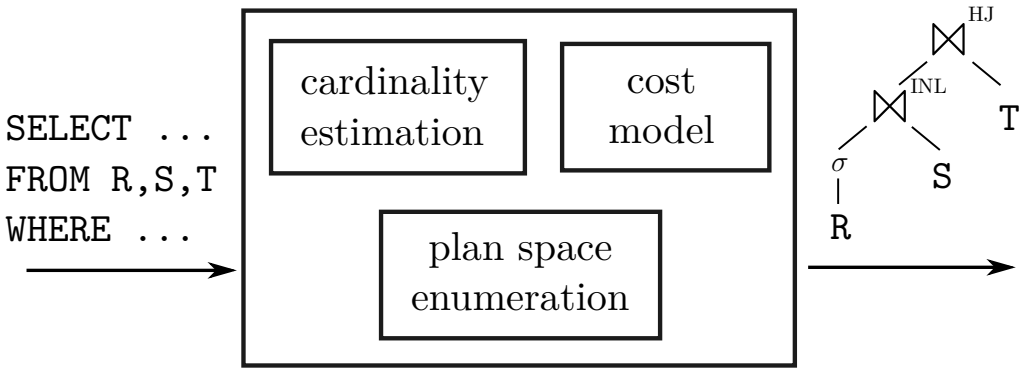
Query Optimization Through the Looking Glass

Some Lessons From Building an LLVM-Based Query Compiler

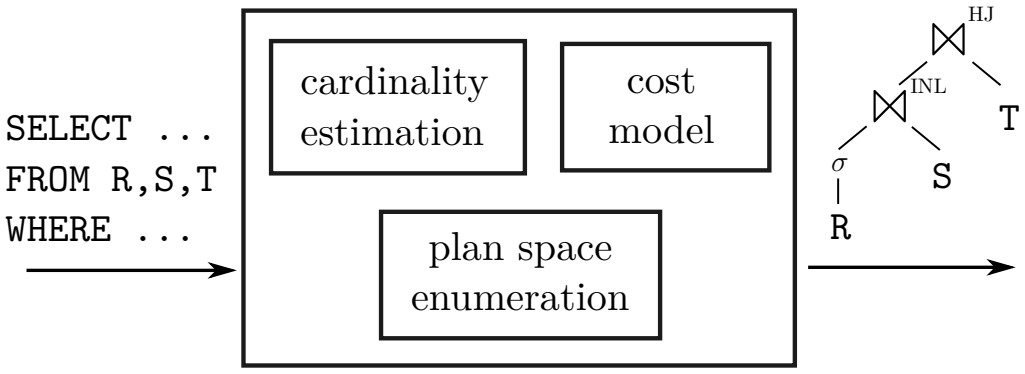
Viktor Leis

Technische Universität München

Introduction: Query Optimization



Introduction: Query Optimization



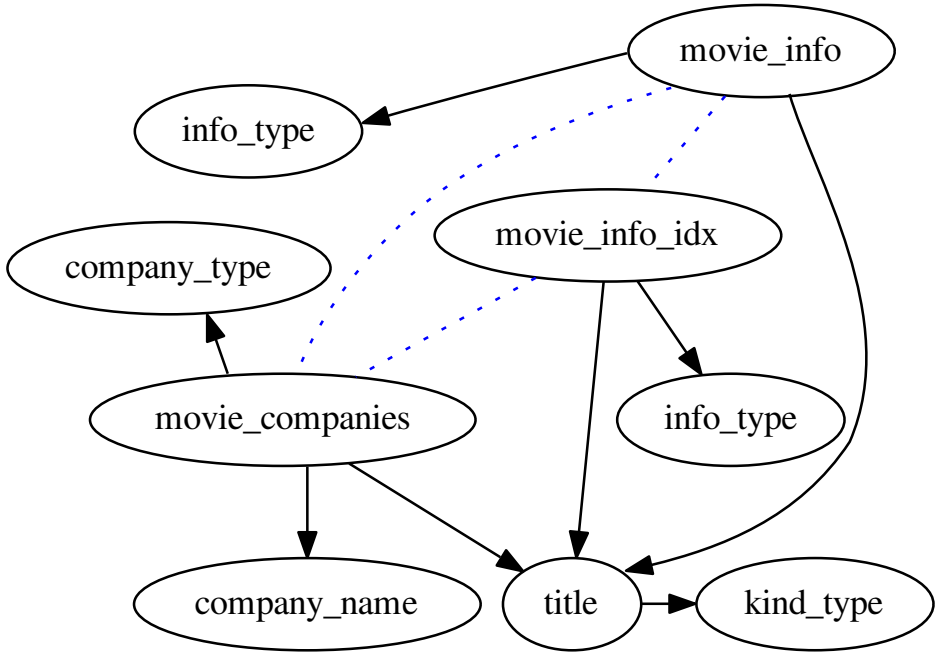
- How good are cardinality estimators?
- When do bad estimates lead to slow queries?
- How important is an accurate cost model for the overall query optimization process?
- (How large does the enumerated plan space need to be?)

Join Order Benchmark: Data Set

- *Internet Movie Data Base* data set (imdb.com)
- around 4GB, 21 relations
- information about movies and related facts about actors, directors, production companies, etc.
- publicly available for non-commercial use
- like all real-world data sets, full of join-crossing correlations

Join Order Benchmark: 113 Queries

```
SELECT cn.name, mi.info as rating, miidx.info as reldate
FROM company_name cn, company_type ct, info_type it,
   info_type it2, title t kind_type kt, movie_info mi,
   movie_companies mc, movie_info_idx miidx
WHERE cn.country_code = '[us]' AND it.info = 'rating'
AND ct.kind = 'production companies' AND kt.kind = 'movie'
AND it2.info = 'release dates' AND ...
```



Methodology

- cardinality extraction
  1. load data set into different systems
  2. run statistics tool (default settings)
  3. collect estimates for all subexpressions (e.g., using `EXPLAIN` in PostgreSQL) to obtain cardinality estimates
  4. also run `SELECT COUNT(*)` queries to obtain true cardinalities
- cardinality injection into PostgreSQL

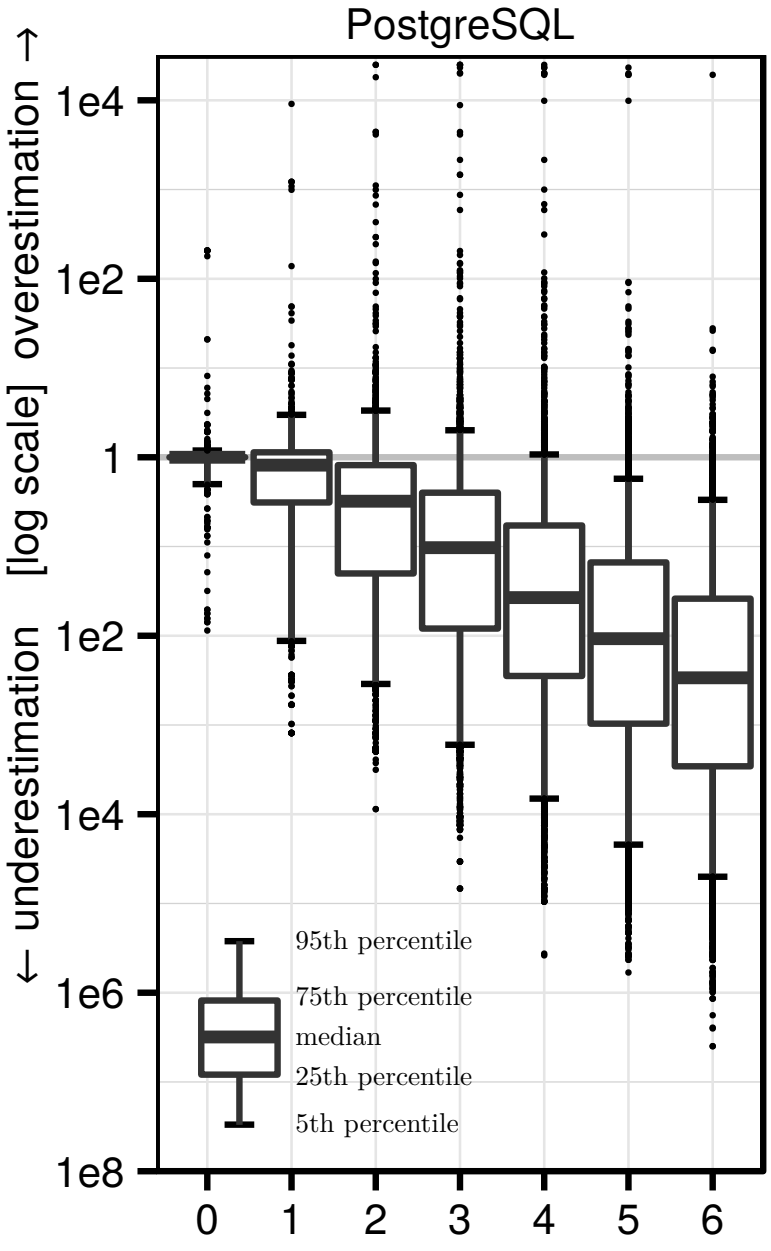
Cardinality Estimation

Cardinality Estimation for Base Table Selections

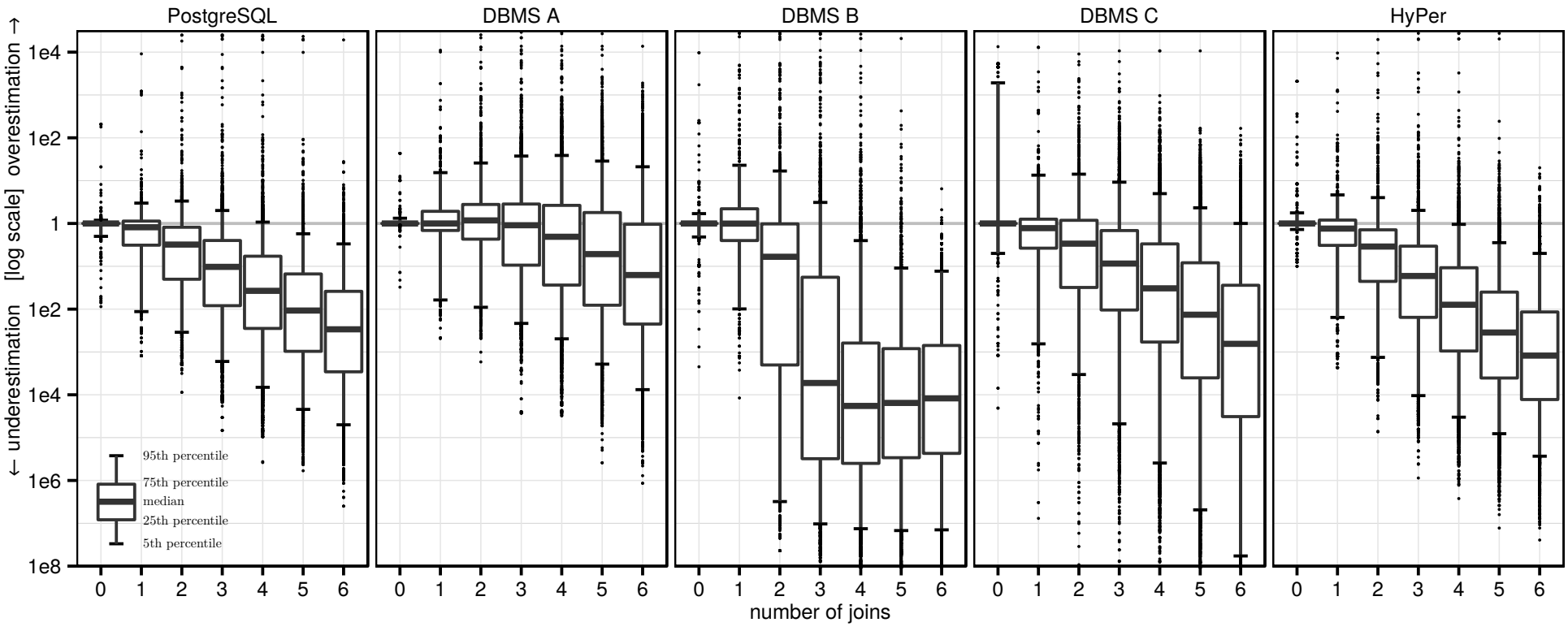
- *q-error*: $\max(e/r, r/e)$ (over- underestimation factor)

	median	90th	95th	max
PostgreSQL	1.00	2.08	6.10	207
DBMS A	1.01	1.33	1.98	43.4
DBMS B	1.00	6.03	30.2	104000
DBMS C	1.06	1677	5367	20471
HyPer	1.02	4.47	8.00	2084

Cardinality Estimation for Joins (1)



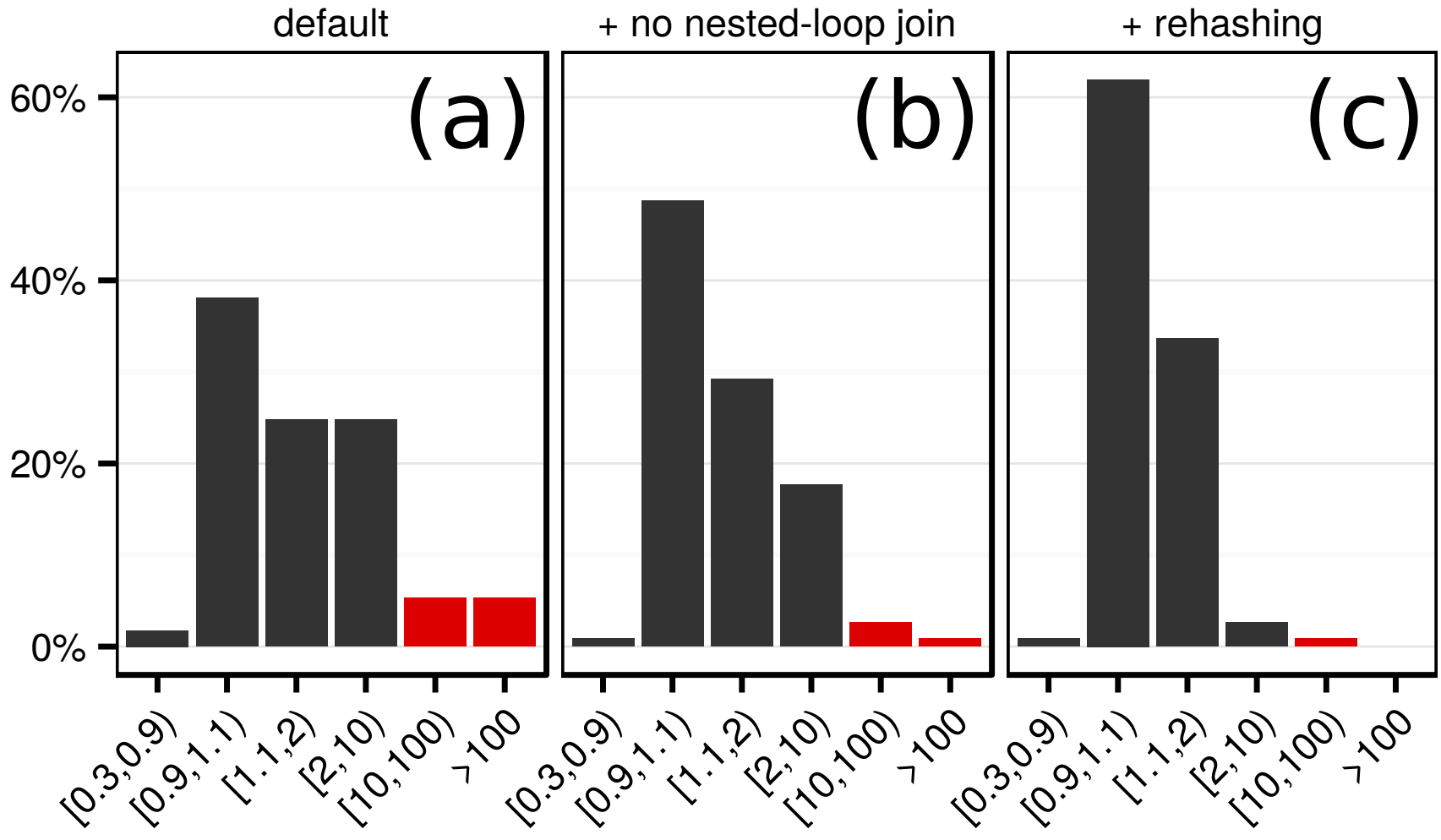
Cardinality Estimation for Joins (2)



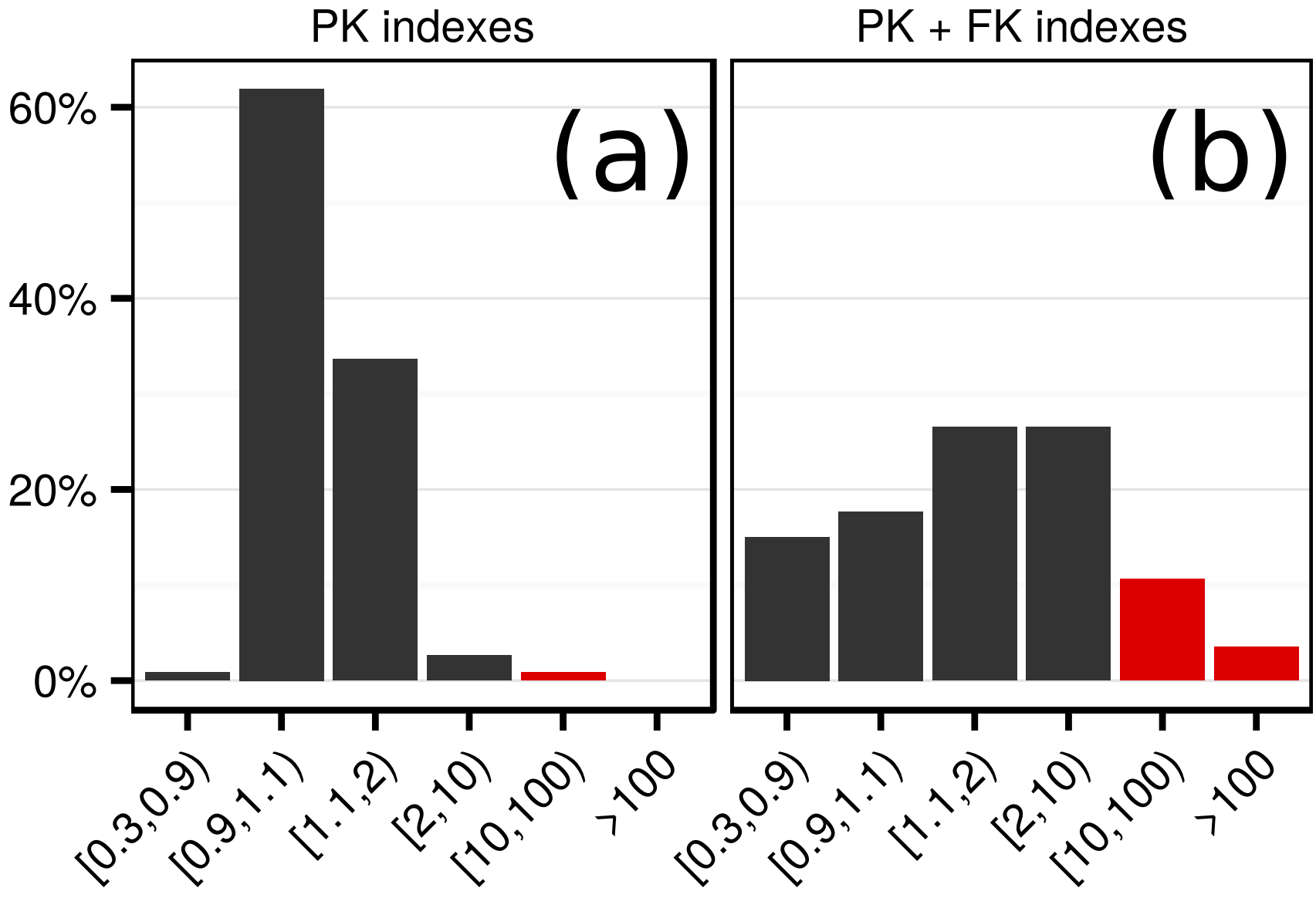
When do estimation errors result in bad query plans?

Effect of Estimates on Query Performance (1)

- performance with true cardinalities vs. PostgreSQL's estimates

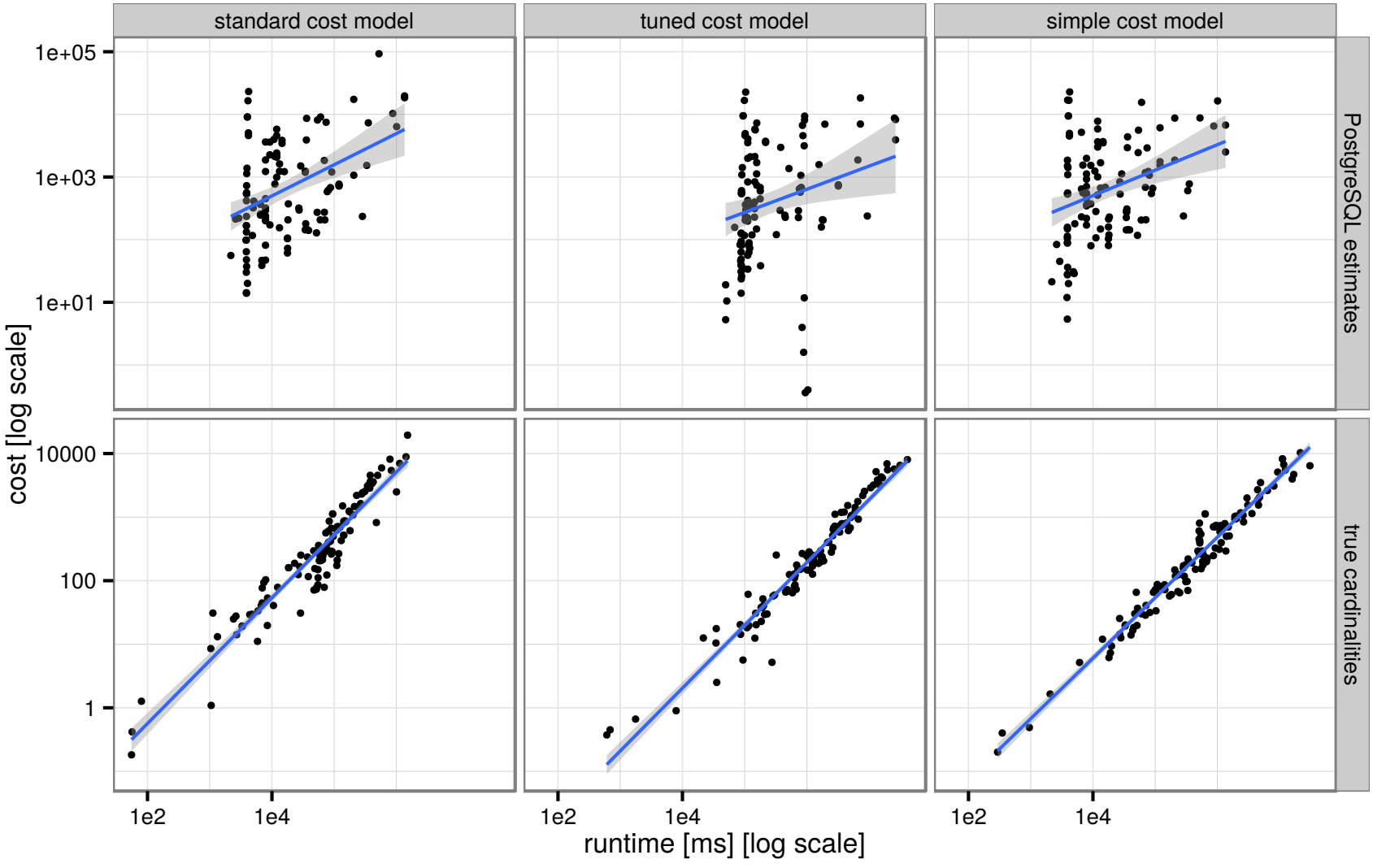


Effect of Estimates on Query Performance (2)



Cost Model

Cost vs. Runtime



Conclusions

- cardinality estimates are quite bad for all tested systems
- nested-loop joins are dangerous
- the more indexes are available the more difficult it becomes to find the optimal plan
- cost model is much less important than cardinality estimates

The paper ("How Good Are Query Optimizers, Really?") is available at: `www.vldb.org/pvldb/vol9/p204-leis.pdf`

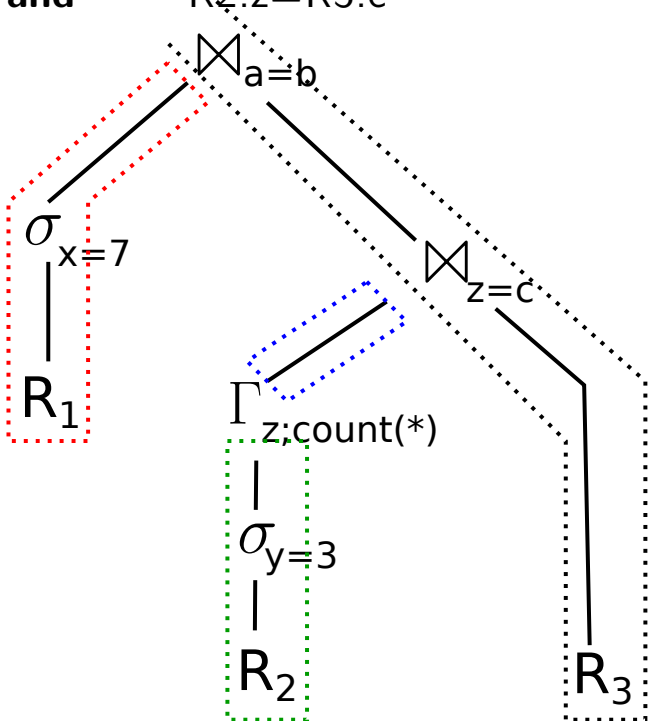
Part II: Building an LLVM-Based Query Compiler

HyPer

- research project started at TU Munich in 2010 by Thomas Neumann and Alfons Kemper
- relational main-memory DBMS
- SQL dialect is (mostly) PostgreSQL compatible, follows PostgreSQL server protocol
- goal: similar performance as hand-written C programs
- startup acquired by Tableau Software in 2016

Compiling Query Plans

```
select   *
from     R1,R3,
         (select   R2.z,
                   count(*)
          from     R2
          where    R2.y=3
          group by R2.z) R2
where    R1.x=7
and      R1.a=R3.b
and      R2.z=R3.c
```



initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z

for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$

for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z

for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$

for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$

Compiling Queries: Interface

- each operator implements the following interface:
  - produce(): generate code for that operator (and its child operators)
  - consume(attributes,source): generate code that receives a tuple from input

Compiling Queries: Example Operators

```
scan.produce():
    print "for each tuple in relation"
    scan.parent.consume(attributes,scan)
σ.produce:
    σ.input.produce()
σ.consume(a,s):
    print "if "+σ.condition
    σ.parent.consume(attr,σ)
⋈.produce():
    ⋈.left.produce()
    ⋈.right.produce()
⋈.consume(a,s):
    if (s==⋈.left)
        print "materialize tuple in hash table"
    else
        print "for each match in hashtable[" +a.joinattr+"]"
        ⋈.parent.consume(a+new attributes)
```

Compiling to LLVM IR

- compilation to LLVM Intermediate Representation (IR) ("machine-independent assembler") using C++ API
- attributes are kept in CPU registers as far as possible
- generating code can be tedious but many compile time abstractions help, e.g.:
  - high-level control flow constructs (`if`, `for`)
  - SQL value abstraction that handles null semantics, overflow checking, etc.
  - these abstractions do not cost any runtime
- for pragmatic and code size reasons some operators are partially/mostly implemented in C++

Performance (1 GB, 1 thread)

TPC-H		O0		O1	
#	PG 9.6	comp.	exec.	comp.	exec.
1	4908	6	161	42	77
2	254	23	13	149	8
3	1258	10	104	69	80
4	193	7	67	47	45
5	516	15	60	104	37

Lessons

- LLVM IR as a target language is great
  - stable
  - portable
  - generates efficient machine code
- for large queries, compilation times can become a problem

More Information

"Efficiently Compiling Efficient Query Plans":
`www.vldb.org/pvldb/vol9/p204-leis.pdf`

"Compiling Database Queries into Machine Code":
`sites.computer.org/debull/A14mar/p3.pdf`

all HyPer papers: `www.hyper-db.com`

interested in doing a PhD at TU Munich?
contact us (leis@in.tum.de)