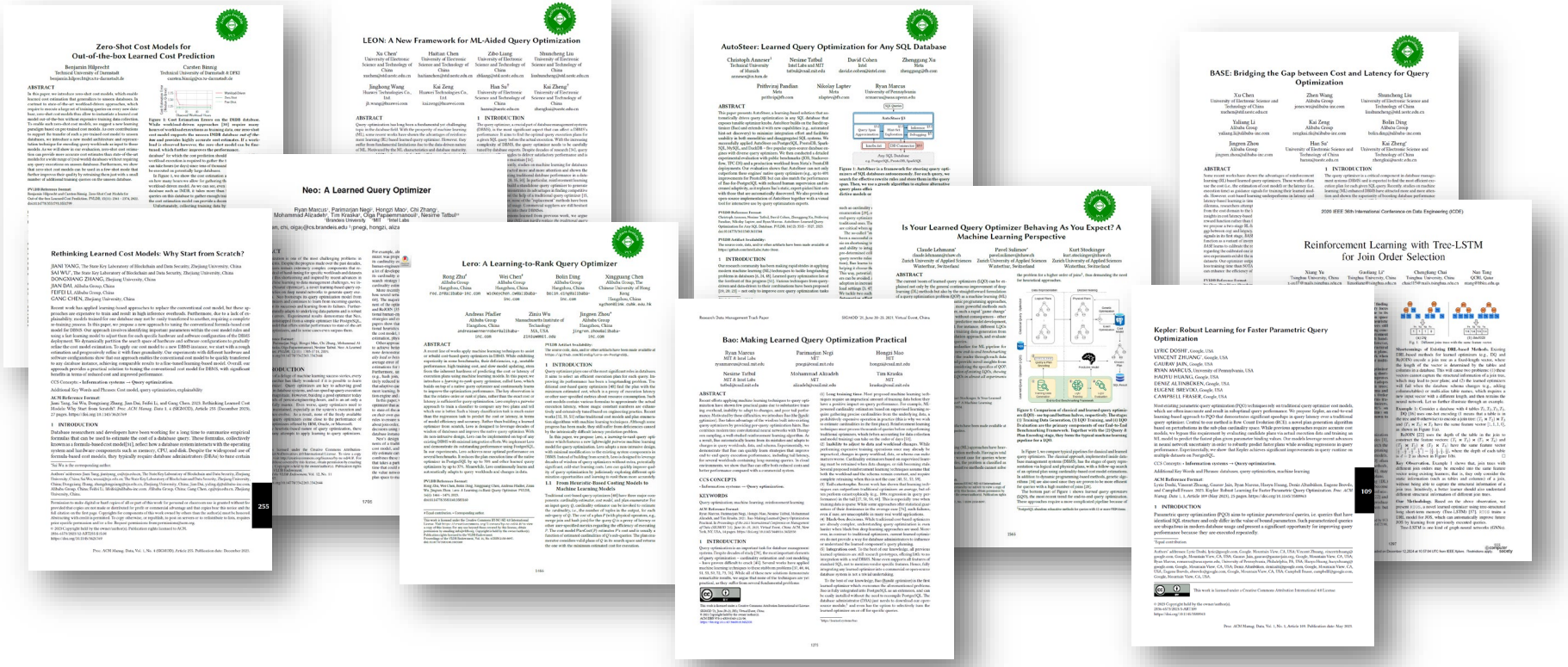
Textbook Query Optimizer



Current Research – Cardinality Estimation

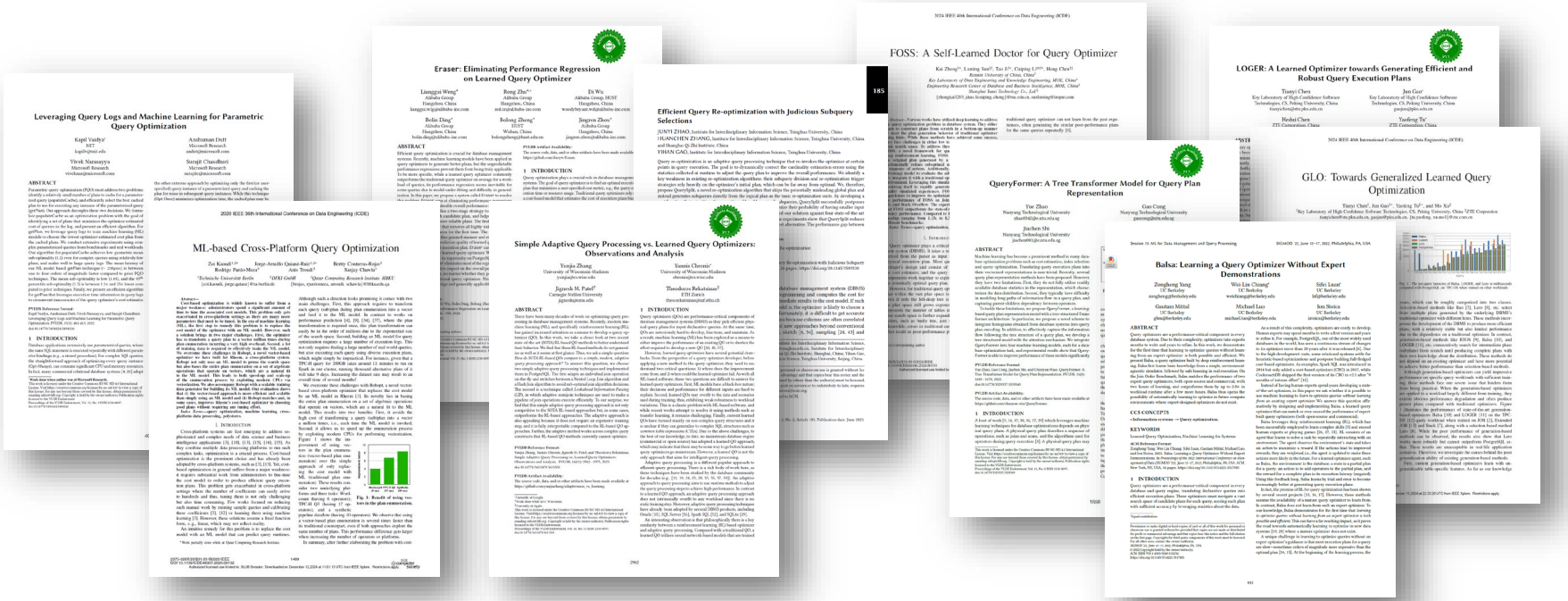
Dresden Database Research Group

Current Research – Cost Model



Current Research – Plan Generation

Dresden Database Research Group



Current Research

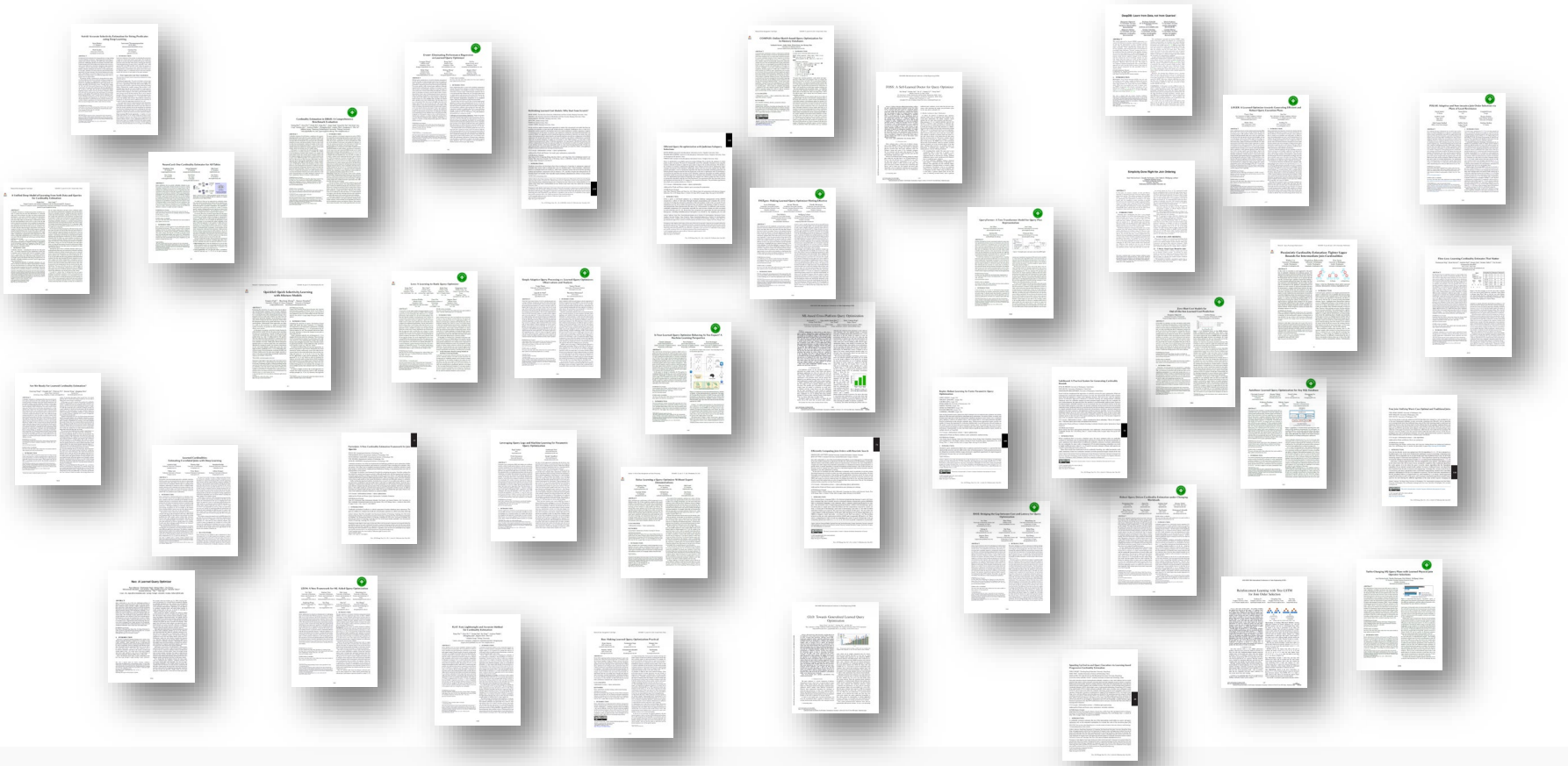
Dresden Database Research Group

Join Ordering

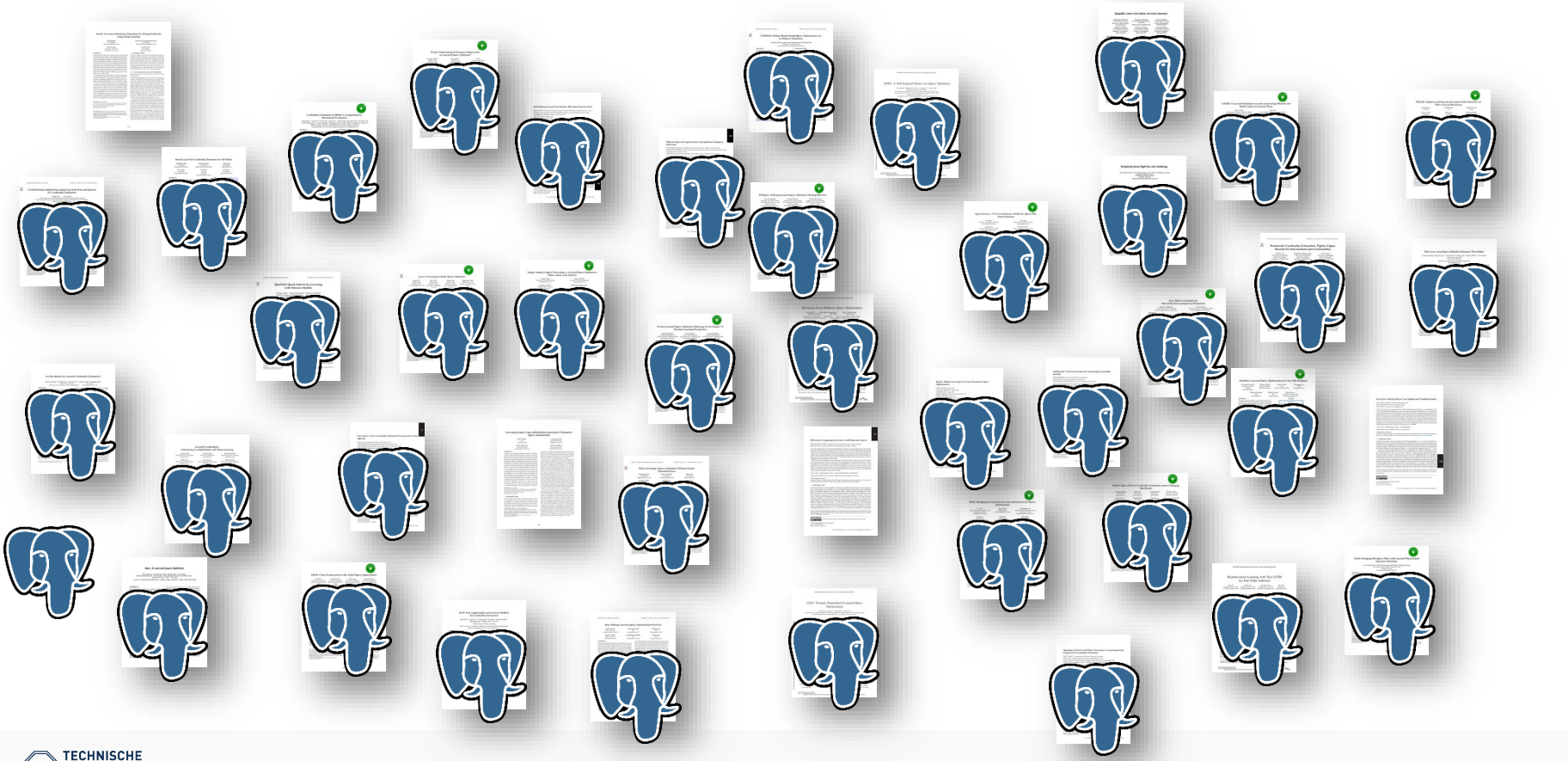
Operator Selection

Pessimistic Optimization

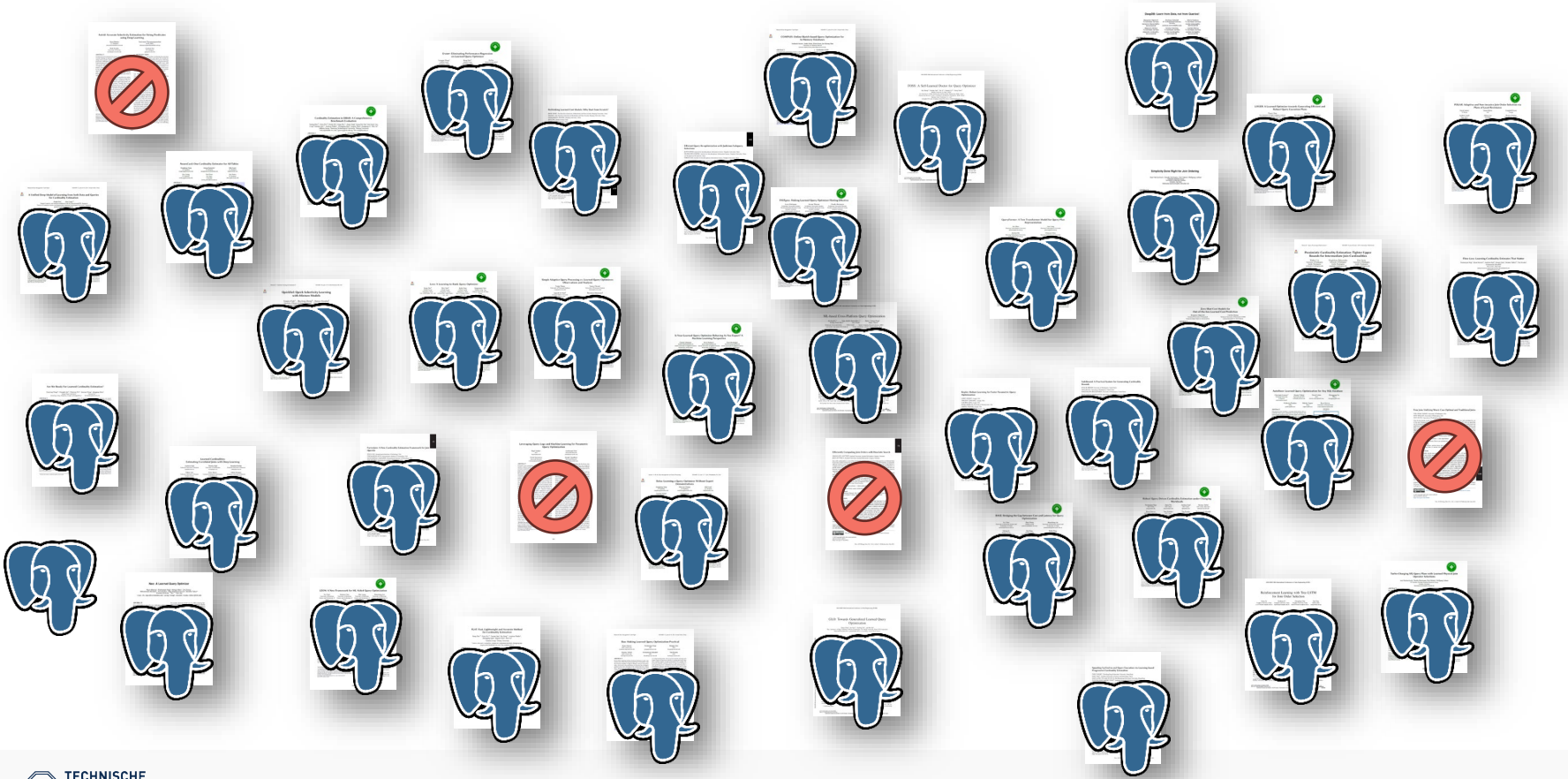
Current Research – Common Ground



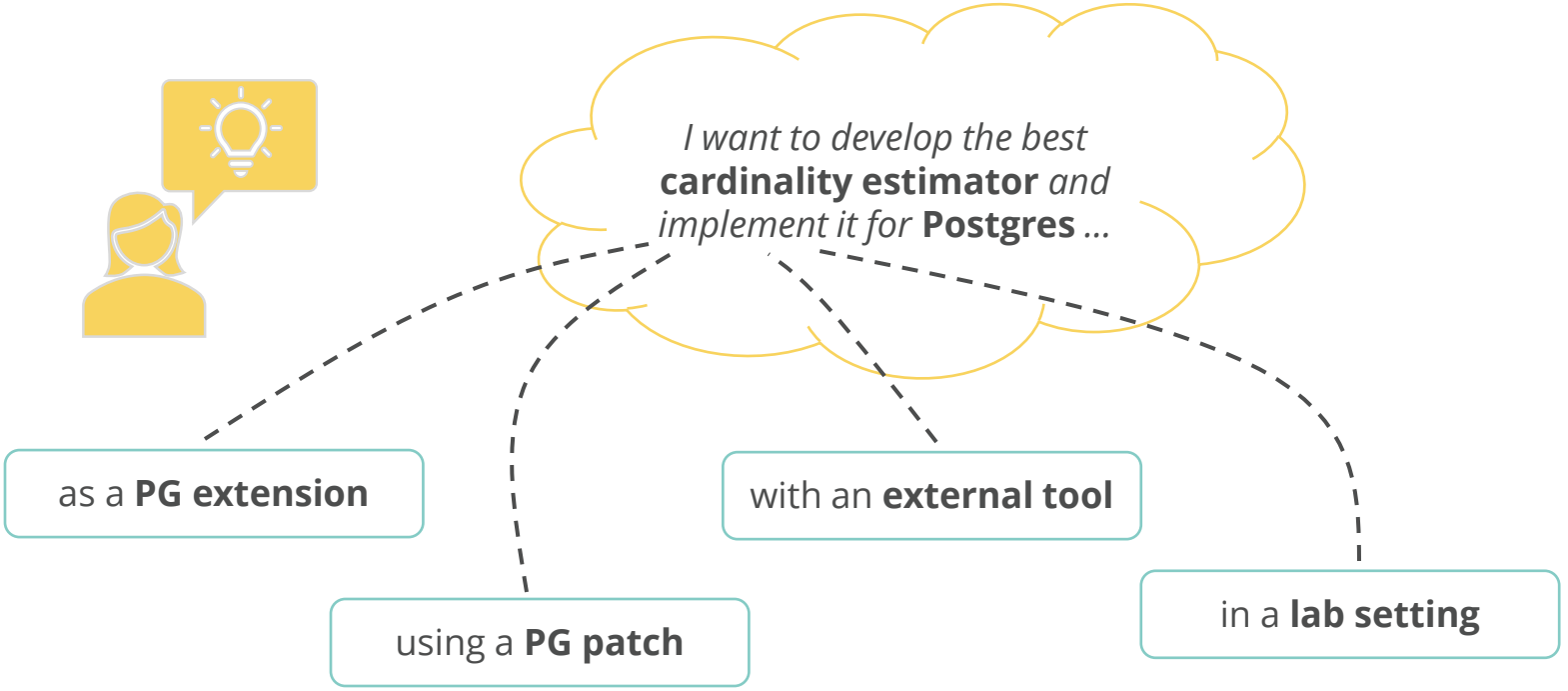
Current Research – Common Ground



Current Research – Common Ground



From Idea To Implementation



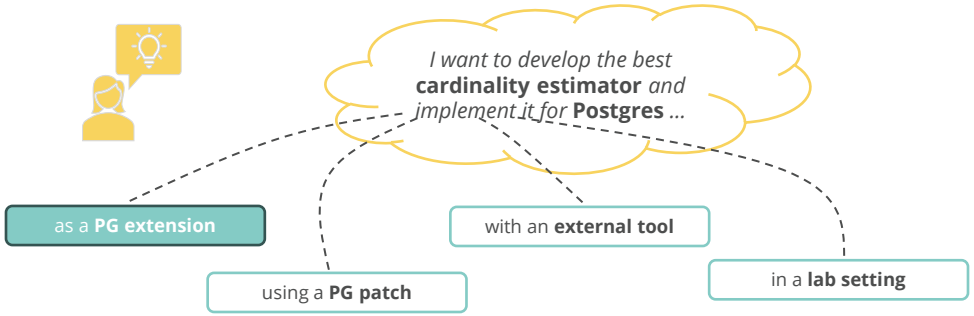
Implementation

Postgres Extension

- Postgres includes powerful extension mechanism
- Dynamically added to the server process
- Provides abstract functions to plug in custom functionality

Existing Hooks

- Complete optimization
  - `planner_hook`
- Join ordering + operator selection
  - `join_search_hook`
- Operator selection
  - `set_rel_pathlist_hook`
  - `set_join_pathlist_hook`
  - `create_upper_paths_hook`



Downsides

- Relies on available hooks
  - E.g., currently no hook for cardinality estimator or cost model
- Requires deep understanding of PG internals

```c
double max_column_frequency(PlannerInfo *root, RelOptInfo *rel, Var *column) {
    AttStatsSlot sslot;
    VariableStatData vardata;
    double max_freq;
    examine_variable(root, (Node *) column, 0, &vardata);
    get_attstatsslot(&sslot, vardata.statsTuple, STATISTIC_KIND_MCV, InvalidOid,
                     ATTSTATSSLOT_VALUES | ATTSTATSSLOT_NUMBERS);

    if (sslot.nvalues > 0)
        /* We found an MCV list, use it */
        max_freq = sslot.numbers[0] * rel->tuples; /* index 0 is the highest frequency */
    else {
        /* No MCV found, assume equal distribution */
        bool default_est; /* ignored, keep compiler quiet */
        double ndistinct = get_variable_numdistinct(&vardata, &default_est);
        if (ndistinct < 0) /* negative means ND is a fraction of the table size */
            ndistinct *= -1.0 * rel->tuples;
        max_freq = rel->tuples / ndistinct;
    }

    ReleaseVariableStats(vardata);
    return max_freq;
}
```

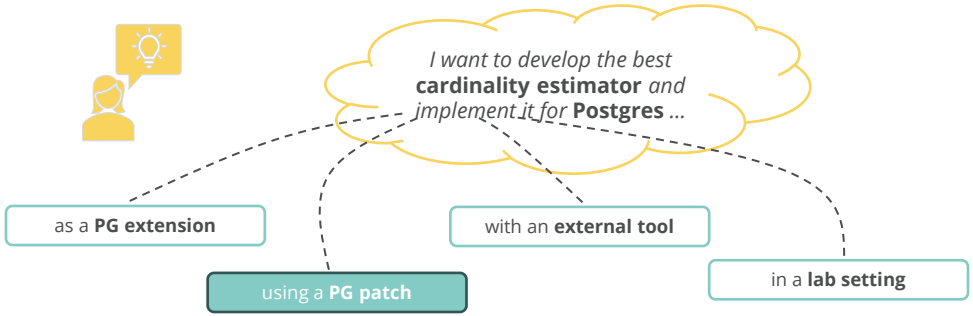
Implementation

Dresden Database Research Group

Postgres Patch

- Direct modifications of the PG source code
- Enables changes to query optimizer and execution engine

→ Adaptive query optimization becomes possible




```
8347     /*
8348      * clamp_row_est
8349     @@ -4733,6 +4758,10 @@ set_joinrel_size_estimates(PlannerInfo *root, RelOptInfo *rel,
8350                                                                             inner_rel->rows,
8351                                                                             sjinfo,
8352                                                                             restrictlist);
8353     +       if (enable_lero) {
8354     +        lero_pgsysml_set_joinrel_size_estimates(root, rel, outer_rel,
8355     +                                                inner_rel, sjinfo, restrictlist);
8356     +    }
8357      }
8358
8359      /*
```

Downsides

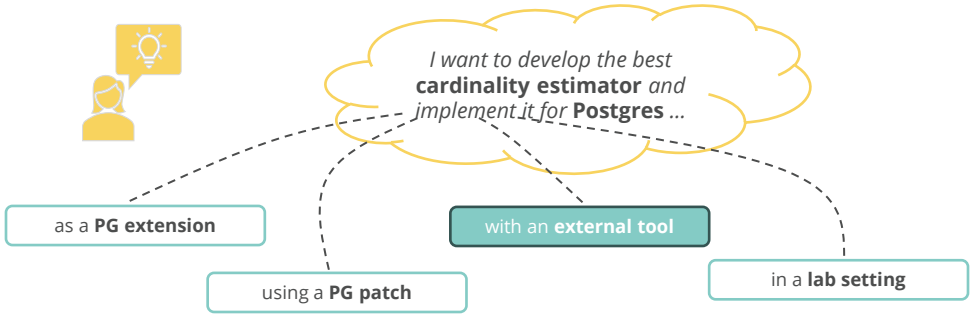
- Requires deep understanding of PG internals
- Changes are hard to track
- Specific to (minor) releases of PG

Implementation

Dresden Database Research Group

pg_hint_plan

- Extension to embed optimizer decisions in comments
- Hints can be generated through arbitrary software
- Allows good tooling integration (e.g., ML frameworks)




```
/*+ NestLoop(t mi) IndexScan(mi) */
SELECT count(*)
FROM title t, movie_info mi
WHERE t.id = mi.movie_id
    AND t.production_year > 2010
```

Downsides

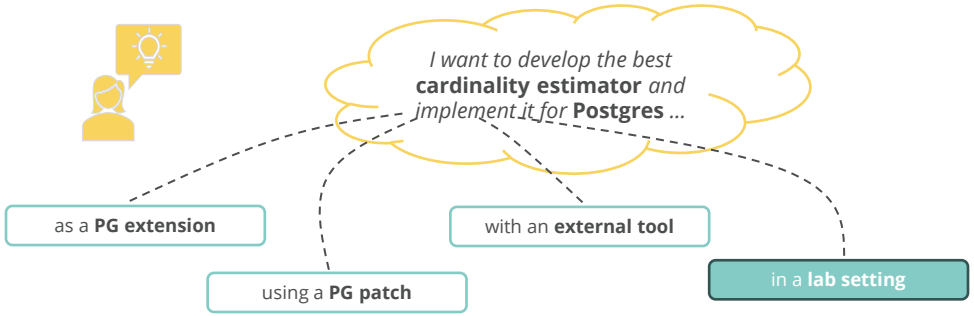
- Opaque implementation – no standardized generation of hints
- Limited to the available hints
  - No cardinalities for base tables
  - No parallel workers

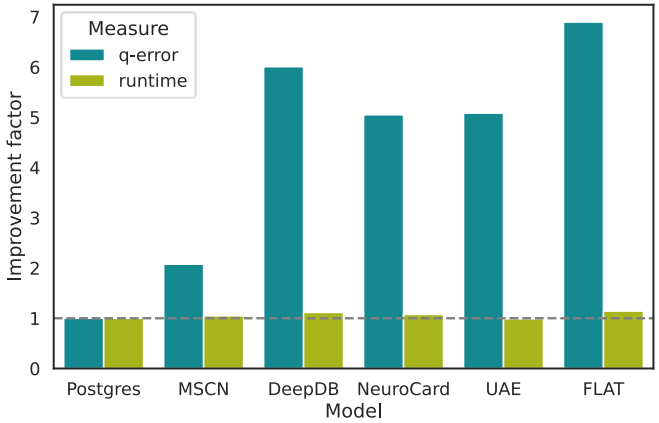
Implementation

Dresden Database Research Group

Artificial Setting

- Don't implement for a real database
- Instead, use a "laboratory setting"
- E.g., measure q-error instead of runtime



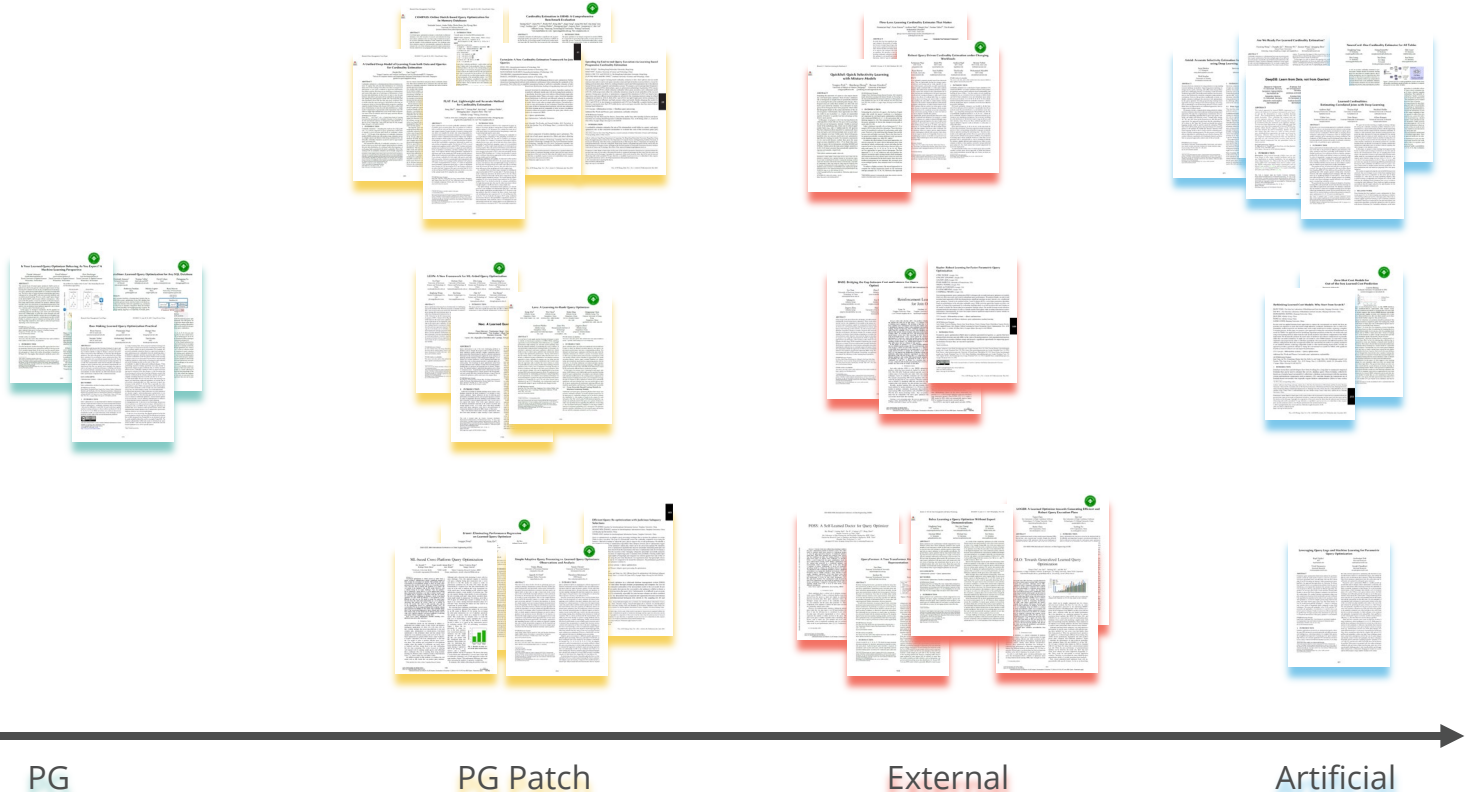


Downsides

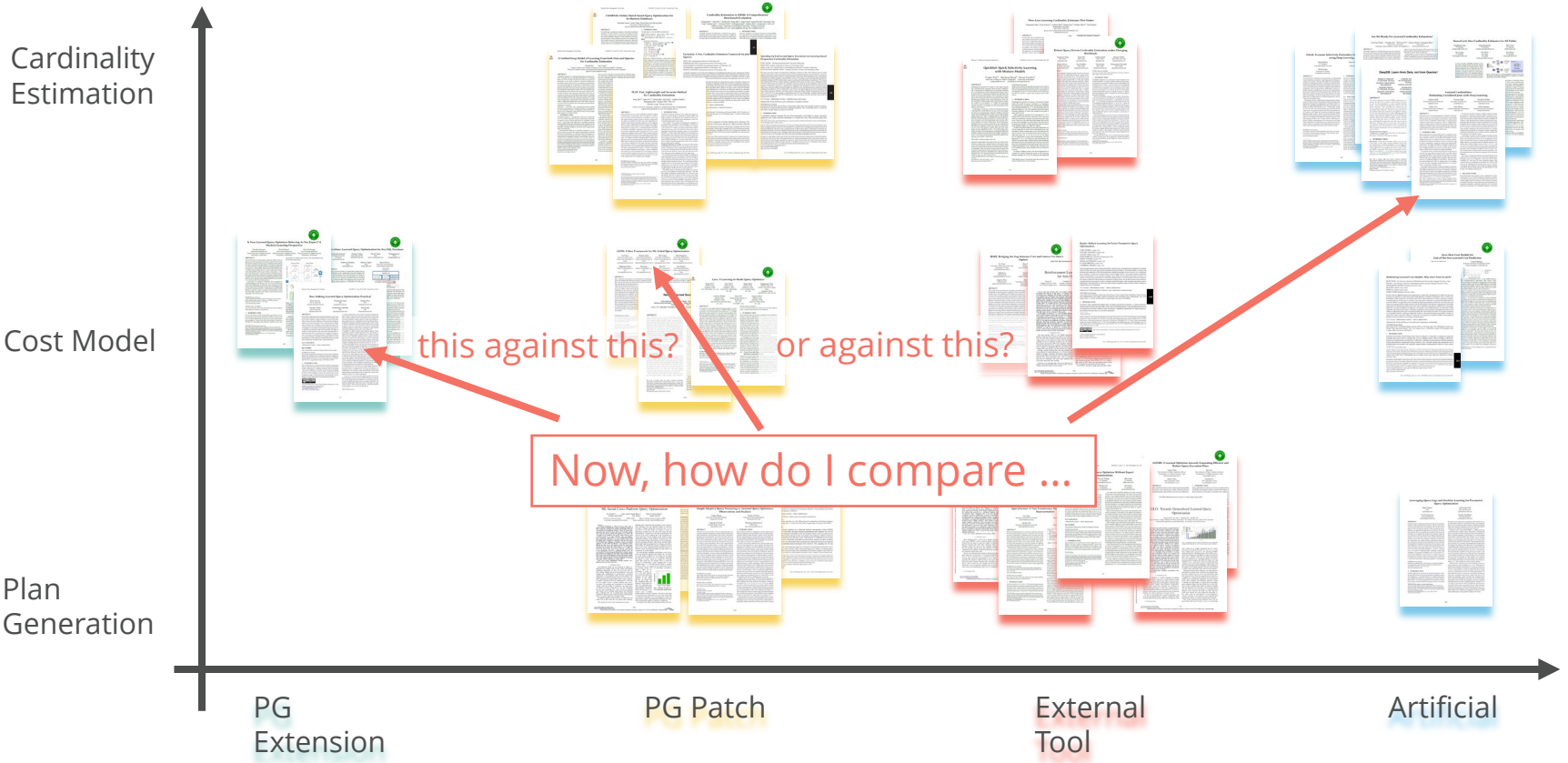
- No real-world execution – how to ensure correlation?

Numbers based on Han et al.:
Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation (VLDB'21)

Current Research Revisited



Current Research Revisited



Let‘s compare with BAO (Marcus et al.)

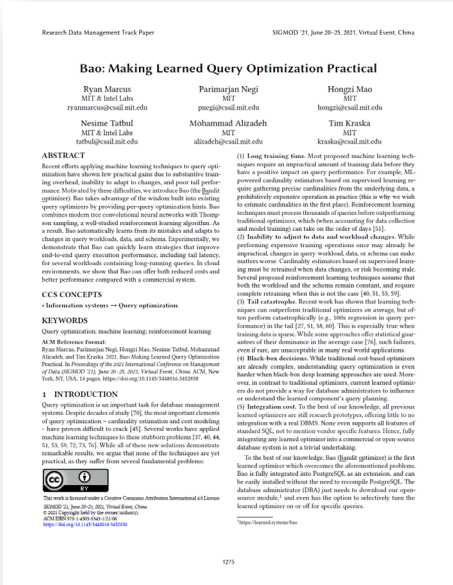
Dresden Database Research Group

BAO in a nutshell

- Uses search space restrictions to obtain different query plans
- Estimates the plan quality via a learned model
- Selects the (estimated) best plan for execution
- SIGMOD best paper award 2021

Implementation details

- Implemented as a PG extension
- Communicates with a Python server
- Based on PG v12.4



Research Data Management Track Paper SIGMOD '21, June 20–25, 2021, Virtual Event, China

Bao: Making Learned Query Optimization Practical

Ryan Marcus, MIT & Intel Labs, ryanmarcus@csail.mit.edu
Parimarjan Negi, MIT, pnegi@csail.mit.edu
Hongzi Mao, MIT, hongzi@csail.mit.edu
Nesime Tatbul, MIT & Intel Labs, tatbul@csail.mit.edu
Mohammad Alizadeh, MIT, alizadeh@csail.mit.edu
Tim Kraska, MIT, kraska@csail.mit.edu

ABSTRACT

Recent efforts applying machine learning techniques to query optimization have shown few practical gains due to substantive training overhead, inability to adapt to changes, and poor tail performance. Motivated by these difficulties, we introduce Bao (the Bandit optimizer). Bao takes advantage of the wisdom built into existing query optimizers by providing per-query optimization hints. Bao combines modern tree convolutional neural networks with Thompson sampling, a well-studied reinforcement learning algorithm. As a result, Bao automatically learns from its mistakes and adapts to changes in query workloads, data, and schema. Experimentally, we demonstrate that Bao can quickly learn strategies that improve end-to-end query execution performance, including tail latency, for several workloads containing long-running queries. In cloud environments, we show that Bao can offer both reduced costs and better performance compared with a commercial system.

CCS CONCEPTS

• Information systems → Query optimization.

KEYWORDS

Query optimization; machine learning; reinforcement learning

ACM Reference Format:

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452838

1 INTRODUCTION

Query optimization is an important task for database management systems. Despite decades of study [70], the most important elements of query optimization – cardinality estimation and cost modeling – have proven difficult to crack [45]. Several works have applied machine learning techniques to these stubborn problems [37, 40, 44, 51, 53, 59, 72, 73, 76]. While all of these new solutions demonstrate remarkable results, we argue that none of the techniques are yet practical, as they suffer from several fundamental problems:

(1) **Long training time.** Most proposed machine learning techniques require an impractical amount of training data before they have a positive impact on query performance. For example, ML-powered cardinality estimators based on supervised learning require gathering precise cardinalities from the underlying data, a prohibitively expensive operation in practice (this is why we wish to estimate cardinalities in the first place). Reinforcement 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 [51].

(2) **Inability to adjust to data and workload changes.** While performing expensive training operations once may already be impractical, changes in query workload, data, or schema can make matters worse. Cardinality estimators based on supervised learning must be retrained when data changes, or risk becoming stale. Several proposed reinforcement learning techniques assume that both the workload and the schema remain constant, and require complete retraining when this is not the case [40, 51, 53, 59].

(3) **Tail catastrophe.** Recent work has shown that learning techniques can outperform traditional optimizers on average, but often perform catastrophically (e.g., 100x regression in query performance) in the tail [27, 51, 58, 60]. This is especially true when training data is sparse. While some approaches offer statistical guarantees of their dominance in the average case [76], such failures, even if rare, are unacceptable in many real world applications.

(4) **Black-box decisions.** While traditional cost-based optimizers are already complex, understanding query optimization is even harder when black-box deep learning approaches are used. Moreover, in contrast to traditional optimizers, current learned optimizers do not provide a way for database administrators to influence or understand the learned component's query planning.

(5) **Integration cost.** To the best of our knowledge, all previous learned optimizers are still research prototypes, offering little to no integration with a real DBMS. None even supports all features of standard SQL, not to mention vendor specific features. Hence, fully integrating any learned optimizer into a commercial or open-source database system is not a trivial undertaking.

To the best of our knowledge, Bao (bandit optimizer) is the first learned optimizer which overcomes the aforementioned problems. Bao is fully integrated into PostgreSQL as an extension, and can be easily installed without the need to recompile PostgreSQL. The database administrator (DBA) just needs to download our open-source module,[1] and even has the option to selectively turn the learned optimizer on or off for specific queries.

[1] https://learned-systems/bao

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3452838

Marcus et al.: “BAO: Making Learned Query Optimization Practical” (SIGMOD’2021)

Let‘s compare with BAO (Marcus et al.)

Dresden Database Research Group

```
 ~/pg-17/contrib/bao/pg_extension [master] $ make && make install
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Werror=vla
-Wendif-labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -Wcast-function-type
-Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv -fexcess-
precision=standard -Wno-format-truncation -Wno-stringop-truncation -O2 -fPIC -
fvisibility=hidden -I. -I./ -I/home/rico/pg-17/build/include/postgresql/server -
I/home/rico/pg-17/build/include/postgresql/internal    -D_GNU_SOURCE   -c -o main.o
main.c
In file included from main.c:9:
bao_bufferstate.h:13:10: fatal error: utils/relfilenodemap.h: No such file or directory
   13 | #include "utils/relfilenodemap.h"
      |          ^~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
make: *** [<builtin>: main.o] Error 1
```

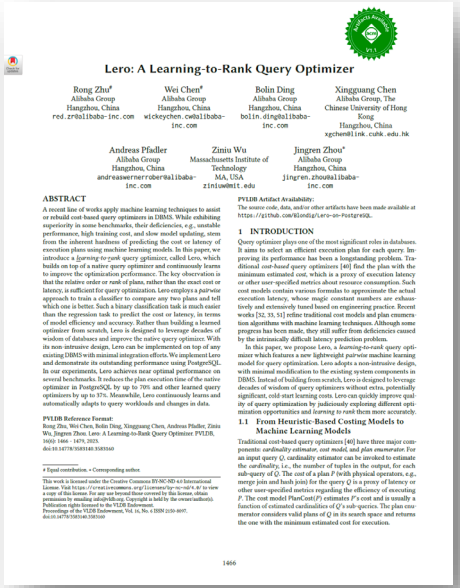
We can’t reproduce on PG 17!

OK, what about Lero? (Zhu et al.)

Dresden Database Research Group

Lero in a nutshell

- Uses modified cardinality estimates to obtain different query plans
- Ranks the plans using a BAO-style model
- Selects the (ranked) best plan for execution



Lero: A Learning-to-Rank Query Optimizer

Rong Zhu* Alibaba Group Hangzhou, China red.zr@alibaba-inc.com

Wei Chen* Alibaba Group Hangzhou, China wickeychen.cwa@alibaba-inc.com

Bolin Ding Alibaba Group Hangzhou, China bolin.ding@alibaba-inc.com

Xingguang Chen Alibaba Group, The Chinese University of Hong Kong Hangzhou, China xgchen@link.cuhk.edu.hk

Andreas Pfadler Alibaba Group Hangzhou, China andreaswernerrober@alibaba-inc.com

Ziniu Wu Massachusetts Institute of Technology MA, USA ziniuw@mit.edu

Jingren Zhou* Alibaba Group Hangzhou, China jingren.zhou@alibaba-inc.com

ABSTRACT

A recent line of works apply machine learning techniques to assist or rebuild cost-based query optimizers in DBMS. While exhibiting superiority in some benchmarks, their deficiencies, e.g., unstable performance, high training cost, and slow model updating, stem from the inherent hardness of predicting the cost or latency of execution plans using machine learning models. In this paper, we introduce a *learning-to-rank* query optimizer, called Lero, which builds on top of a native query optimizer and continuously learns to improve the optimization performance. The key observation is that the relative order or rank of plans, rather than the exact cost or latency, is sufficient for query optimization. Lero employs a *pairwise* approach to train a classifier to compare any two plans and tell which one is better. Such a binary classification task is much easier than the regression task to predict the cost or latency, in terms of model efficiency and accuracy. Rather than building a learned optimizer from scratch, Lero is designed to leverage decades of wisdom of databases and improve the native query optimizer. With its non-intrusive design, Lero can be implemented on top of any existing DBMS with minimal integration efforts. We implement Lero and demonstrate its outstanding performance using PostgreSQL. In our experiments, Lero achieves near optimal performance on several benchmarks. It reduces the plan execution time of the native optimizer in PostgreSQL by up to 70% and other learned query optimizers by up to 37%. Meanwhile, Lero continuously learns and automatically adapts to query workloads and changes in data.

PVLDB Reference Format:

Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, Jingren Zhou. Lero: A Learning-to-Rank Query Optimizer. PVLDB, 16(6): 1466 - 1479, 2023.

doi:10.14778/3583140.3583160

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/Klaebing/Lero-on-PostgreSQL.

1 INTRODUCTION

Query optimizer plays one of the most significant roles in databases. It aims to select an efficient execution plan for each query. Improving its performance has been a longstanding problem. Traditional *cost-based* query optimizers [40] find the plan with the minimum estimated cost, which is a proxy of execution latency or other user-specified metrics about resource consumption. Such cost models contain various formulas to approximate the actual execution latency, whose magic constant numbers are exhaustively and extensively tuned based on engineering practice. Recent works [32, 33, 51] refine traditional cost models and plan enumeration algorithms with machine learning techniques. Although some progress has been made, they still suffer from deficiencies caused by the intrinsically difficult latency prediction problem.

In this paper, we propose Lero, a *learning-to-rank* query optimizer which features a new *lightweight pairwise* machine learning model for query optimization. Lero adopts a non-intrusive design, with minimal modification to the existing system components in DBMS. Instead of building from scratch, Lero is designed to leverage decades of wisdom of query optimizers without extra, potentially significant, cold-start learning costs. Lero can quickly improve quality of query optimization by judiciously exploring different optimization opportunities and *learning to rank* them more accurately.

1.1 From Heuristic-Based Costing Models to Machine Learning Models

Traditional cost-based query optimizers [40] have three major components: *cardinality estimator*, *cost model*, and *plan enumerator*. For an input query Q, cardinality estimator can be invoked to estimate the *cardinality*, i.e., the number of tuples in the output, for each sub-query of Q. The cost of a plan P (with physical operators, e.g., merge join and hash join) for the query Q is a proxy of latency or other user-specified metrics regarding the efficiency of executing P. The cost model PlanCost(P) estimates P's cost and is usually a function of estimated cardinalities of Q's sub-queries. The plan enumerator considers valid plans of Q in its search space and returns the one with the minimum estimated cost for execution.

1466

Implementation details

- Implemented as a PG patch
- Communicates with a Python server
- Based on PG v13.1

Zhu et al.: “Lero: A Learning-to-Rank Query Optimizer” (VLDB’2023)

OK, what about Lero? (Zhu et al.)

Dresden Database Research Group

```
 ~/pg-17 [master] $ git apply contrib/lero/0001-init-lero.patch
error: patch failed: src/backend/Makefile:19
error: src/backend/Makefile: patch does not apply
error: patch failed: src/backend/optimizer/path/costsize.c:107
error: src/backend/optimizer/path/costsize.c: patch does not apply
error: patch failed: src/backend/optimizer/plan/planner.c:64
error: src/backend/optimizer/plan/planner.c: patch does not apply
error: patch failed: src/backend/utils/misc/guc.c:98
error: src/backend/utils/misc/guc.c: patch does not apply
error: patch failed: src/include/optimizer/cost.h:67
error: src/include/optimizer/cost.h: patch does not apply
```

We can’t reproduce on PG 17!

Can we at least use MSCN? (Kipf et al.)

Dresden Database Research Group

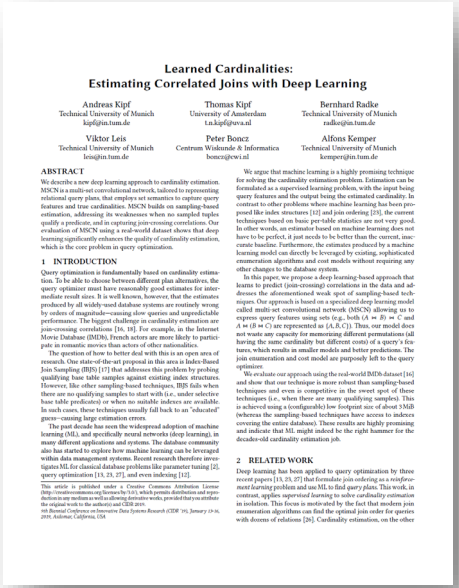
Artificial evaluation

=

No implementation on database system

=

We can’t reproduce on PG 17!



Learned Cardinalities:
Estimating Correlated Joins with Deep Learning

Andreas Kipf
Technical University of Munich
kipf@in.tum.de

Thomas Kipf
University of Amsterdam
t.n.kipf@uva.nl

Bernhard Radke
Technical University of Munich
radke@in.tum.de

Viktor Leis
Technical University of Munich
leis@in.tum.de

Peter Boncz
Centrum Wiskunde & Informatica
boncz@cwi.nl

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

ABSTRACT

We describe a new deep learning approach to cardinality estimation. MSCN is a multi-set convolutional network, tailored to representing relational query plans, that employs set semantics to capture query features and true cardinalities. MSCN builds on sampling-based estimation, addressing its weaknesses when no sampled tuples qualify a predicate, and in capturing join-crossing correlations. Our evaluation of MSCN using a real-world dataset shows that deep learning significantly enhances the quality of cardinality estimation, which is the core problem in query optimization.

1 INTRODUCTION

Query optimization is fundamentally based on cardinality estimation. To be able to choose between different plan alternatives, the query optimizer must have reasonably good estimates for intermediate result sizes. It is well known, however, that the estimates produced by all widely-used database systems are routinely wrong by orders of magnitude—causing slow queries and unpredictable performance. The biggest challenge in cardinality estimation are join-crossing correlations [16, 18]. For example, in the Internet Movie Database (IMDb), French actors are more likely to participate in romantic movies than actors of other nationalities.

The question of how to better deal with this is an open area of research. One state-of-the-art proposal in this area is Index-Based Join Sampling (IBJS) [17] that addresses this problem by probing qualifying base table samples against existing index structures. However, like other sampling-based techniques, IBJS fails when there are no qualifying samples to start with (i.e., under selective base table predicates) or when no suitable indexes are available. In such cases, these techniques usually fall back to an "educated" guess—causing large estimation errors.

The past decade has seen the widespread adoption of machine learning (ML), and specifically neural networks (deep learning), in many different applications and systems. The database community also has started to explore how machine learning can be leveraged within data management systems. Recent research therefore investigates ML for classical database problems like parameter tuning [2], query optimization [13, 23, 27], and even indexing [12].

9th Biennial Conference on Innovative Data Systems Research (CIDR '19), January 13-16, 2019, Asilomar, California, USA

We argue that machine learning is a highly promising technique for solving the cardinality estimation problem. Estimation can be formulated as a supervised learning problem, with the input being query features and the output being the estimated cardinality. In contrast to other problems where machine learning has been proposed like index structures [12] and join ordering [23], the current techniques based on basic per-table statistics are not very good. In other words, an estimator based on machine learning does not have to be perfect, it just needs to be better than the current, inaccurate baseline. Furthermore, the estimates produced by a machine learning model can directly be leveraged by existing, sophisticated enumeration algorithms and cost models without requiring any other changes to the database system.

In this paper, we propose a deep learning-based approach that learns to predict (join-crossing) correlations in the data and addresses the aforementioned weak spot of sampling-based techniques. Our approach is based on a specialized deep learning model called multi-set convolutional network (MSCN) allowing us to express query features using sets (e.g., both $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ are represented as $\{A, B, C\}$). Thus, our model does not waste any capacity for memorizing different permutations (all having the same cardinality but different costs) of a query's features, which results in smaller models and better predictions. The join enumeration and cost model are purposely left to the query optimizer.

We evaluate our approach using the real-world IMDb dataset [16] and show that our technique is more robust than sampling-based techniques and even is competitive in the sweet spot of these techniques (i.e., when there are many qualifying samples). This is achieved using a (configurable) low footprint size of about 5 MiB (whereas the sampling-based techniques have access to indexes covering the entire database). These results are highly promising and indicate that ML might indeed be the right hammer for the decades-old cardinality estimation job.

2 RELATED WORK

Deep learning has been applied to query optimization by three recent papers [13, 23, 27] that formulate join ordering as a *reinforcement learning* problem and use ML to find query plans. This work, in contrast, applies *supervised learning* to solve *cardinality estimation* in isolation. This focus is motivated by the fact that modern join enumeration algorithms can find the optimal join order for queries with dozens of relations [26]. Cardinality estimation, on the other

Kipf et al.: “Learned Cardinalities: Estimating Correlated Joins with Deep Learning” (CIDR’2019)

Optimization Frameworks

Dresden Database Research Group

Could frameworks help us?

Ideally, we would like:

○	Broad support for different optimization strategies
○	Transparent implementation of prototypes
○	Query execution on Postgres
○	Easy comparison of different prototypes

Existing Frameworks

Research Frameworks

Dresden Database Research Group

Apache Calcite

https://calcite.apache.org/

- Java-based framework to implement entire optimizer
- Community-driven development
- Deep integration in Java/Apache ecosystem
- Allows customization of a traditional optimizer model

✓	Broad support for different optimization strategies
✓	Transparent implementation of prototypes
✗	Execution on Postgres
✓	Easy comparison of different prototypes

Research Frameworks

Dresden Database Research Group

LingoDB

https://www.lingo-db.com/

- Research system focused on MLIR-based query processing
- Developed at TUM
- Optimizer implemented as LLVM compiler passes

−	Broad support for different optimization strategies
✓	Transparent implementation of prototypes
✗	Execution on Postgres
−	Easy comparison of different prototypes

Research Frameworks

Dresden Database Research Group

PilotScope

https://github.com/alibaba/pilotscope

- Research framework focused on ML4DB
- Not limited to query optimization
  - E.g., also support for index recommendation
- Developed at Alibaba
- Uses Push/Pull connectors on an actual DBMS

✗	Broad support for different optimization strategies
✓	Transparent implementation of prototypes
✓	Execution on Postgres
✓	Easy comparison of different prototypes

PostBOUND

PostBOUND

Dresden Database Research Group

Overview

- Python framework for research in query optimization
- Two primary use cases:
  - Prototyping of new optimization strategies
  - Transparent evaluation of different optimizers

○	Broad support for different optimization strategies
○	Transparent implementation of prototypes
○	Query execution on Postgres
○	Easy comparison of different prototypes

PostBOUND

Dresden Database Research Group

Overview

- Python framework for research in query optimization
- Two primary use cases
  - Prototyping of new optimization strategies
  - Transparent evaluation of different optimization strategies

✓	Broad support for different optimization strategies
✓	Transparent implementation of prototypes
○	Query execution on Postgres
○	Easy comparison of different prototypes

- Provides different optimization pipelines
- Pipelines correspond to different optimizer architectures
- Researchers select suitable pipeline
- Researchers implement interfaces of the pipeline

PostBOUND

Dresden Database Research Group

Overview

- Python framework for research in query optimization
- Two primary use cases
  - Prototyping of new optimization strategies
  - Transparent evaluation of different optimizers

✓	Broad support for different optimization approaches
✓	Transparent implementation of prototypes
✓	Query execution on Postgres
○	Easy comparison of different prototypes

- Use hinting to enforce PostBOUND's optimization decisions
- Hints overwrite optimizer of the target database

PostBOUND

Dresden Database Research Group

Overview

- Python framework for research in query optimization
- Two primary use cases
  - Prototyping of new optimization strategies
  - Transparent evaluation of different optimizers

✓	Broad support for different optimization strategies
✓	Transparent implementation of prototypes
✓	Query execution on Postgres
✓	Easy comparison of different prototypes

- Benchmark “support suite“, e.g.,
- Query plan analysis
- Automatic cache prewarming
- Pre-defined workloads + data sets (IMDB, Stats, Stack, SSB)

PostBOUND

Dresden Database Research Group

Non-functional goals

- Easy to use
- Nice programming experience
  - Powerful abstractions
  - Low boilerplate
  - High customizability
- Fast onboarding
  - Provide many optimization algorithms out of the box
- Low configuration overhead
  - Use sensible defaults whenever possible
- Good documentation

Postl

Non-fu

- Easy
- Nice
  - P
  - L
  - H
- Fast
  - P
    o
- Low
  - U
- Goc

https://postbound.readthedocs.io/en/latest/

PostBOUND 0.16.0

Search Ctrl + K

10 Minutes to PostBOUND
Setup
Core concepts
Advanced usage
Cookbook
API

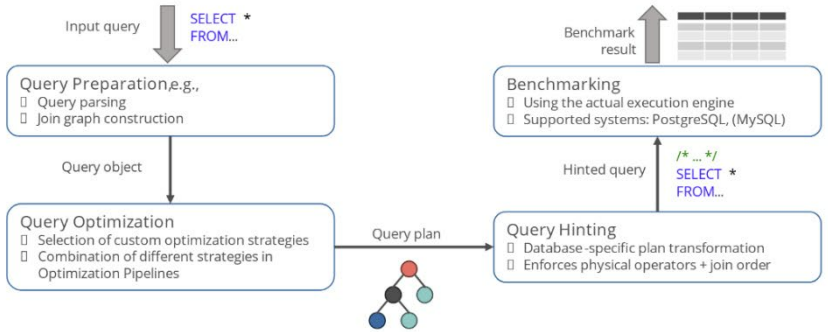
PostBOUND documentation

PostBOUND is a Python framework for research in query optimization. It provides a high-level interface to implement novel optimization algorithms and to analyze their performance.

Tip

New users should start by reading the 10 Minutes to PostBOUND tutorial and the Setup guide. The remainder of the documentation describes the different parts of the framework in more detail.

In general, using the PostBOUND framework for optimizer research uses a workflow similar to the following:



Typical workflow for using PostBOUND for optimizer research.

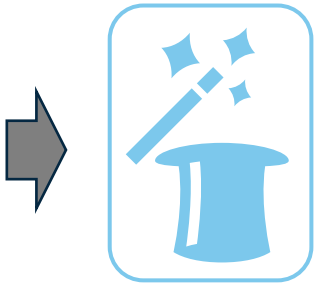
Contents

PostBOUND documentation
Contents
Example
History
Citation
Indices and tables
References

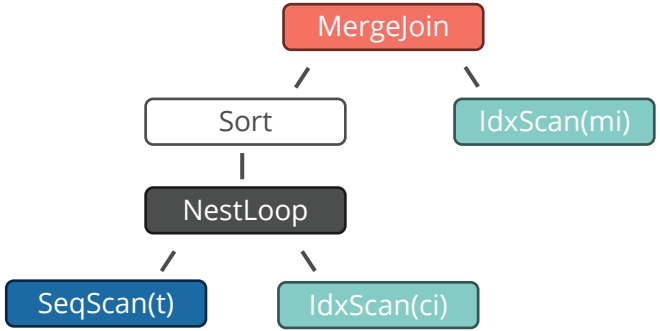
Plan Enforcement via Query Hinting

Dresden Database Research Group

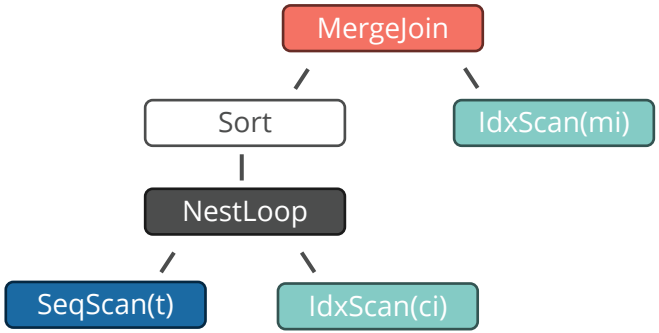
```
SELECT *
FROM title t
JOIN movie_info mi ON t.id = mi.movie_id
JOIN cast_info  ci ON t.id = ci.movie_id
WHERE t.production_year > 2010
```

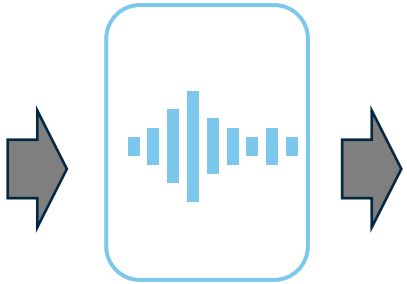


PostBOUND “magic”
(for now)



Plan Enforcement via Query Hinting





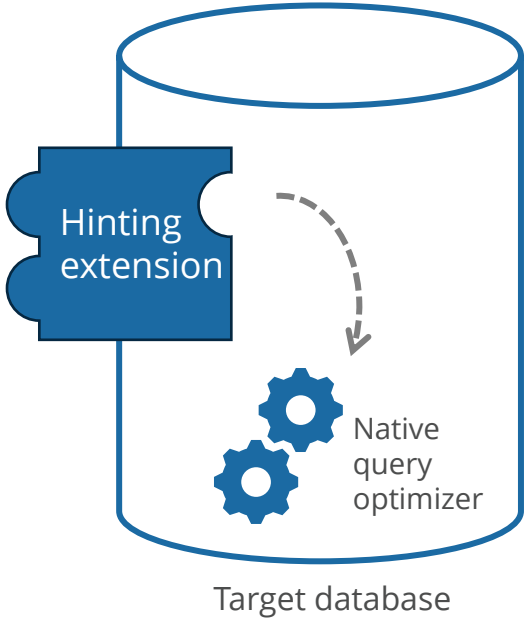
PostBOUND
hinting module

```
/*+
    Leading(((t ci) mi))
    MergeJoin(t ci mi) NestLoop(t ci)
    SeqScan(t) IndexScan(mi) IndexScan(ci)
 */
SELECT *
FROM title t
JOIN movie_info mi ON t.id = mi.movie_id
JOIN cast_info ci ON t.id = ci.movie_id
WHERE t.production_year > 2010
```

Plan Enforcement via Query Hinting

Dresden Database Research Group

```
/*+
    Leading(((t ci) mi))
    MergeJoin(t ci mi) NestLoop(t ci)
    SeqScan(t) IndexScan(mi) IndexScan(ci)
 */
SELECT *
FROM title t
JOIN movie_info mi ON t.id = mi.movie_id
JOIN cast_info ci ON t.id = ci.movie_id
WHERE t.production_year > 2010
```

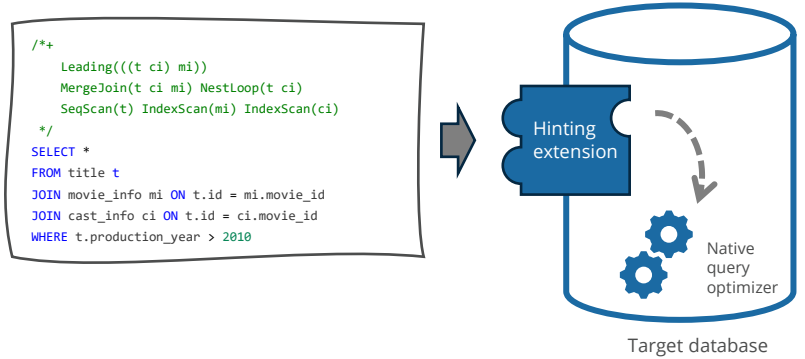


Plan Enforcement via Query Hinting

Dresden Database Research Group

Supported Hinting Backends

- pg_hint_plan
  - Hints for join order
  - Hints for most physical operators
  - Hints for join cardinalities
- pg_lab
  - Same as pg_hint_plan, but additionally:
  - Hints for parallel plans
  - Hints for base table cardinalities



Consequenses

- Can only enforce plan features that are supported by the hinting backend
- Currently no support for re-optimization approaches

pg_lab

Dresden Database Research Group

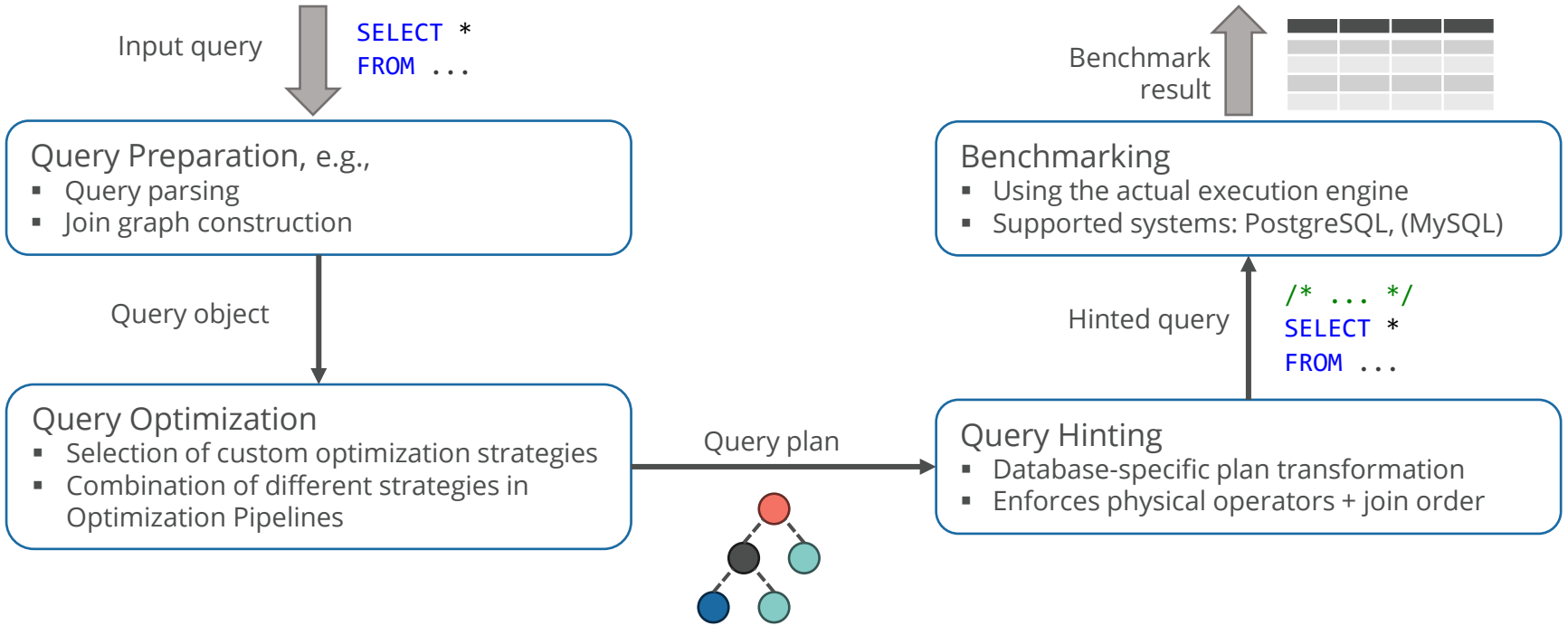
“Postgres for Optimizer Research”

- Lightweight fork of PostgreSQL
- Adds additional extension points to the optimization workflow
- Provides fine-grained hinting capabilities

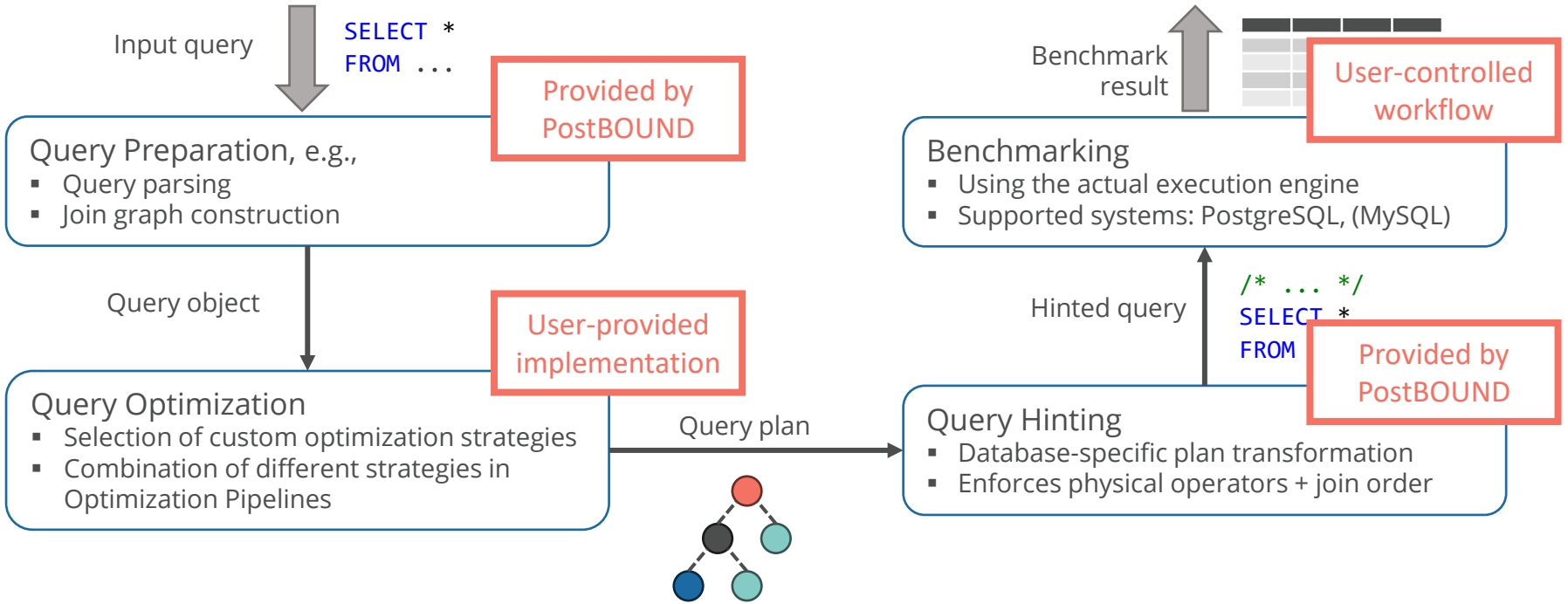
Extension points

- Cardinality estimation
- Cost estimation
- Path pruning
- Parallel worker estimation

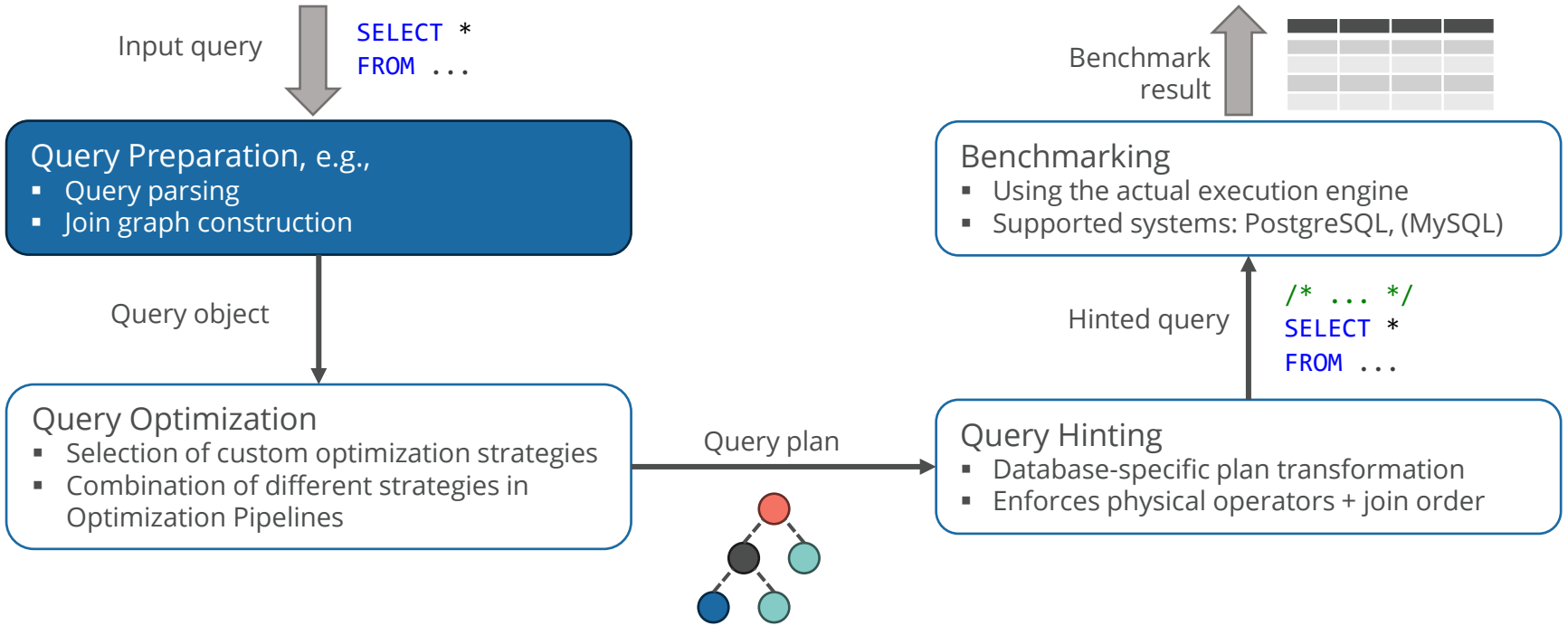
PostBOUND Workflow



PostBOUND Workflow



PostBOUND Workflow



Stats Benchmark

Dresden Database Research Group

CrossValidated

Dataset

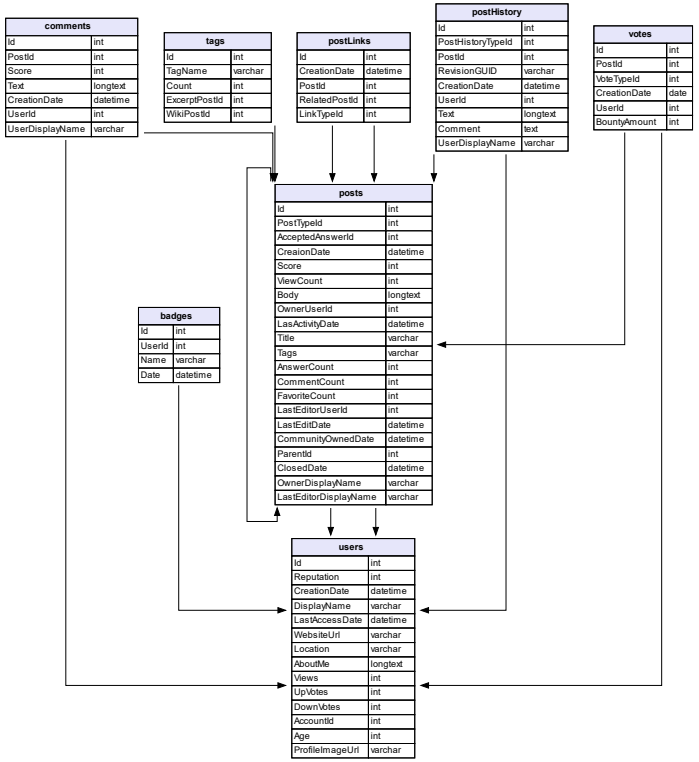
- Based on the Statistics Stackoverflow (CrossValidated)
- Models relation between users, their questions, and answers
- 8 tables with complex (foreign-key) correlations

Workload

- 146 queries in total, between 1 and 6 joins
- Auto-generated queries based on complex templates

Purpose

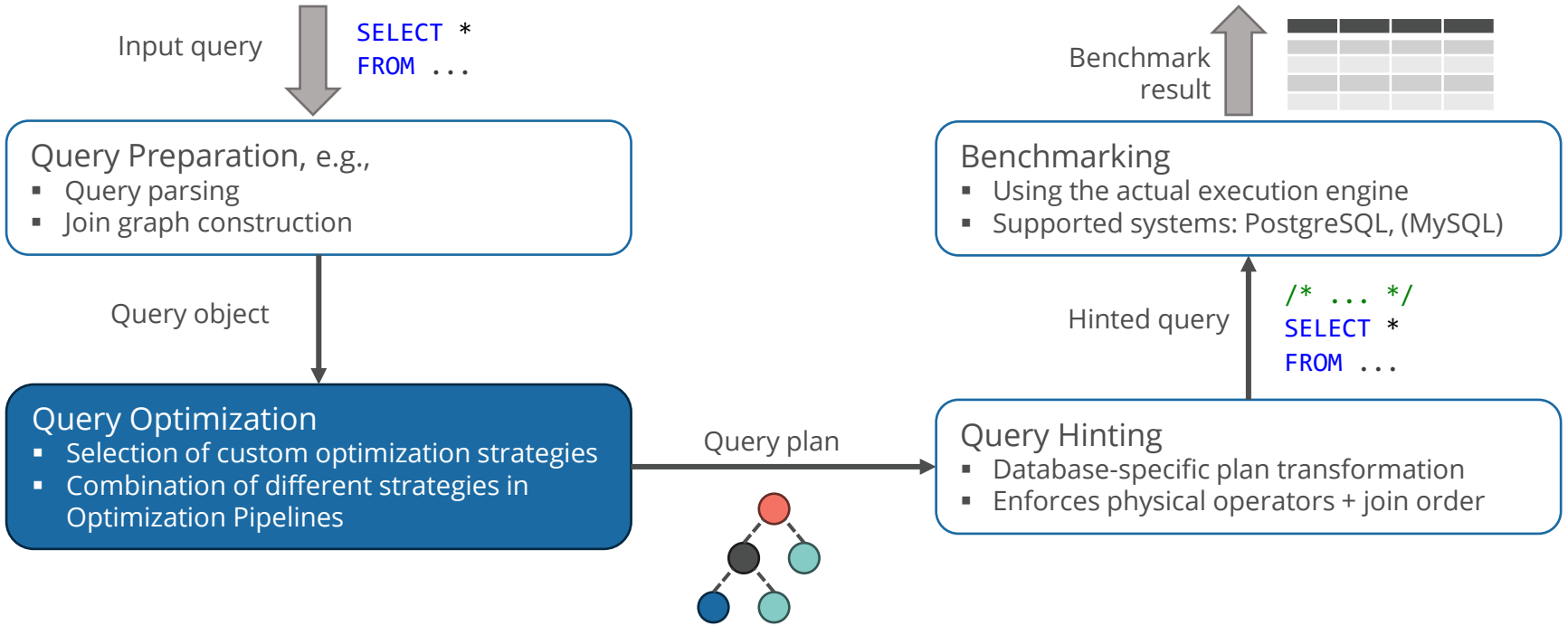
- Lightweight benchmark (<125MB compressed) for cardinality estimation
- Challenging correlations, skewed data distributions
- Presented by Han et al.: "*Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation*" (VLDB 2022, Link)



Dresden Database Research Group

Demo

PostBOUND Workflow

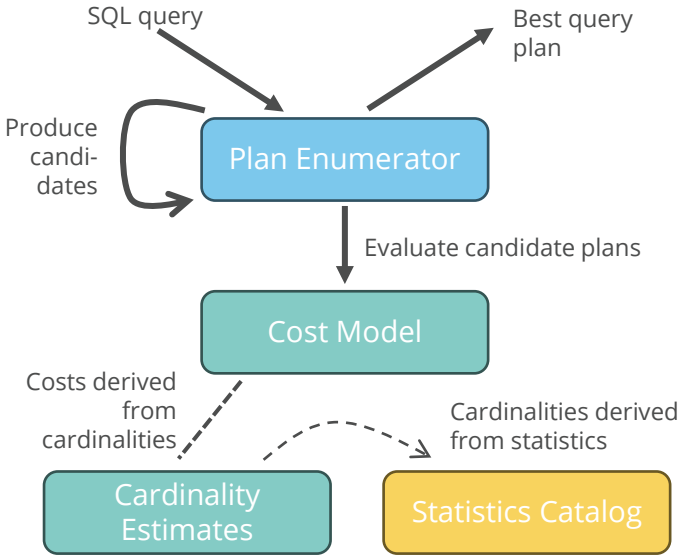


Optimizer Implementation

Dresden Database Research Group

Optimization pipelines

- Mental models for different optimizer architectures
- Pipelines define specific optimization steps (e.g., cardinality estimation)
- Researchers need to implement (subsets of) the stages
- “Fill the gaps“ principle
  - Researchers only need to worry about their specific stages
  - PostBOUND uses reasonable defaults for the rest



Textbook pipeline

- Available stages:
  - Plan enumerator
  - Cost model
  - Cardinality estimator
- Defaults: native estimators and simulated enumerator

Optimizer Implementation

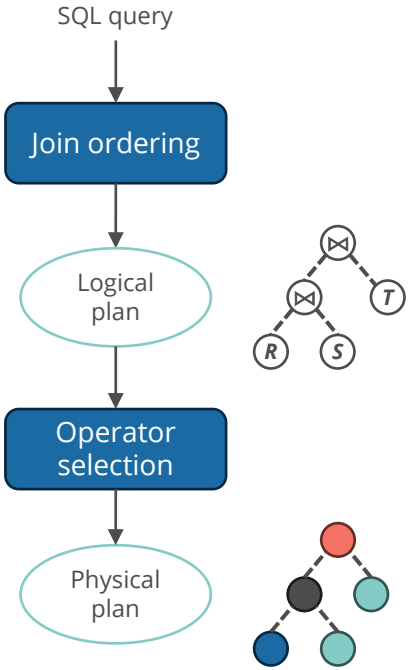
Dresden Database Research Group

Optimization pipelines

- Mental models for different optimizer architectures
- Pipelines define specific optimization steps (e.g., cardinality estimation)
- Researchers need to implement (subsets of) the stages
- “Fill the gaps“ principle
  - Researchers only need to worry about their specific stages
  - PostBOUND uses reasonable defaults for the rest

Multi-stage pipeline

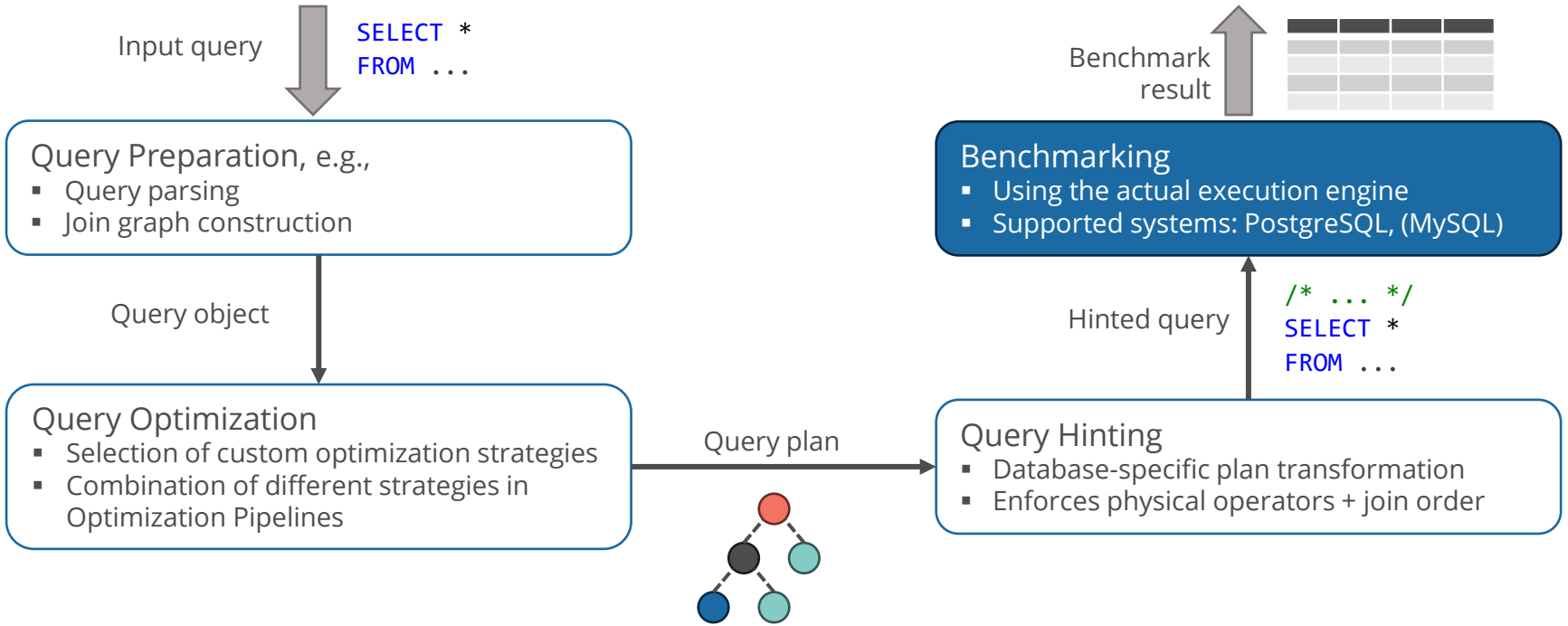
- Available stages:
  - Join ordering
  - Operator selection
  - (Plan parameterization)
- Defaults: native strategies



Dresden Database Research Group

Demo

PostBOUND Workflow



Benchmarking

Dresden Database Research Group

Benchmark execution

- Goal: reproducible + transparent evaluation
- Query preparation for common tasks (e.g., warm start)
- Tools generate many additional data points
  - Pipeline configuration
  - Database schema
  - ...
- Integrated into Python data analysis ecosystem (e.g., pandas)

Support Tooling

- Setup scripts for common databases + workloads (IMDB, SSB, Stats, StackOverflow)
- Setup scripts for different Postgres versions

Dresden Database Research Group

Demo

Dresden Database Research Group

Implementing Your Own Optimizer

Hands-On Material:
github.com/db-tu-dresden/SIGMOD25-OptimizerTutorial

More Examples

Dresden Database Research Group

SIGMOD Research Paper

- „An Elephant Under The Microscope“
- Broad analysis of the interaction of optimizer components
- Custom optimization “strategies“ for
  - Join ordering
  - Cardinality estimation
  - Pessimistic query optimization

An Elephant Under The Microscope: Analyzing The Interaction Of Optimizer Components In PostgreSQL

RICO BERGMANN, Technische Universität Dresden, Germany
CLAUDIO HARTMANN, Technische Universität Dresden, Germany
DIRK HABICH, Technische Universität Dresden, Germany
WOLFGANG LEHNER, Technische Universität Dresden, Germany

Despite an ever-growing corpus of novel query optimization strategies, the interaction of the core components of query optimizers is still not well understood. This situation can be problematic for two main reasons: On the one hand, this may cause surprising results when two components influence each other in an unexpected way. On the other hand, this can lead to wasted effort in regard to both engineering and research, e.g., when an improvement for one component is dwarfed or entirely canceled out by problems of another component. Therefore, we argue that making improvements to a single optimization component requires a thorough understanding of how these changes might affect the other components. To achieve this understanding, we present results of a comprehensive experimental analysis of the interplay in the traditional optimizer architecture using the widely-used PostgreSQL system as prime representative. Our evaluation and analysis revisit the core building blocks of such an optimizer, i.e. per-column statistics, cardinality estimation, cost model, and plan generation. In particular, we analyze how these building blocks influence each other and how they react when faced with faulty input, such as imprecise cardinality estimates. Based on our results, we draw novel conclusions and make recommendations on how these should be taken into account.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: Query Optimization; Data Analysis; Cardinality Estimation; Join Enumeration; Cost Model

ACM Reference Format:
Rico Bergmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2025. An Elephant Under The Microscope: Analyzing The Interaction Of Optimizer Components In PostgreSQL. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 9 (February 2025), 28 pages. https://doi.org/10.1145/3709659

1 Introduction

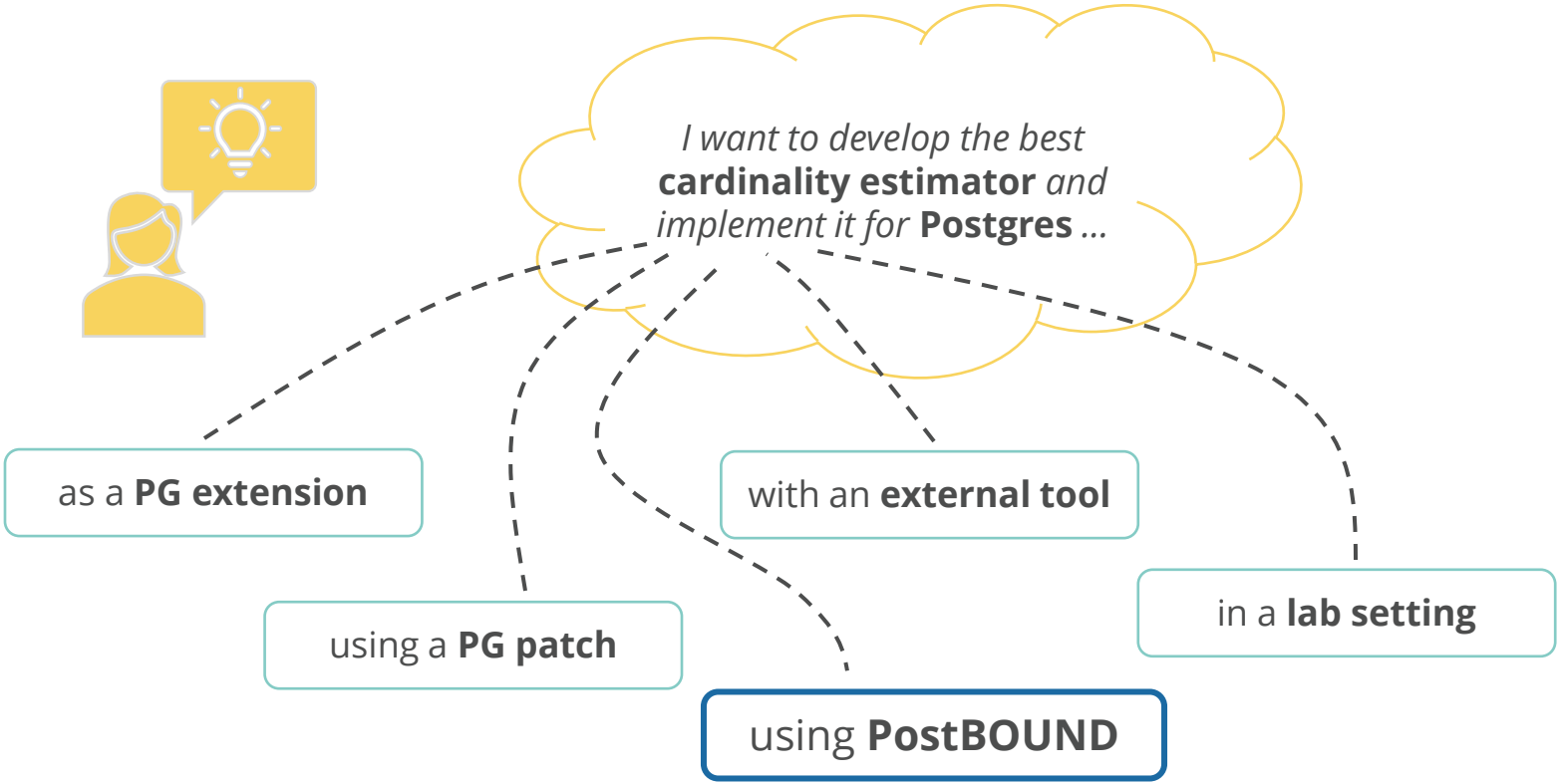
Although query optimization in relational database systems has been studied in decades of research [31, 35, 39, 53], there is still no consensus on how to do query optimization the "right" way and novel approaches are published each year. Those contributions usually focus on different aspects of the optimization process, for example by introducing new strategies for cardinality estimation [27, 63], by presenting improved cost models [23, 36], or by introducing refined strategies for plan enumeration [21, 34]. However, these improvements are typically made in isolation, without taking the interaction of the various query optimizer components into account and without comprehensively analyzing the effect on these other components. Since a query optimizer is a

Authors' Contact Information: Rico Bergmann, Technische Universität Dresden, Dresden, Germany, rico.bergmann1@tu-dresden.de; Claudio Hartmann, Technische Universität Dresden, Dresden, Germany, claudio.hartmann@tu-dresden.de; Dirk Habich, Technische Universität Dresden, Dresden, Germany, dirk.habich@tu-dresden.de; Wolfgang Lehner, Technische Universität Dresden, Dresden, Germany, wolfgang.lehner@tu-dresden.de.

ACM 2836-6573/2025/2-ART9
https://doi.org/10.1145/3709659

From Idea To Implementation

Dresden Database Research Group

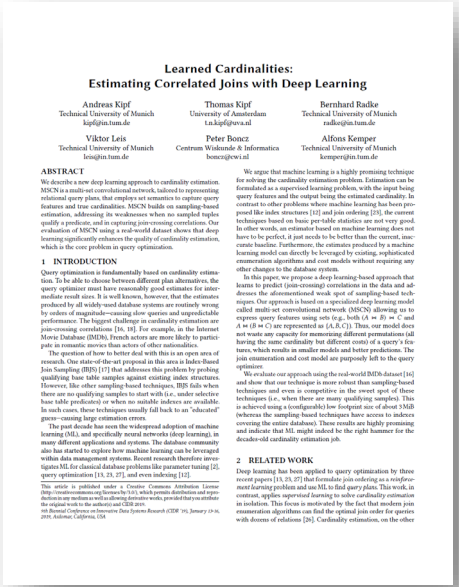


Supervised Cardinality Estimation *(MSCN-light)*

Dresden Database Research Group

MSCN in a nutshell

- Supervised cardinality estimation model
- Queries are featurized according to
  - Their tables
  - The join graph
  - Filter predicates
- Different sub-modules for each component
- Modules are merged in final model



Learned Cardinalities:
Estimating Correlated Joins with Deep Learning

Andreas Kipf, Technical University of Munich, kipf@in.tum.de
Thomas Kipf, University of Amsterdam, t.n.kipf@uva.nl
Bernhard Radke, Technical University of Munich, radke@in.tum.de
Viktor Leis, Technical University of Munich, leis@in.tum.de
Peter Boncz, Centrum Wiskunde & Informatica, boncz@cwi.nl
Alfons Kemper, Technical University of Munich, kemper@in.tum.de

ABSTRACT

We describe a new deep learning approach to cardinality estimation. MSCN is a multi-set convolutional network, tailored to representing relational query plans, that employs set semantics to capture query features and true cardinalities. MSCN builds on sampling-based estimation, addressing its weaknesses when no sampled tuples qualify a predicate, and in capturing join-crossing correlations. Our evaluation of MSCN using a real-world dataset shows that deep learning significantly enhances the quality of cardinality estimation, which is the core problem in query optimization.

1 INTRODUCTION

Query optimization is fundamentally based on cardinality estimation. To be able to choose between different plan alternatives, the query optimizer must have reasonably good estimates for intermediate result sizes. It is well known, however, that the estimates produced by all widely-used database systems are routinely wrong by orders of magnitude—causing slow queries and unpredictable performance. The biggest challenge in cardinality estimation are join-crossing correlations [16, 18]. For example, in the Internet Movie Database (IMDb), French actors are more likely to participate in romantic movies than actors of other nationalities.

The question of how to better deal with this is an open area of research. One state-of-the-art proposal in this area is Index-Based Join Sampling (IBJS) [17] that addresses this problem by probing qualifying base table samples against existing index structures. However, like other sampling-based techniques, IBJS fails when there are no qualifying samples to start with (i.e., under selective base table predicates) or when no suitable indexes are available. In such cases, these techniques usually fall back to an "educated" guess—causing large estimation errors.

The past decade has seen the widespread adoption of machine learning (ML), and specifically neural networks (deep learning), in many different applications and systems. The database community also has started to explore how machine learning can be leveraged within data management systems. Recent research therefore investigates ML for classical database problems like parameter tuning [2], query optimization [13, 23, 27], and even indexing [12].

9th Biennial Conference on Innovative Data Systems Research (CIDR '19), January 13-16, 2019, Asilomar, California, USA

We argue that machine learning is a highly promising technique for solving the cardinality estimation problem. Estimation can be formulated as a supervised learning problem, with the input being query features and the output being the estimated cardinality. In contrast to other problems where machine learning has been proposed like index structures [12] and join ordering [23], the current techniques based on basic per-table statistics are not very good. In other words, an estimator based on machine learning does not have to be perfect, it just needs to be better than the current, inaccurate baseline. Furthermore, the estimates produced by a machine learning model can directly be leveraged by existing, sophisticated enumeration algorithms and cost models without requiring any other changes to the database system.

In this paper, we propose a deep learning-based approach that learns to predict (join-crossing) correlations in the data and addresses the aforementioned weak spot of sampling-based techniques. Our approach is based on a specialized deep learning model called multi-set convolutional network (MSCN) allowing us to express query features using sets (e.g., both $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ are represented as $\{A, B, C\}$). Thus, our model does not waste any capacity for memorizing different permutations (all having the same cardinality but different costs) of a query's features, which results in smaller models and better predictions. The join enumeration and cost model are purposely left to the query optimizer.

We evaluate our approach using the real-world IMDb dataset [16] and show that our technique is more robust than sampling-based techniques and even is competitive in the sweet spot of these techniques (i.e., when there are many qualifying samples). This is achieved using a (configurable) low footprint size of about 5 MiB (whereas the sampling-based techniques have access to indexes covering the entire database). These results are highly promising and indicate that ML might indeed be the right hammer for the decades-old cardinality estimation job.

2 RELATED WORK

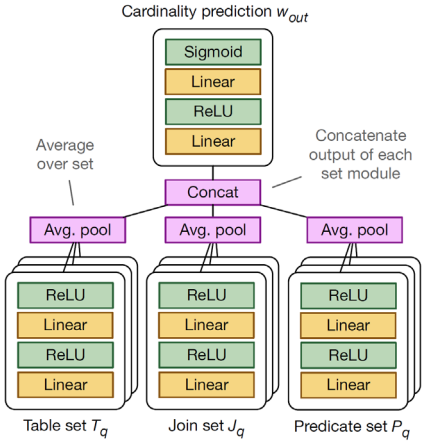
Deep learning has been applied to query optimization by three recent papers [13, 23, 27] that formulate join ordering as a *reinforcement learning* problem and use ML to find *query plans*. This work, in contrast, applies *supervised learning* to solve *cardinality estimation* in isolation. This focus is motivated by the fact that modern join enumeration algorithms can find the optimal join order for queries with dozens of relations [26]. Cardinality estimation, on the other

Kipf et al.: "Learned Cardinalities: Estimating Correlated Joins with Deep Learning" (CIDR'2019)

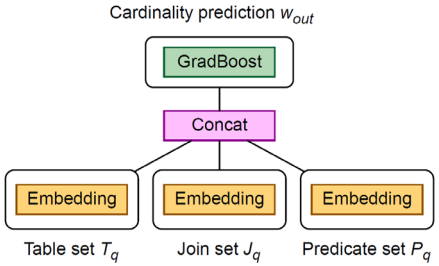
Supervised Cardinality Estimation *(MSCN-light)*

Dresden Database Research Group

Original MSCN model



Light model



Kipf et al.: “Learned Cardinalities: Estimating Correlated Joins with Deep Learning” (CIDR’2019)

Dresden Database Research Group

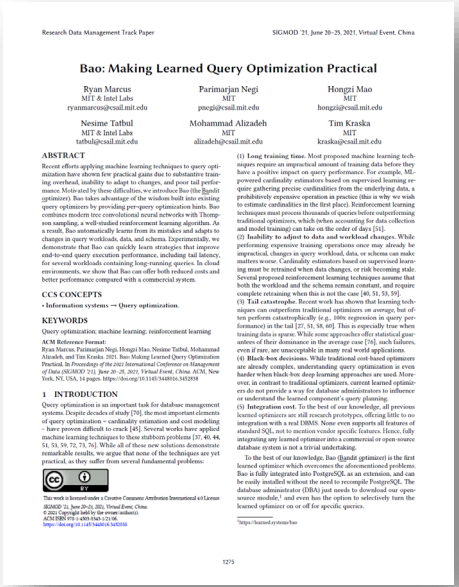
Live!

Plan Selection (*BAO-light*)

Dresden Database Research Group

BAO in a nutshell

- Use (global) planner hints to obtain different query plans
- Vectorize query plans using binarization + schema-independent features
- Estimate plan quality using a TCNN
- Selects the (estimated) best plan for execution



Research Data Management Track Paper SIGMOD '21, June 20–25, 2021, Virtual Event, China

Bao: Making Learned Query Optimization Practical

Ryan Marcus, MIT & Intel Labs, ryanmarcus@csail.mit.edu
Parimarjan Negi, MIT, pnegi@csail.mit.edu
Hongzi Mao, MIT, hongzi@csail.mit.edu
Nesime Tatbul, MIT & Intel Labs, tatbul@csail.mit.edu
Mohammad Alizadeh, MIT, alizadeh@csail.mit.edu
Tim Kraska, MIT, kraska@csail.mit.edu

ABSTRACT

Recent efforts applying machine learning techniques to query optimization have shown few practical gains due to substantive training overhead, inability to adapt to changes, and poor tail performance. Motivated by these difficulties, we introduce Bao (the bandit optimizer). Bao takes advantage of the wisdom built into existing query optimizers by providing per-query optimization hints. Bao combines modern tree convolutional neural networks with Thompson sampling, a well-studied reinforcement learning algorithm. As a result, Bao automatically learns from its mistakes and adapts to changes in query workloads, data, and schema. Experimentally, we demonstrate that Bao can quickly learn strategies that improve end-to-end query execution performance, including tail latency, for several workloads containing long-running queries. In cloud environments, we show that Bao can offer both reduced costs and better performance compared with a commercial system.

CCS CONCEPTS

• Information systems → Query optimization.

KEYWORDS

Query optimization; machine learning; reinforcement learning

ACM Reference Format:
Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452838

1 INTRODUCTION

Query optimization is an important task for database management systems. Despite decades of study [70], the most important elements of query optimization – cardinality estimation and cost modeling – have proven difficult to crack [45]. Several works have applied machine learning techniques to these stubborn problems [37, 40, 44, 51, 53, 59, 72, 73, 76]. While all of these new solutions demonstrate remarkable results, we argue that none of the techniques are yet practical, as they suffer from several fundamental problems:

(1) **Long training time.** Most proposed machine learning techniques require an impractical amount of training data before they have a positive impact on query performance. For example, ML-powered cardinality estimators based on supervised learning require gathering precise cardinalities from the underlying data, a prohibitively expensive operation in practice (this is why we wish to estimate cardinalities in the first place). Reinforcement 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 [51].

(2) **Inability to adjust to data and workload changes.** While performing expensive training operations once may already be impractical, changes in query workload, data, or schema can make matters worse. Cardinality estimators based on supervised learning must be retrained when data changes, or risk becoming stale. Several proposed reinforcement learning techniques assume that both the workload and the schema remain constant, and require complete retraining when this is not the case [40, 51, 53, 59].

(3) **Tail catastrophe.** Recent work has shown that learning techniques can outperform traditional optimizers on average, but often perform catastrophically (e.g., 100x regression in query performance) in the tail [37, 51, 58, 60]. This is especially true when training data is sparse. While some approaches offer statistical guarantees of their dominance in the average case [76], such failures, even if rare, are unacceptable in many real world applications.

(4) **Black-box decisions.** While traditional cost-based optimizers are already complex, understanding query optimization is even harder when black-box deep learning approaches are used. Moreover, in contrast to traditional optimizers, current learned optimizers do not provide a way for database administrators to influence or understand the learned component's query planning.

(5) **Integration cost.** To the best of our knowledge, all previous learned optimizers are still research prototypes, offering little to no integration with a real DBMS. None even supports all features of standard SQL, not to mention vendor specific features. Hence, fully integrating any learned optimizer into a commercial or open-source database system is not a trivial undertaking.

To the best of our knowledge, Bao (bandit optimizer) is the first learned optimizer which overcomes the aforementioned problems. Bao is fully integrated into PostgreSQL as an extension, and can be easily installed without the need to recompile PostgreSQL. The database administrator (DBA) just needs to download our open-source module,[1] and even has the option to selectively turn the learned optimizer on or off for specific queries.

[1] https://learned.systems/bao

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3452838

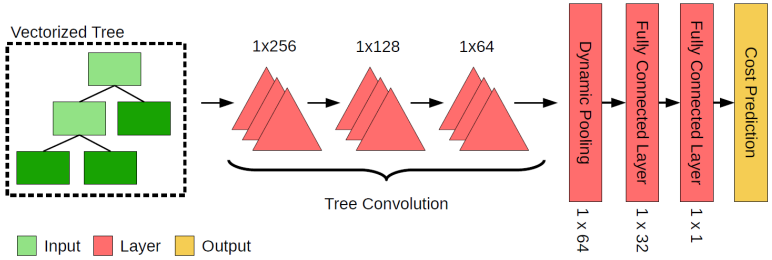
1275

Marcus et al.: "BAO: Making Learned Query Optimization Practical" (SIGMOD'2021)

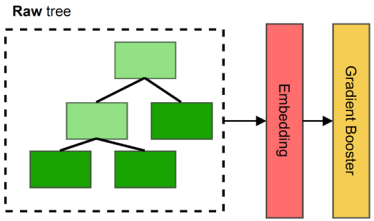
Plan Selection (*BAO-light*)

Dresden Database Research Group

Original BAO model



Light model



Marcus et al.: "BAO: Making Learned Query Optimization Practical" (SIGMOD'2021)

Dresden Database Research Group

Live!

Beyond Learned Optimizers: Pessimistic

Dresden Database Research Group

Core idea

- Use upper bounds instead of cardinality point-estimates to derive a robust execution plan
- Disable dangerous join operators, e.g., nested-loop join

Our adaptation

- Use the UES bound formula
- Only enable hash joins
- DBMS is free to select a join order
- DBMS is free to select scan operators

Simplicity Done Right for Join Ordering

Axel Hertzschuch, Claudio Hartmann, Dirk Habich, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
Dresden, Germany
firstname.lastname@tu-dresden.de

ABSTRACT

In this paper, we propose a simple, yet fast and effective approach to determine good join orders for arbitrary select-project-join queries. Our scheme comprises three building blocks: (i) a simple upper bound for arbitrary multi-joins, (ii) appropriate join enumeration according to the upper bound, and (iii) sampling as query execution to provide fast and near-exact estimates for complex conjunctive filters. As we are going to show, using the Join-Order-Benchmark (JOB), our simple approach provides better join orderings with significantly less optimization overhead, resulting in a substantially faster response time for all 113 JOB queries compared to state-of-the-art and recent approaches.

1. INTRODUCTION

Although query optimization has been a core research topic for decades, it is still far from being solved [7]. This holds especially true for one of the most studied challenges in query optimization: *finding a good join order* [2, 7]. To tackle this issue, reliable cardinality estimates for arbitrary joins are required [2, 7]. This includes joins over intermediate join results and pre-filtered base tables. However, the question whether it is even possible to achieve such estimates without join execution is yet to be answered [15].

Traditional estimation techniques frequently rely on basic heuristics that may assume predicate independence and a uniform distribution of attribute values [7]. However, relying on these assumptions can lead to disastrous join orderings [7]. To overcome this issue, various sophisticated techniques for the join cardinality estimation have been proposed in recent years. On the one hand, sampling approaches seem appealing at first glance, for example [8, 11, 21], but they do not scale well to many joins [3, 21]. On the other hand, we see an increasing popularity of modern machine learning techniques [6, 20] as they model complex data characteristics. However, these models do not yet cover all relevant filter predicate types and their training depends on executing a plethora of joins, which may take days or weeks [19].

As recently shown by Cai et al. [2], guaranteed bounds for join cardinalities lead to better and more robust join orderings for arbitrary select-project-join (SPJ) queries. However, their sketch-based approach to generate upper bounds does not scale well for many joins in terms of optimization time, as these sketches have to be populated at run time if filters are present [2]. To comprehensively tackle that shortcoming, we introduce our novel *UES* concept in this paper. The most outstanding feature of our concept is its simplicity which is achieved by three building blocks:

***U*-Block:** Assuming basic attribute statistics and accurate selectivity estimates for filters over base tables, we demonstrate a simple, yet effective Upper bound for an arbitrary number of joins.

***E*-Block:** Appropriately Enumerating joins according to our upper bound effectively prevents overly aggressive (sometimes disastrous) join orderings.

***S*-Block:** To guarantee accurate selectivity estimates even for complex filters in SPJ-queries required by the *U*-block, we treat Sampling like query execution.

Using the *Join-Order-Benchmark (JOB)* [7], we demonstrate that our novel *UES* concept outperforms built-in state-of-the-art optimizers of Postgres as well as of MonetDB. Moreover, we compare our concept with the sketch-based upper bound approach of Cai et al. [2]. As we are going to show, our *UES* concept achieves highly competitive join orderings, resulting in similar (or slightly better) execution times for all 113 JOB queries while requiring significantly less planning time due to the simplicity of our concept compared to the sketch-based technique.[1]

2. UE-BLOCKS: JOIN ORDERING

This section describes our two fundamental building blocks *U* and *E* for a scalable join ordering. For that, we assume access to the common statistic of most frequent values (top-k statistics) and precise filter selectivity estimates. While histograms may suffice for basic filters, Section 3 details a more sophisticated approach for complex filters (*S*-Block).

2.1 U-Block: Simple Upper Bound for Joins

The first building block *U* includes a simple upper bound for an arbitrary number of joins. To describe our upper bound, we start with a single join and discuss arbitrary joins afterwards. Given the (*estimated*) cardinality of two (*pre-filtered*) tables, we calculate the smallest number of distinct

[1] All materials to reproduce and further analyze the results reported in this work are available at: https://github.com/aihertz/SimplicityDoneRight

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

Hertzschuch et al.: "Simplicity Done Right for Join Ordering" (CIDR'2021)

Dresden Database Research Group

Live!

TECHNISCHE UNIVERSITÄT DRESDEN

Dresden Database Research Group

Reproducible Prototyping of Query Optimizer Components

Rico Bergmann and Dirk Habich

Technische Universität Dresden, Germany

SIGMOD'25 Tutorial, June 27, 2025, Berlin, Germany