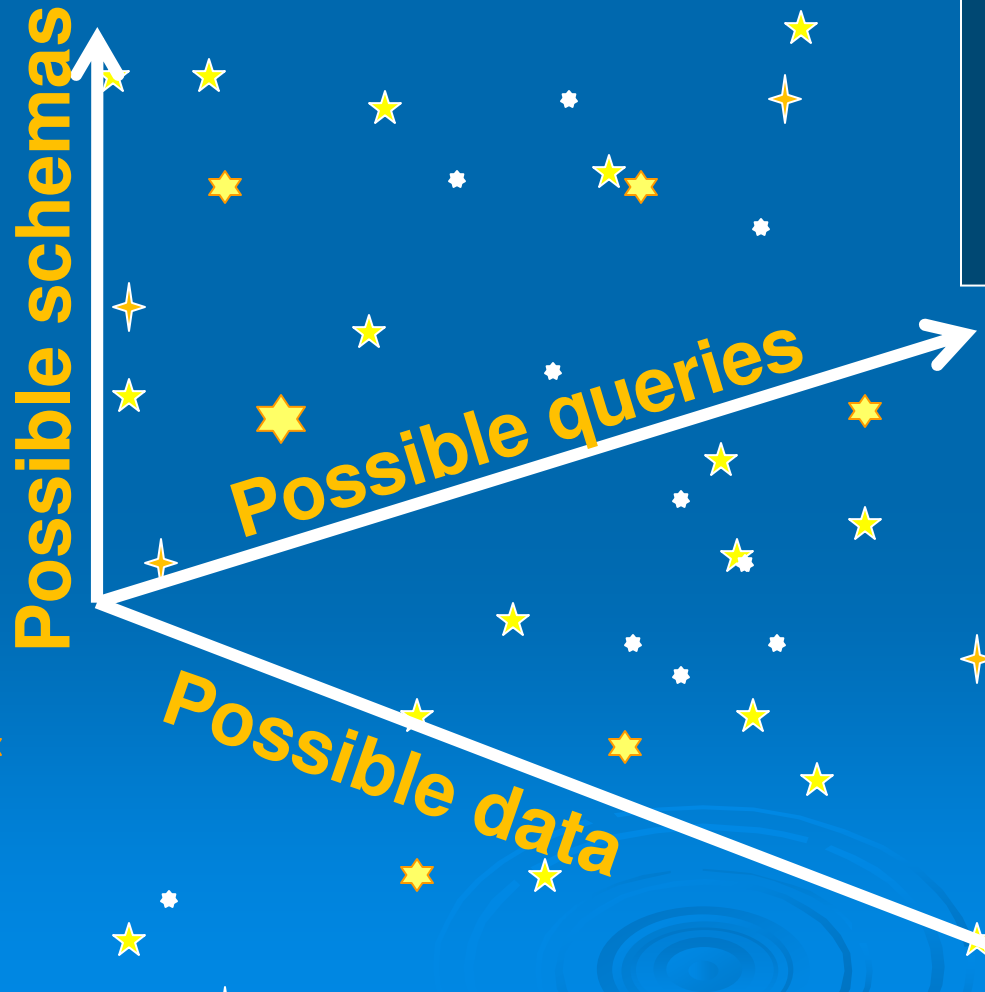
Using the Optimizer to Generate an Effective Regression Suite: A First Step

Murali M. Krishna

Presented by Harumi Kuno

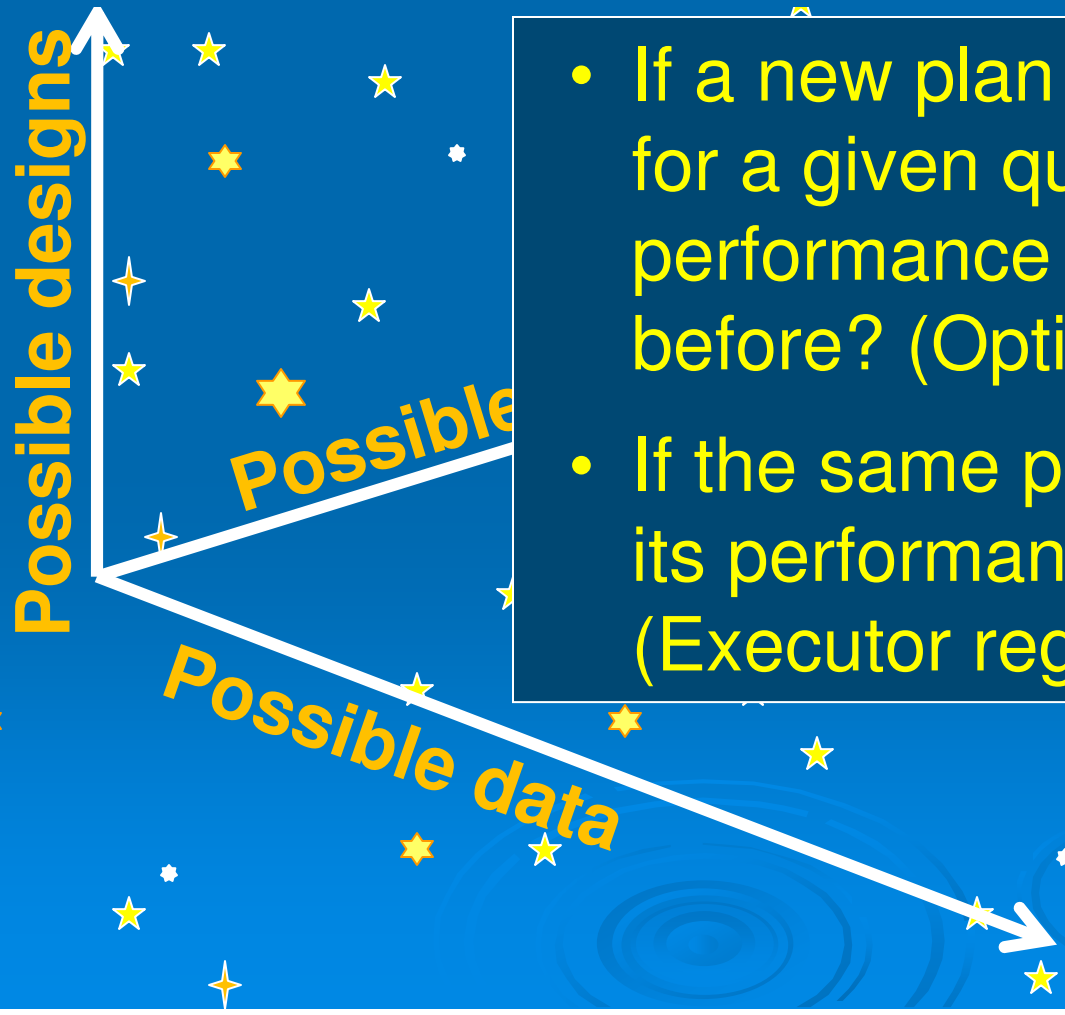
HP

Space of possible queries, schemas, data is practically infinite



Innumerable plans are possible for any query in this space.

Problem: how to construct an effective and economical query optimizer / executor regression suite?



- If a new plan is now produced for a given query, is its performance worse than before? (Optimizer regression)
- If the same plan is produced, is its performance worse? (Executor regression)

Goal: Building the regression test suite should…

- … cover the plan space (code coverage is not enough)
- … be economical not only for the optimizer but also for the execution engine
- … require little knowledge of the optimizer (e.g., without hiring an expensive expert)

Historically, test suite focus has been on known workloads

- Developers and QA add "point" test cases
  - Test specific optimization rules
  - Improve code coverage
  - May be very complex (large search space)

Disadvantages of historical approach

- Can't quantify/evaluate benefit added by new tests
- When a regression occurs (especially if query is complex), hard to find the exact cause; Need an human expert to find and fix problem
- Time to run increases with added tests
- QA groups use code coverage as metric but this does not guarantee plan coverage

Insight

- Enumerate the Optimizer Plan Space (OPS) over physical operators (PO)
- Target OPS (PO, n): All possible legal plans with n operators chosen from PO
  - PO: subset of physical operators used by DBMS that we want to test
  - n will have to be quite small (≤ 10)
- Goal: try to cover this finite space with as few queries as possible – a non trivial problem!

Difficult design decisions

- How many table instances in each query?
  - 4, 10, 20? Plan space grows exponentially
  - A 4 table query may be good enough for our purposes (regression testing the cost model)
  - ***A 20 table query plan is composed of a series of 4 table queries***
- How many physical operators per query?
  - 3 successive hash joins behave very differently from 3 NL joins at exec. time
  - Regression suite should capture execution dependencies (e.g., intra-query resource contention)
  - We settle for a small number of operators in each of our regression queries

Initial Experiment

- Focus on join space only
- Start with a simple skeleton query

```
select T1.a
from T1, T2, T3, T4
where T1.a = T2.a & T2.c = T3.c & T3.d = T4.d &
T1.b ≤ C1 & T2.b ≤ C2 & T3.b ≤ C3 & T4.b ≤ C4
1 ≤ Ci ≤ |T|,  1 ≤ i ≤ 4
```

T1 → T2 → T3 → T4

- Column b is unique and exactly controls # of rows from each instance
- All queries in the regression suite will have the same structure

Assumption

- 5 different join methods:

J1: cartesian product,

J2: regular hash join (where one or both inputs have to be optionally partitioned on the fly),

J3: small table broadcast hash join (here the smaller input is broadcast to all the sites/cpus of the bigger table),

J4: merge join (one or both inputs may need to be sorted), and

J5: index nested loops join.

Enumerating Join Plan Space

- # of non bushy join plans = (4!)*5^3 = 3000
  = subset of OPS ({J1, J2, J3, J4, J5}, 3)
- Insight: We can do with just one table and use 4 instances of the same table
- What is important is the number of rows from each table participating in each query (from the costing perspective) – not the order of the tables
- **Therefore the number of join plans of interest is 3,000/(4!) = 125**

Structure of Table T
(details in paper)

- After some trial and error, we picked T with the following schema
- T (int a, int b, int c, int d, [e, f, g])

a: primary clust. key, hash part. 8 ways on column a

b: unique random secondary key

c: Beta (4, 4): integer-valued, Normal-like over [1 .. 8388608] with mean = 4194419, std. dev = 1398131

d: Beta (2, 0.5): integer-valued, Zipfian-like over [1 .. 8388608] with mean = 6711152 , skew = -1.25

e: uniformly distributed between 1 and 256

f: uniformly distributed between 1 and 4096

g: uniformly distributed between 1 and 65536

Generating the Regression Queries

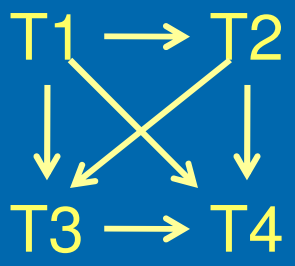
- Ultimately, we want 125 queries, each with a distinct join plan
- We can try to generate them manually – very tedious; will require immense knowledge of the optimizer
- Skeleton query lets us generate many queries with diverse plans ‘automatically’ by varying constants in predicates; No knowledge of optimizer required. No pain and lots of gain ☺

Generating the Regression Queries (2)

- Key idea: Vary the constants C1 thru C4 to 'cover' the 4 dimensional cardinality space
- ***By varying the cardinalities of the participating tables we are essentially covering the cardinality space, which in turn will cover the plan space by letting the optimizer choose among the various join implementations***
- We chose each Ci from these 10 values {1, 10, 100, 1K,10K, 100K, 1M, 2M, 4M, 8M} resulting in 10^4 queries
- These queries were prepared and only 42 (< 35%) distinct plans were found
- Linear Skeleton query not good enough! Only one cartesian product possible; no star joins

Generalized Skeleton Query

select T1.a
from T T1, T T2, T T3, T T4
where T1.a = T2.a &T2.c = T3.c & T3.d = T4.d &
T1.e = T3.e & T1.f = T4.f & T2.g = T4.g &
T1.b ≤ C1 & T2.b ≤ C2 & T3.b ≤ C3 & T4.b ≤ C4



- 6 = (4 choose 2) join predicates
- 2^6 subsets of join predicates possible to allow for all join geometries (includes star joins, 0-3 CPs etc.)
- Number of queries generated = 64*10K = 640,000
- When these queries were compiled, # of distinct plans found was 101 (> 80%) ☺
- Picking the right skeleton query crucial

Generating the Regression Suite

- Pick a small number of queries corresponding to each distinct plan
- How to pick these queries may be a good research topic
- We picked 2 queries for each plan (‘closest’ and ‘farthest’ from the origin)
- Intuition: These represent the two extremes for the plan
- For a good starting point, verify cardinality estimates and optimality of plans in as many cases as possible. Possible because queries are simple.

Comparison with TPC-H/DS

- Compiled queries with at least 4 tables in H and DS (turned off bushy plans)
- Found all sequences of 3 consecutive joins in non bushy plans
- Note: J1-J2-J3-J1-J4-J5 yields 4 sets of 3 joins
- Even so, found 67 distinct plans (incrementally harder to generate additional plans manually)
- Our suite was 50% better ☺ Hard to do manually
- Key point: Our queries are very SIMPLE while DS queries are very COMPLEX (easy to diagnose and fix problems)
- Our suite can be generated easily as well
- Not suggesting that our suite replace H/DS

Conclusions

- Use enumeration to generate a regression suite that covers the plan space – automatically
- No knowledge of optimizer internals required
- BUT need to come up with a good skeleton query
- Validated approach by applying it to join plan space.

Future Work

- How to extend to other SQL operators?
- How to extend to larger number of tables (number of generated queries is ‘doubly exponential’)? If T = 5, generated queries = 102.4 million
- Will sampling work to reduce #of queries?
- How to come up with a better metric that takes into account the quality of plans in the suite?