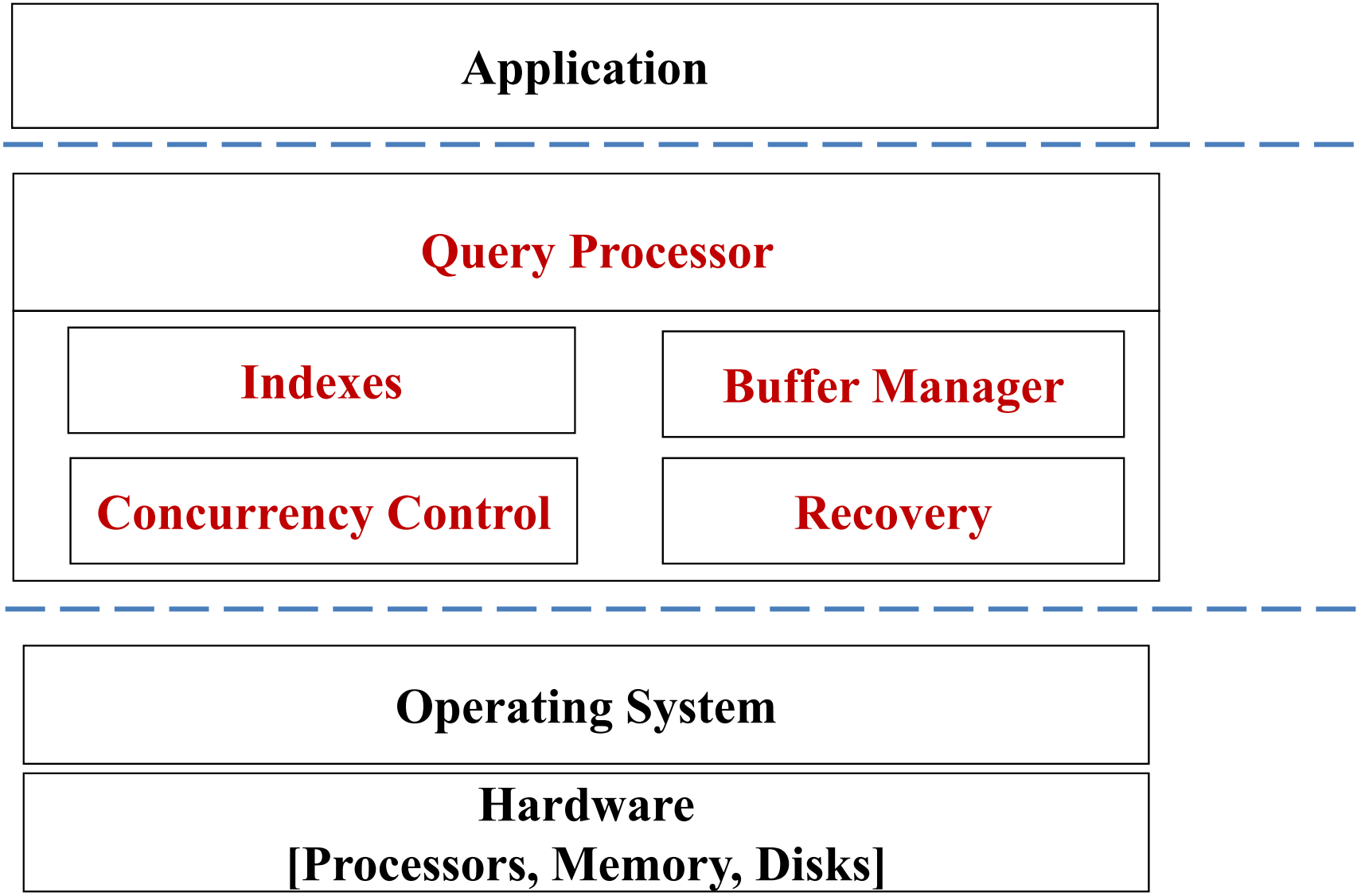
ROBUST QUERY PROCESSING: *Mission Possible!*

Jayant Haritsa
Database Systems Lab
Indian Institute of Science, Bangalore

Relational DBMS

- Workhorse of today's Information Industry
  - Commercial
    - IBM DB2, MS SQL Server, Oracle Exadata, HP SQL/MX
  - Public-domain
    - PostgreSQL, MySQL, Berkeley DB
- Extensively researched for over four decades
  - Journals
    - ACM TODS, IEEE TKDE, VLDBJ, …
  - Conferences
    - ACM SIGMOD, IEEE ICDE, VLDB, EDBT, CIKM, …

Typical RDBMS Engine



Design of RDBMS Engines

- Transaction Processing (ACID)
  - WAL/ARIES for Atomicity/Recovery
  - 2PL for Concurrency Control
- Data Access Methods
  - B-trees/Hashing for Large Ordered Domains
  - Bitmaps for Small Categorical Domains
  - R-trees for Geometric Domains
- Memory Management
  - LRU-k (k=2 balances history and responsiveness)
- Query Processing (SQL)
  - "Black Art"

Query Execution Plans

- SQL is a declarative language
  - Specifies ends, not means

```
select   STUDENT.Name, COURSE.Title
from     STUDENT,  COURSE,  REGISTER
where    STUDENT.RollNo = REGISTER.RollNo   and
           REGISTER.CourseNo = COURSE.CourseNo
```

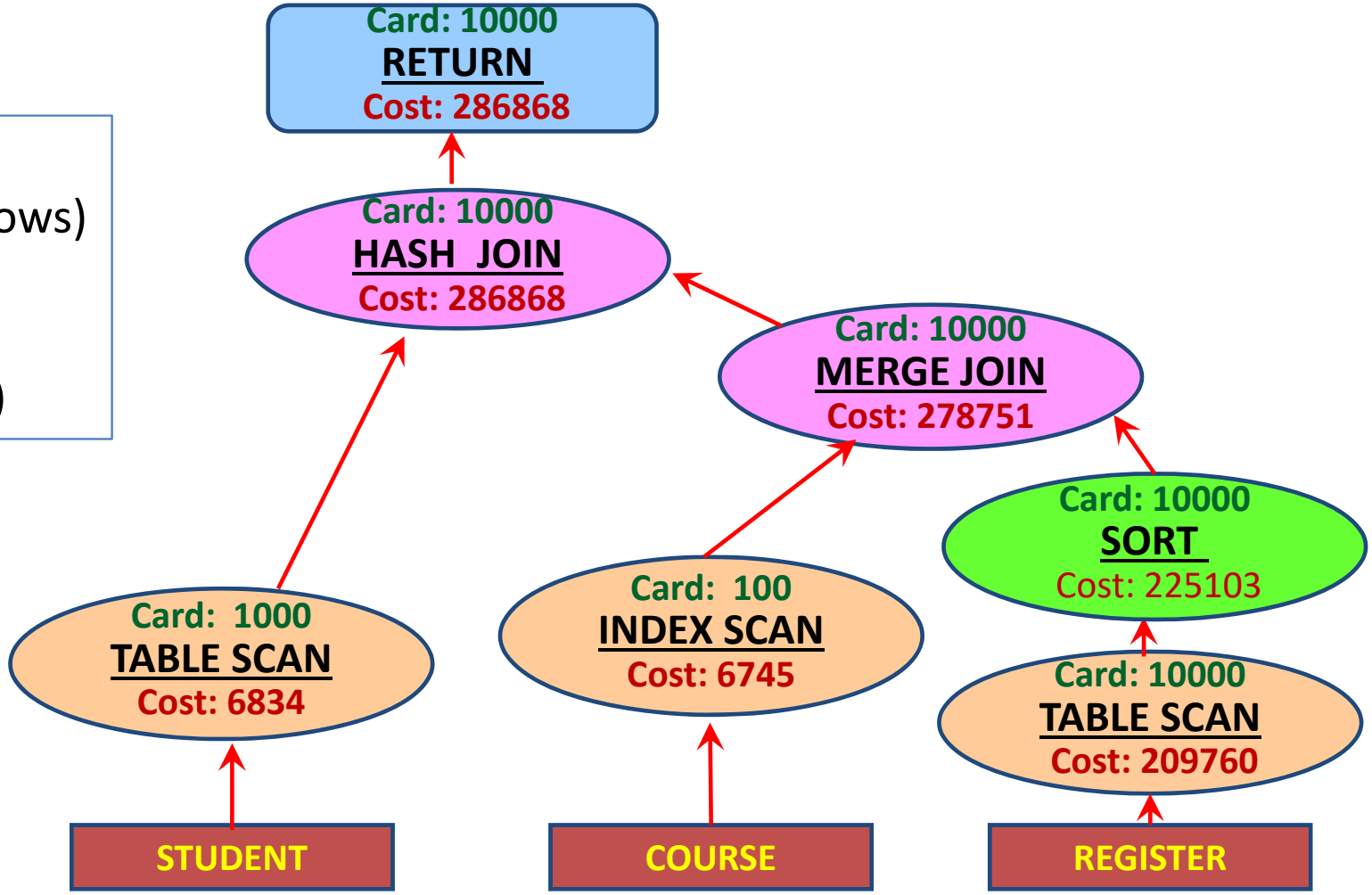
Unspecified: **join order [((S ⋈ R) ⋈ C) or ((R ⋈ C) ⋈ S) ?]**
join technique [Nested-Loops / Sort-Merge / Hash?]

- DBMS query optimizer identifies the optimal evaluation strategy: "query execution plan"

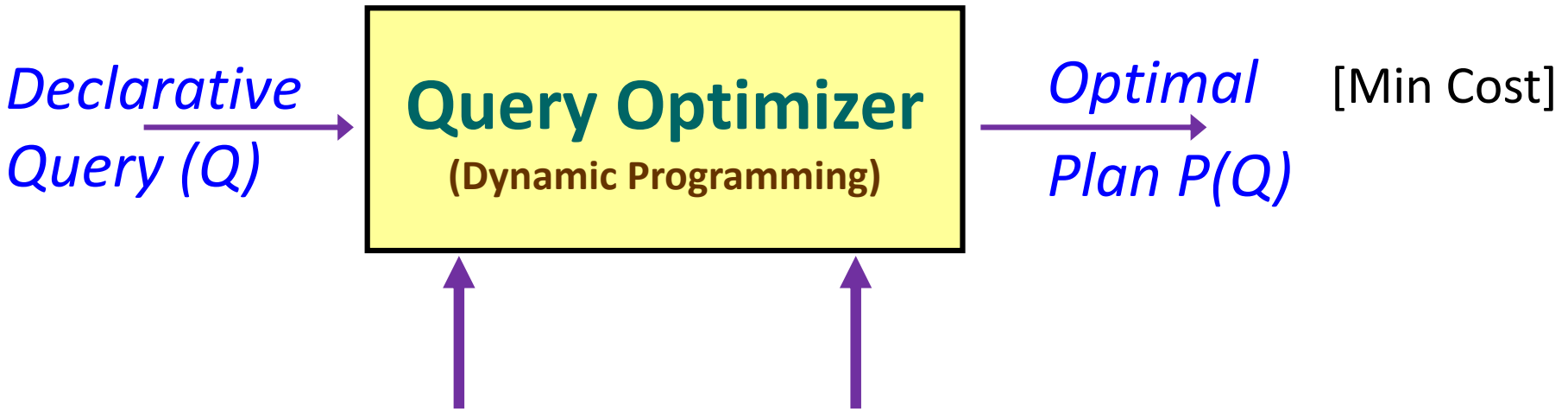
Sample Execution Plan

Card:
Output Cardinality (rows)

Cost:
Execution Cost (time)



Query Optimization Framework



Operator Execution Cost Estimation Model

Function of
Hardware and DB Engine

Operator Output Cardinality Estimation Model

Function of
Data Distributions and Correlations

Run-time Sub-optimality

The supposedly optimal plan-choice may actually turn out to be highly sub-optimal (e.g. a 1000 times worse!) when the query is executed with this plan. This adverse effect is due to errors in:

(a) cost model

- Reasons: Simple linear models, operator-agnostic features, fixed coefficients, system dynamics …

(b) cardinality model

- Reasons: Coarse statistics, outdated statistics, attribute value independence (AVI) assumption, multiplicative error propagation, query construction, …

What have the QP guys been doing all these years?

DB2 Oracle SQL Server

"Elephants"† are highly sensitive animals!

(† Stonebraker-speak for enterprise DBMS)

Cardinality Sensitivity Example

EMPLOYEE

EID	Name	Age
1	Cohen	25
2	Giuliani	25
3	Manafort	25
4	Melania	25
5	Ivanka	25
6	Donald	25
7	Jared	25
….	….	25
10^9	Eric	25

MANAGER

MID	Name	Age
1	Trump	50
2	Pence	50
3	Mnuchin	50
4	Shanahan	50
5	Whitaker	50
6	Bernhardt	50
7	Perdue	50
….	….	50
10^6	Ross	50

EMPLOYEE.AGE ⋈ MANAGER.AGE

- Output cardinality of the join is **ZERO**
- One new employee aged **50** joins the company
- Output cardinality of the join jumps to a **million!**
- No summary mechanism can capture such **"nanoscopic"** changes

Proof by Authority [Guy Lohman, IBM]

Snippet from April 2014 Sigmod blog post on

"Is Query Optimization a "Solved" Problem?"

The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities. The cardinality model can easily introduce errors of many orders of magnitude! With such errors, the wonder isn't "Why did the optimizer pick a bad plan?" Rather, the wonder is "Why would the optimizer ever pick a decent plan?"

Sound-bites

- Little difference between worst-case and average-case in Query Processing

- It is far easier to win money at the Macau casinos than to get query processing right!

Prior DB Research (lots!)

- Sophisticated estimation techniques
  - SIGMOD 1999/2010, VLDB 2001/2009/2011, …, CIDR 2019
  - e.g. wavelet histograms, self-tuning histograms, deep learning
- Selection of stable plans
  - SIGMOD 1994/2005/2010, PODS 1999/2002, VLDB 2008, …, VLDB 2017
  - e.g. Variance-aware plan selection
- Runtime re-optimization techniques
  - SIGMOD 1998/2000/2004/2005, …, arXiv 2019 [Stonebraker et al]
  - e.g. POP (progressive optimization), RIO (re-optimizer)

Several novel ideas and formulations,
but are they robust?

Is there any hope?

Over last decade, several promising advances that collectively promise to soon make robustness a contemporary reality – we will survey these techniques in the tutorial.

Thanks

Gratefully acknowledge presentation material provided by

- Renata Borovica-Gajic (U of Melbourne, Australia)
- Goetz Graefe (Google Madison Labs, USA)
- Thomas Neumann (TU Munich, Germany)
- Wolf Roediger (Tableau, Germany)
- Wentao Wu (Microsoft Research, USA)
- Srinivas Karthik (IISc Bangalore, India)

QP Robustness

Importance of Robustness

- Dagstuhl Seminars
  - 2010 (#10381), 2012 (#12321), 2017 (#17222)
- ICDE 2011 panel on Robust Query Processing
- Immediate relevance to database vendors
- Huge impact on database users and customers
- Critical for Big Data world!

ROBUSTNESS DEFINITION

- Multiple perspectives, no consensus
  - If worst-case performance is improved at the expense of average-case performance, is that acceptable?
  - Is it to be defined on a query instance basis, or “in expectation”?
  - …
- Ultimately, robustness definition is application dependent
- Graceful performance profile – no “cliffs”
- Seamless scaling with workload complexity, database size, distributional skew, join correlations
- Provable guarantees on worst-case performance (relative to an offline ideal that makes all the right decisions)

TUTORIAL OUTLINE

- Stage 1: Robust Operators
- Stage 2: Robust Plans
- Stage 3: Robust Query Execution
- Stage 4: Robust Cost Models
- Stage 5: Future Research Directions

Stage 1: Robust Operators

Approaches

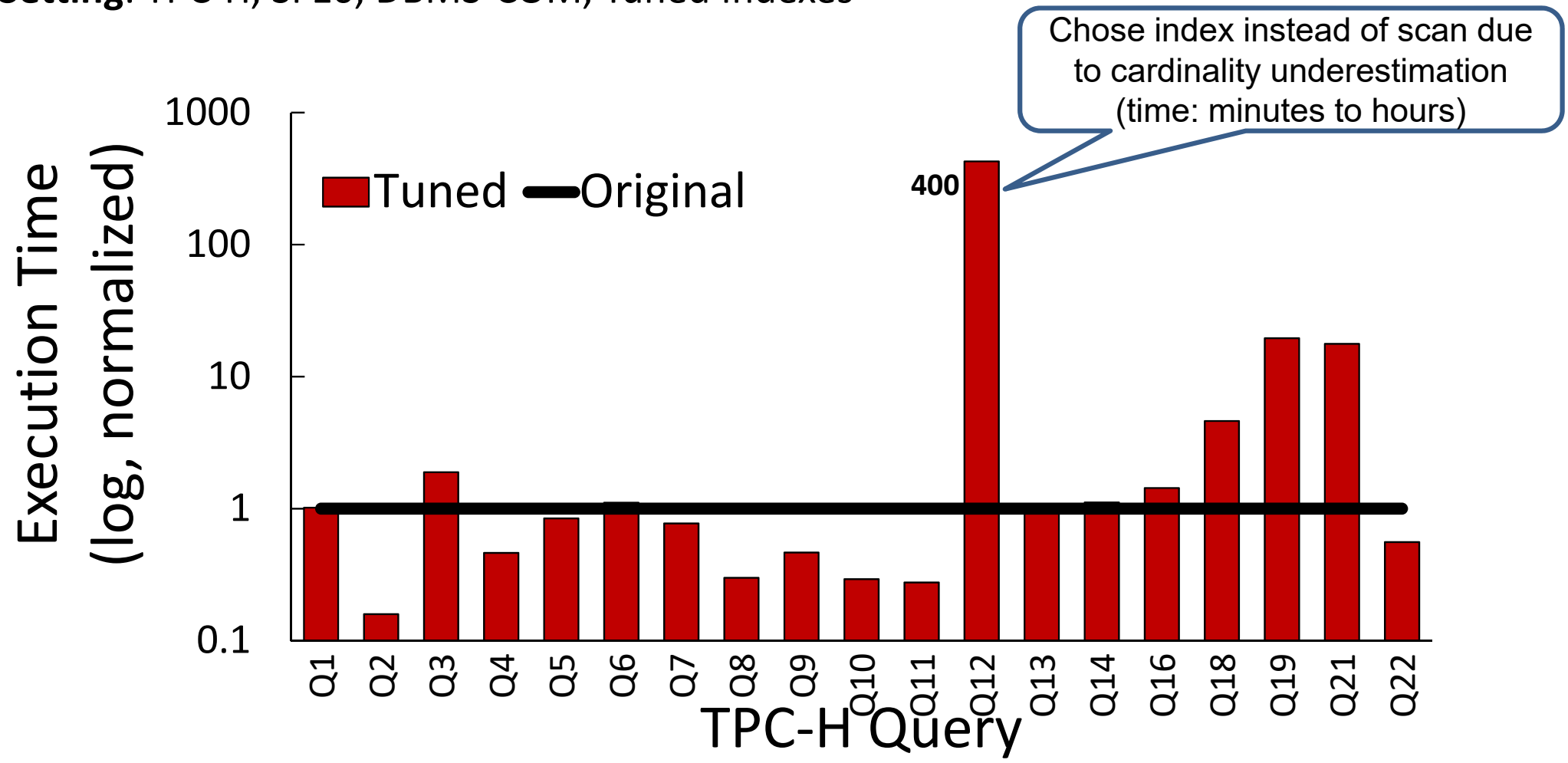
- Unified operators
  - *Basic Idea: If choice is eliminated, cannot make mistakes, by definition! The key challenge is retaining, in the absence of choice, comparable performance to the multi-choice environment.*
  - Smooth Scan (ICDE 2015, VLDBJ 2018 [7])
    - Unifying Table Scan, Index Scan
  - G-join (CS R&D, 2012 [16])
    - Unifying Nested-loops, Sort-merge, Hash-join
- Scaling operators
  - Flow-join (ICDE 2016 [36])
    - Broadcast "heavy hitter" tuples to handle skew in distributed systems

Smooth Scan

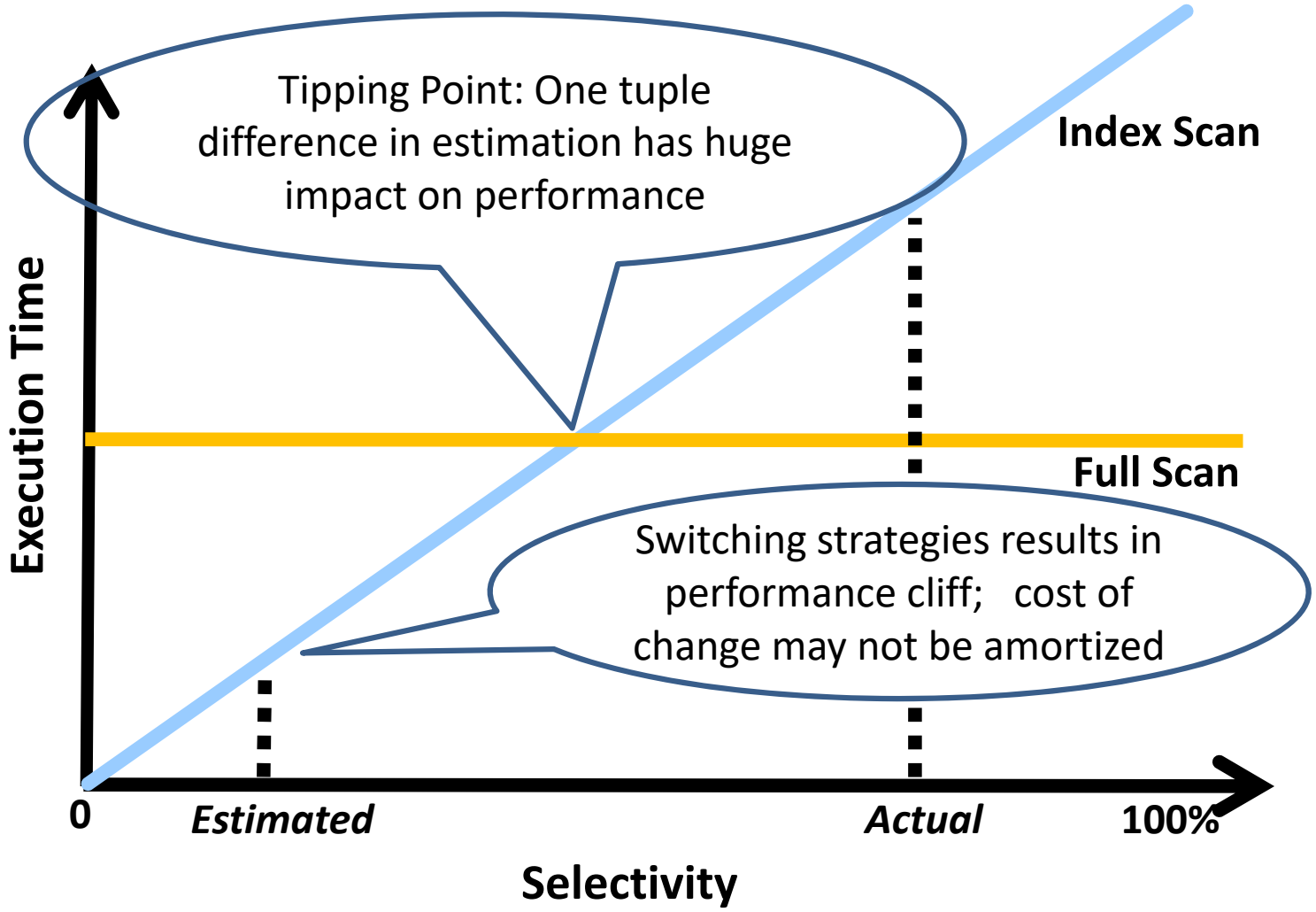
(Morph between Index Scan **and** Sequential Scan**)**

Sub-optimal Access Paths: Example

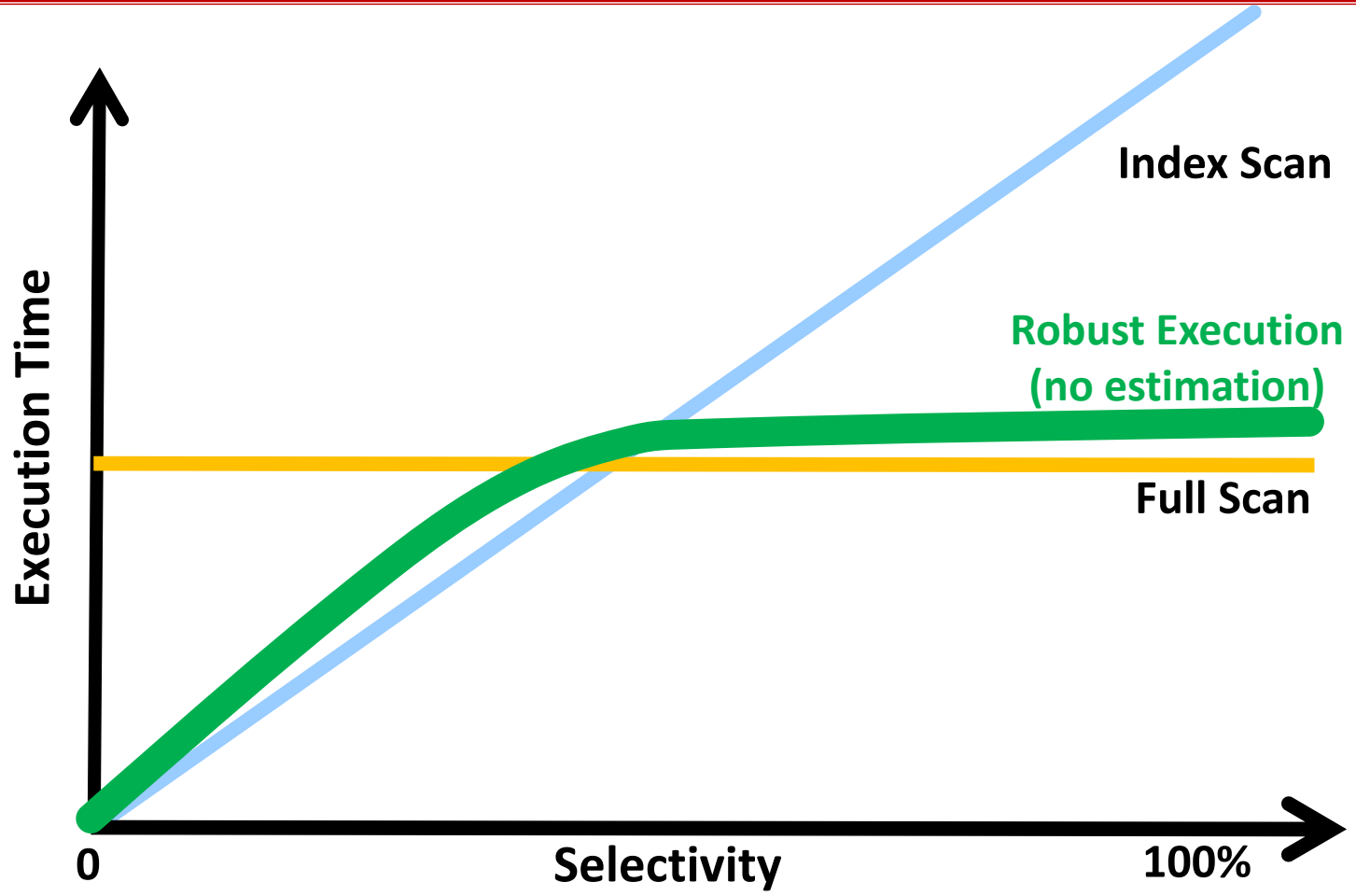
Setting: TPC-H, SF10, DBMS-COM, Tuned Indexes



Access path selection problem



Quest for robust access paths



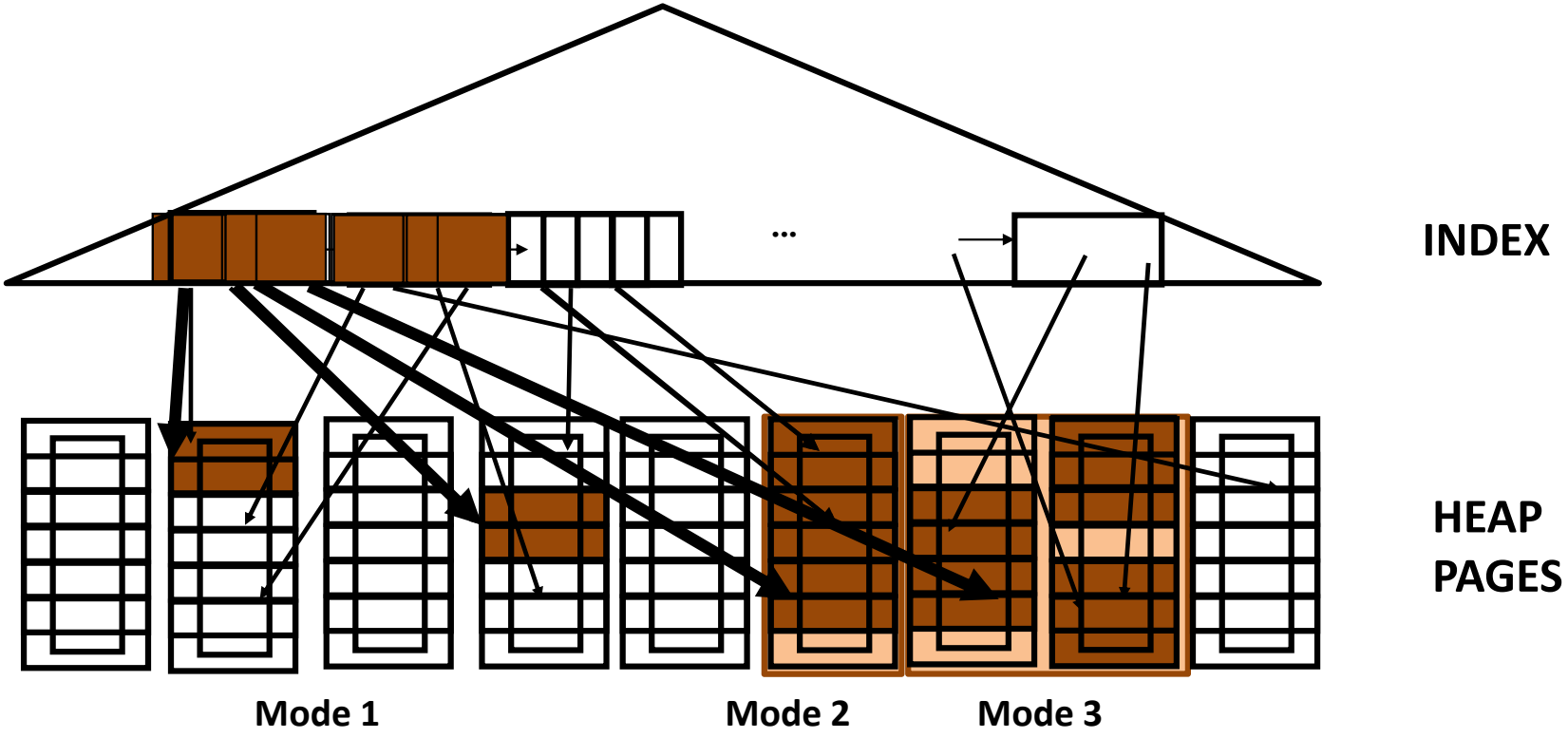
Near-optimal (= min(IS,FS)) throughout entire selectivity range

Smooth Scan in a nutshell

- Statistics-oblivious access path
- Learn result distribution at run-time
- Adapt as you go

Morphing mechanism

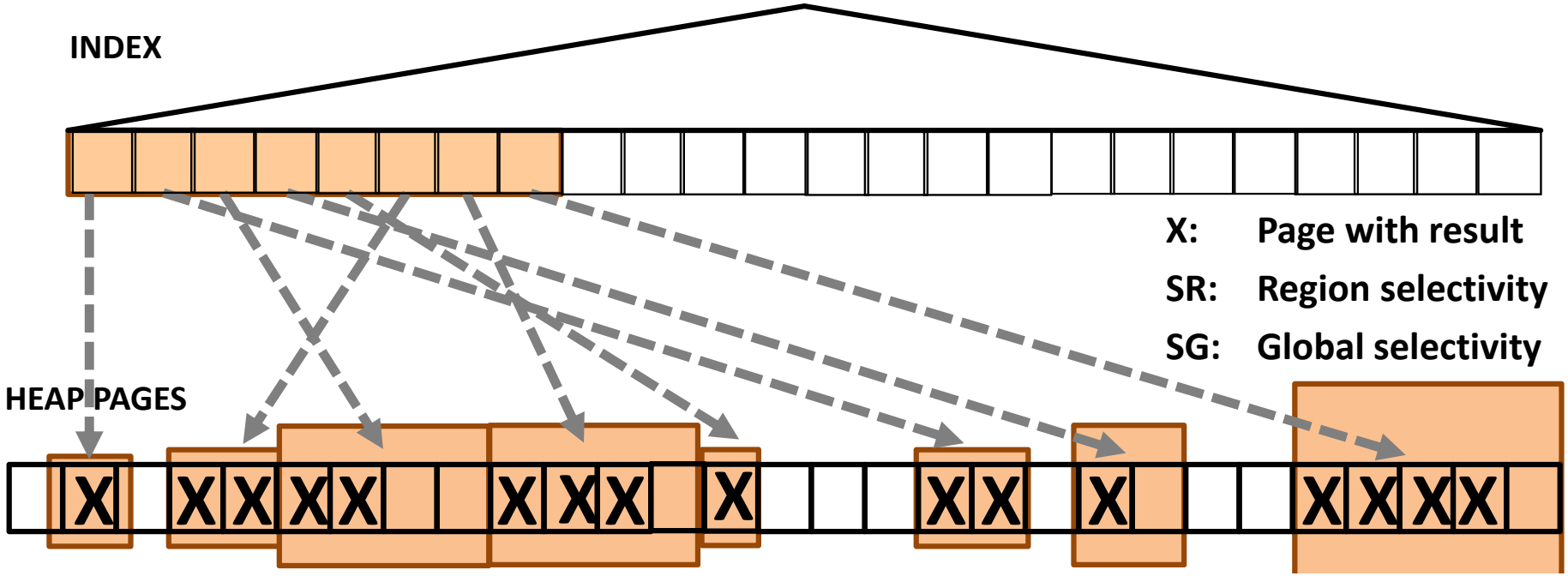
- Modes:
  1. **Index Access:** Traditional index access
  2. **Entire Page Probe:** Index access probes entire page
  3. **Gradual Flattening Access:** Probe adjacent region(s)



Morphing policies

- Greedy
- Selectivity Increase
- Elastic

Selectivity increase → Mode Increase
SEL_region > SEL_global
Selectivity decrease → Mode Decrease
SEL_region < SEL_global



Region snooping = Selectivity driven adaptation

Smooth Scan benefits

	Index Scan	Full Scan	Sort Scan	Smooth Scan
Avoid repeated accesses	✗	✓	✓	✓
Fast sequential I/O	✗	✓	✓	✓
Avoid full table read	✓	✗	✓	✓
Tuples pipelining	✓	✓	✗	✓

Sort Scan: Get all qualifying RIDs from the index, sort them, and then sequentially retrieve the records.

Experimental setup

Hardware:

2 Intel Xeon 6-core CPU @2.8 GHz, 48GB RAM

HDD: I/O transfer rate 120 MB/s, Random vs. Sequential ratio = 10

Software:

PostgreSQL 9.2.1: Index Scan, Full Scan, Sort Scan, Smooth Scan

Workload:

TPC-H: SF 10

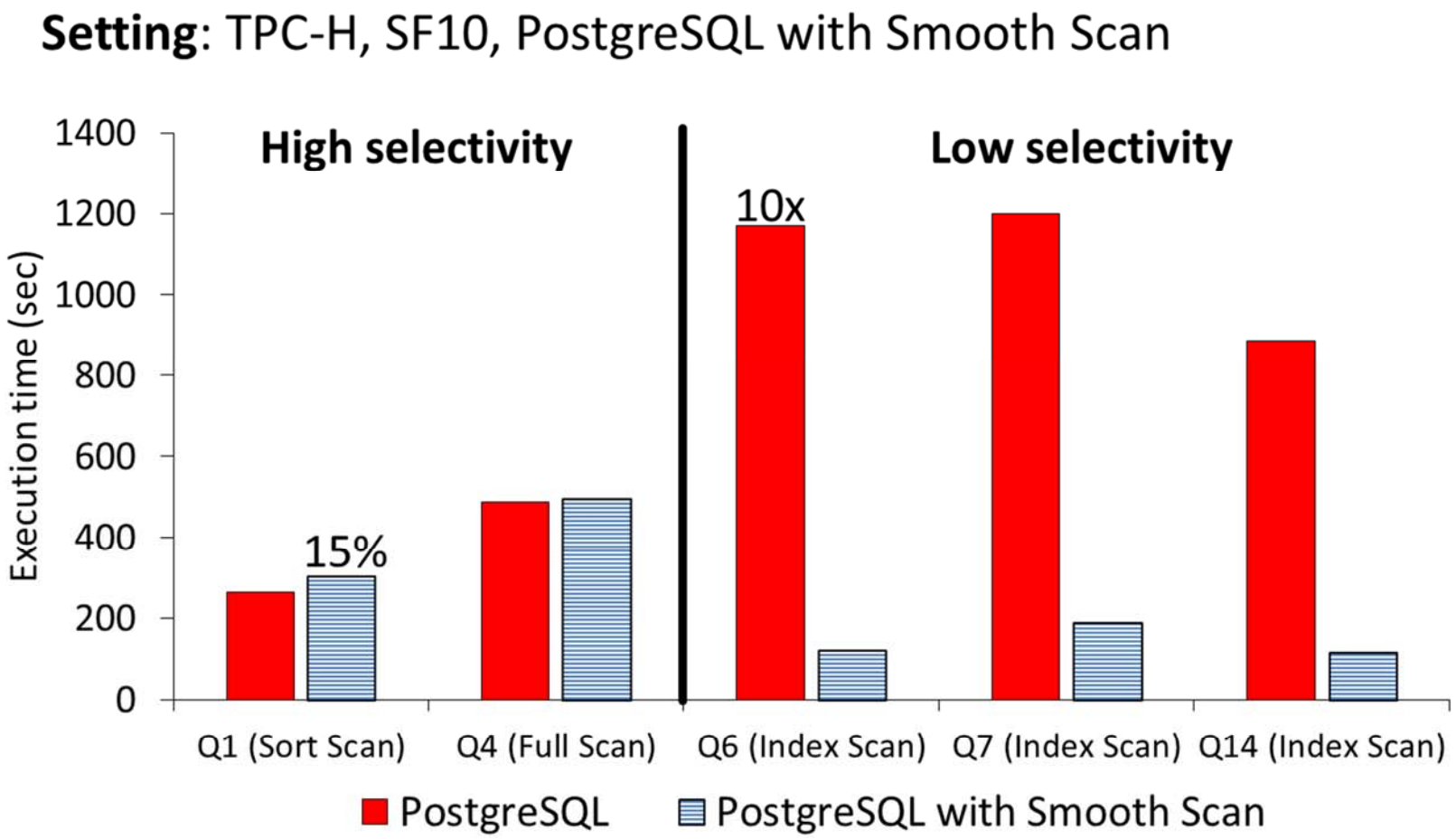
Micro-benchmark: 400M tuples, 10 columns random (1 – 10^5), 25GB

Q1: select * from relation where c2 >= 0 and c2< X% [order by c2];

Experimental Condition:

Cold file system cache

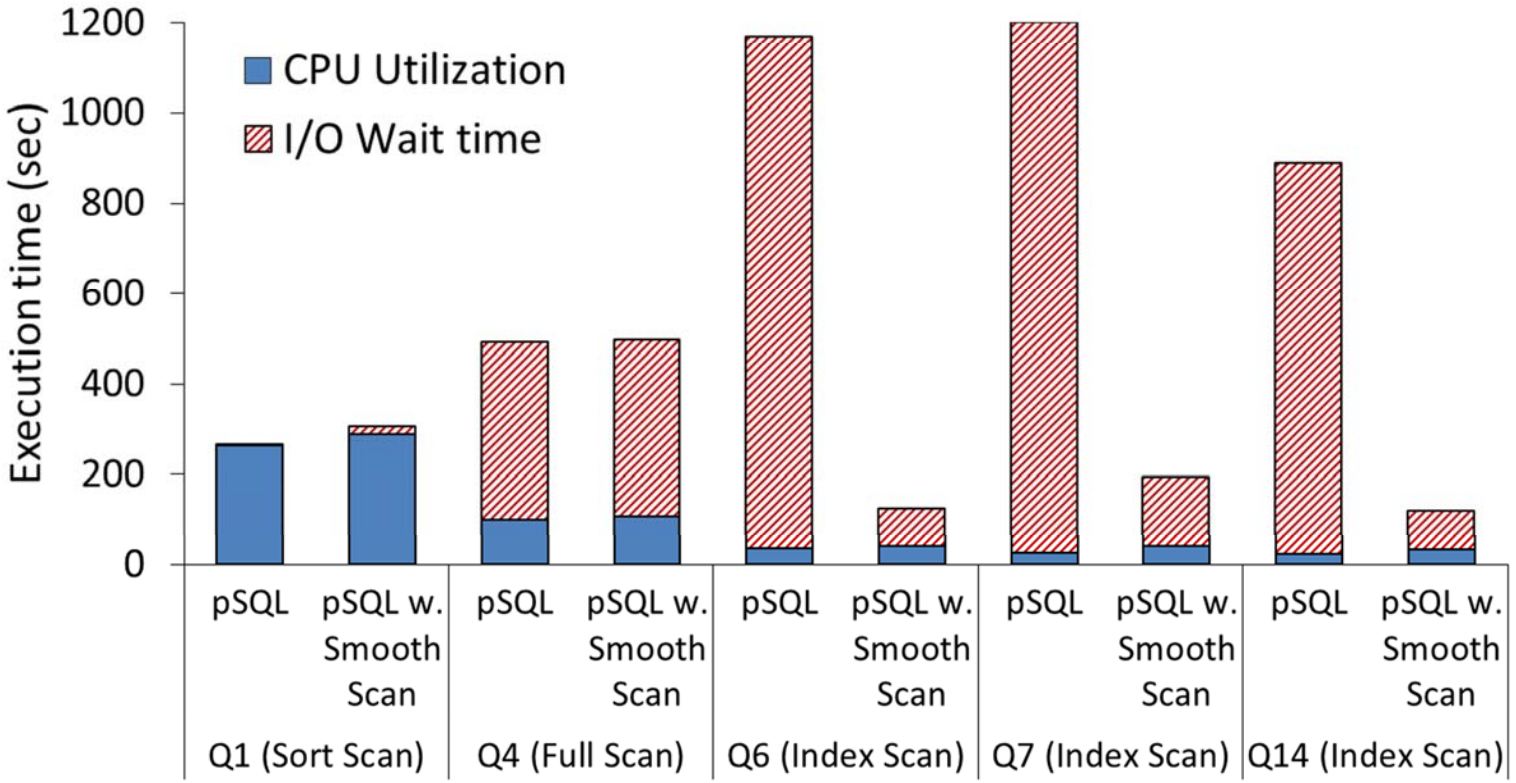
TPC-H with Smooth Scan



Robust execution for all queries

I/O drill-down

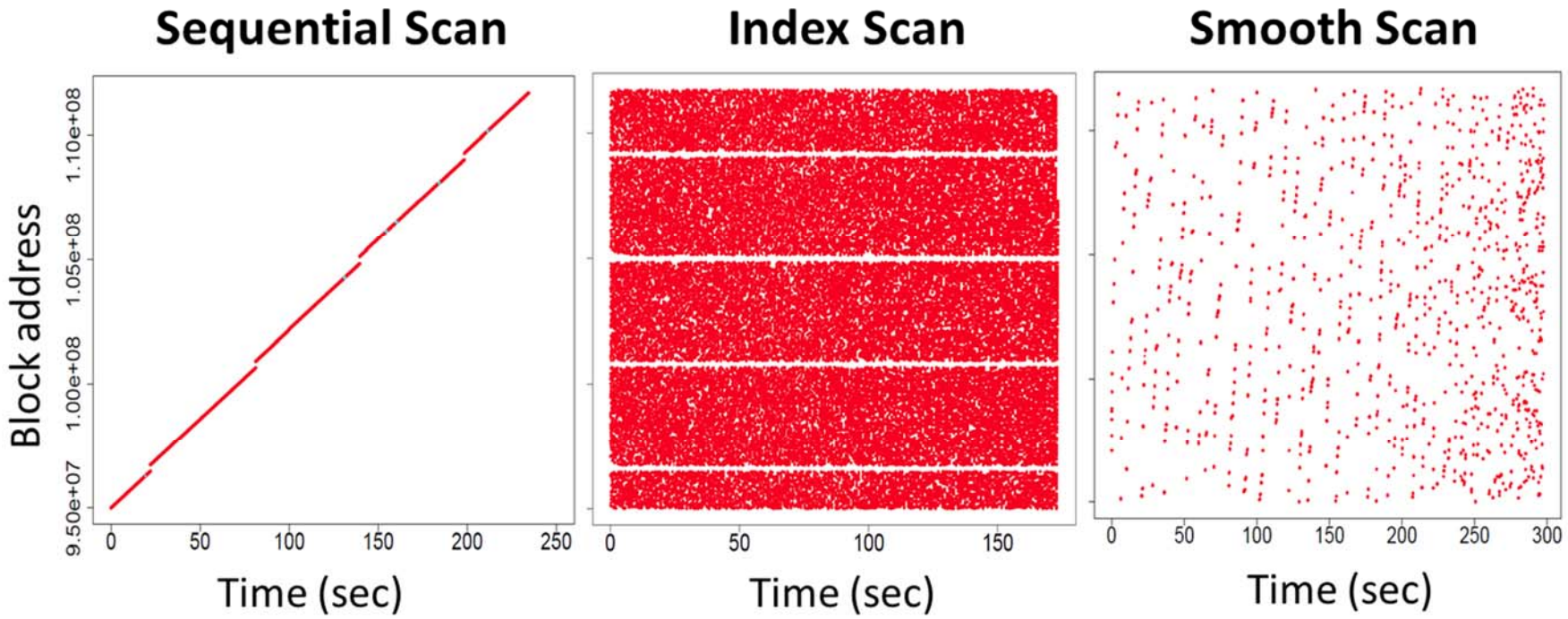
	Q1		Q4		Q6		Q7		Q14	
	pSQL	Smooth S.	pSQL	Smooth S.	pSQL	Smooth S.	pSQL	Smooth S.	pSQL	Smooth S.
# I/O Requests (K)	70	77	224	235	566	95	745	124	416	87



Smooth Scan significantly decreases I/O wait time

Snooping I/O access

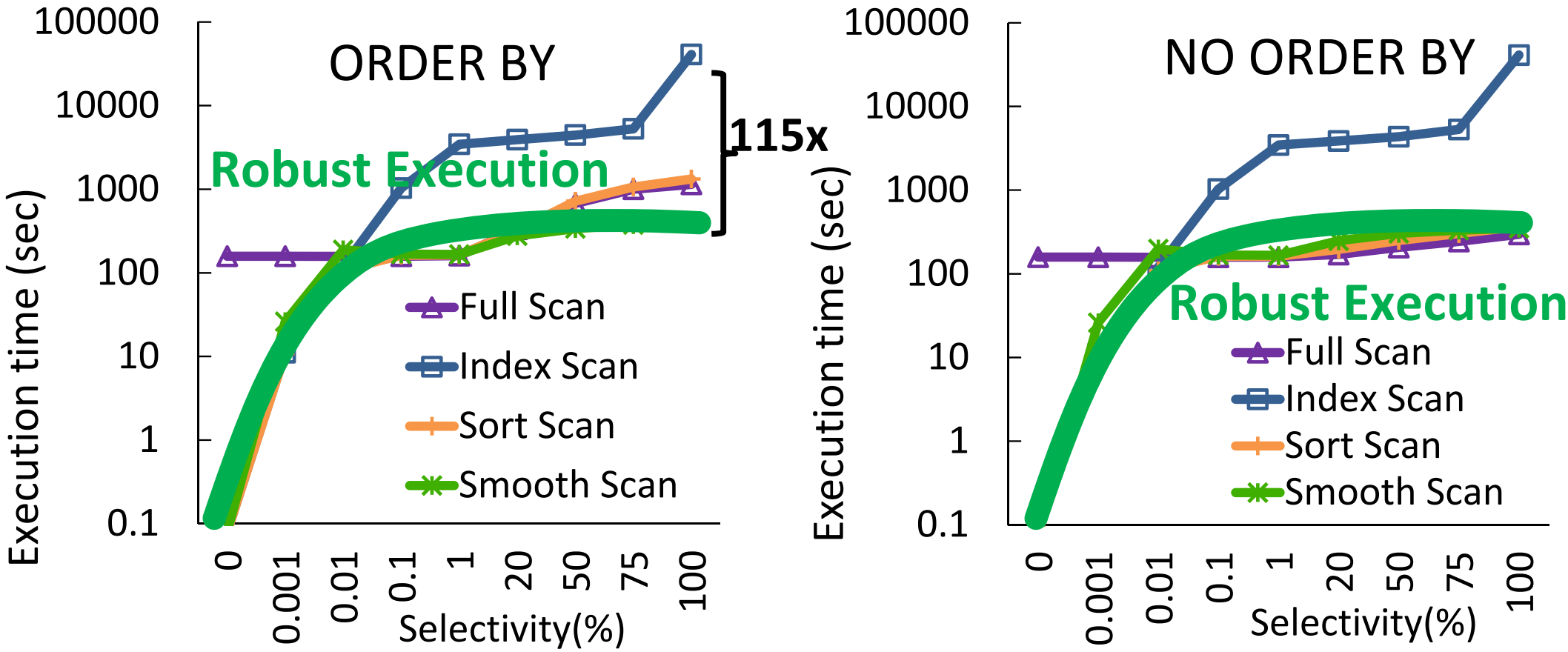
Setting: TPC-H, Q1, Lineitem table, iosnoop tool



Smooth Scan reduces random I/O requests

Adaptivity over selectivity range

Setting: Micro-benchmark, Q1 (w. and w/o. order), Selectivity 0-100%



Near-optimal performance throughout entire range

Performance Guarantee

Ideal: SortScan without Sorting Cost – i.e. sequentially read only the relevant pages.

$$\frac{SmoothScan}{Ideal} \leq (1+ \frac{\text{rand\_io\_cost}}{\text{seq\_io\_cost}})$$

For representative HDD parameters, factor is 11, while for SSD, factor is 6.

Limitations

- Several book-keeping data structures required to maintain result semantics (duplicates/ordering)
  - Page ID cache (to not process page twice)
  - Tuple ID Cache (to not produce same tuple twice)
  - Result Cache (for ordered output)
  - Memory Management (for above structures)
- Requires changes to database engine internals

G-join: Generalized Join

(Morph across Indexed-NL Join, Sort-Merge Join, Hybrid-Hash Join)

Comparative Algorithm Strengths

	INL Join	SM Join	HH Join	G-join
Sorted inputs		✓		✓
Indexed input	✓			✓
Input size differences			✓	✓

Basic Idea

- Implement Sort-Merge using concepts from Hash-Join
- If inputs are already sorted, just do Merge Join
- If inputs are not sorted, create internally sorted runs as usual for both inputs, but do not carry out merging steps.

  Instead, similar to hash partitions, store "key-covering pages" from the small-input (**R**) in a buffer pool, and a single buffer page for the large-input (**S**). Dynamically expand the **R** buffer pool until it key-covers the buffer page of **S** – then join the memory-resident pages. After this is done, bring the next **S** page into memory. Shrink the **R** buffer pool if any page goes below the key coverage range.

G-join: Phase 1

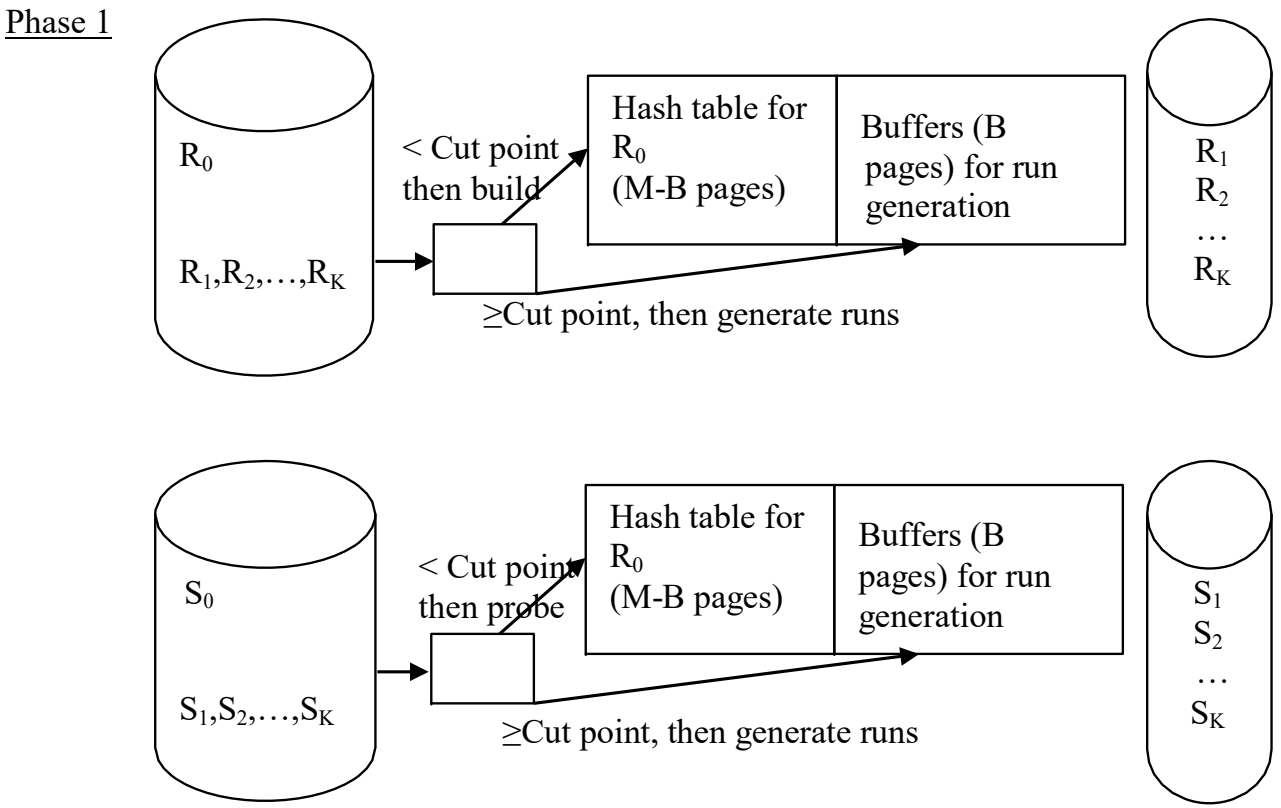


Figure 3.1 Phase 1 of G-Join

G-join: Phase 2

LK: Low Key. For example, $LK_{1.1}$ represents the Low Key from Run#1 Page #1

HK: High Hey. For example, $HK_{2.0}$ represents the High Key from Run#2 Page#0

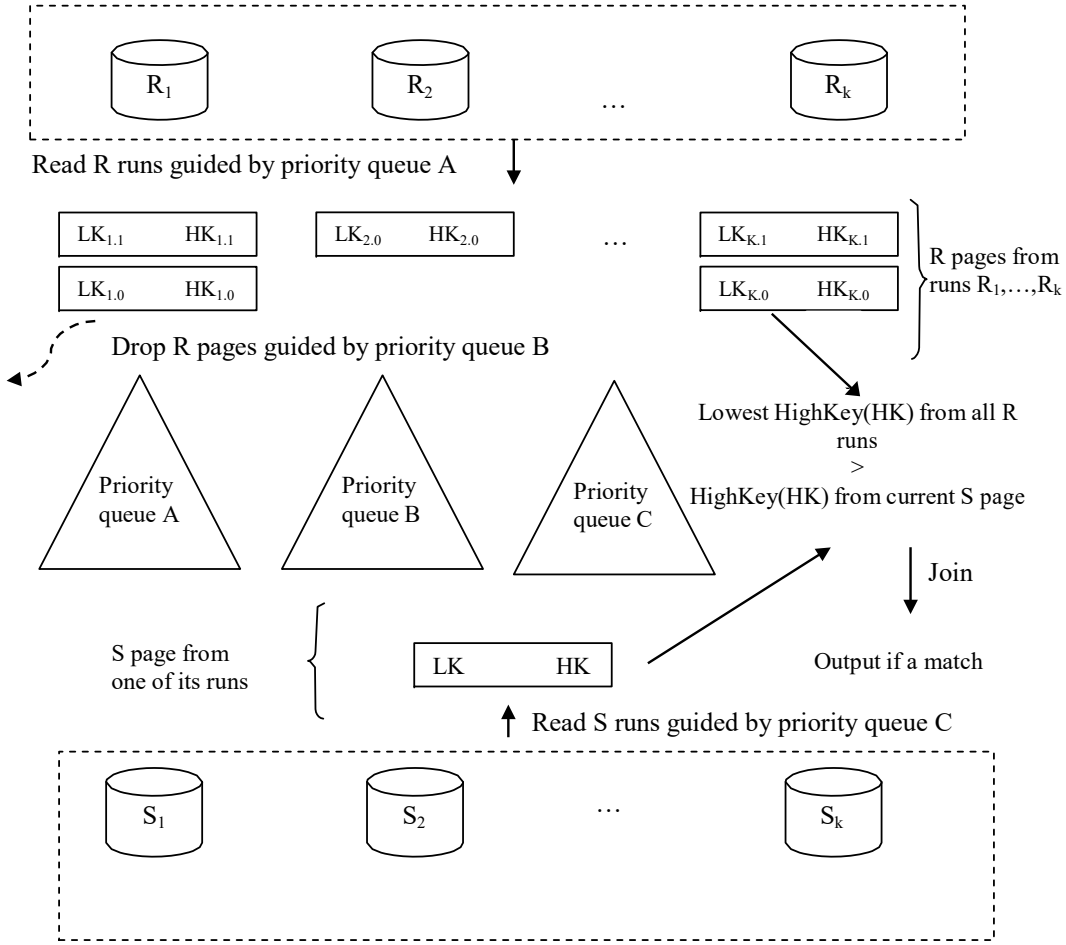
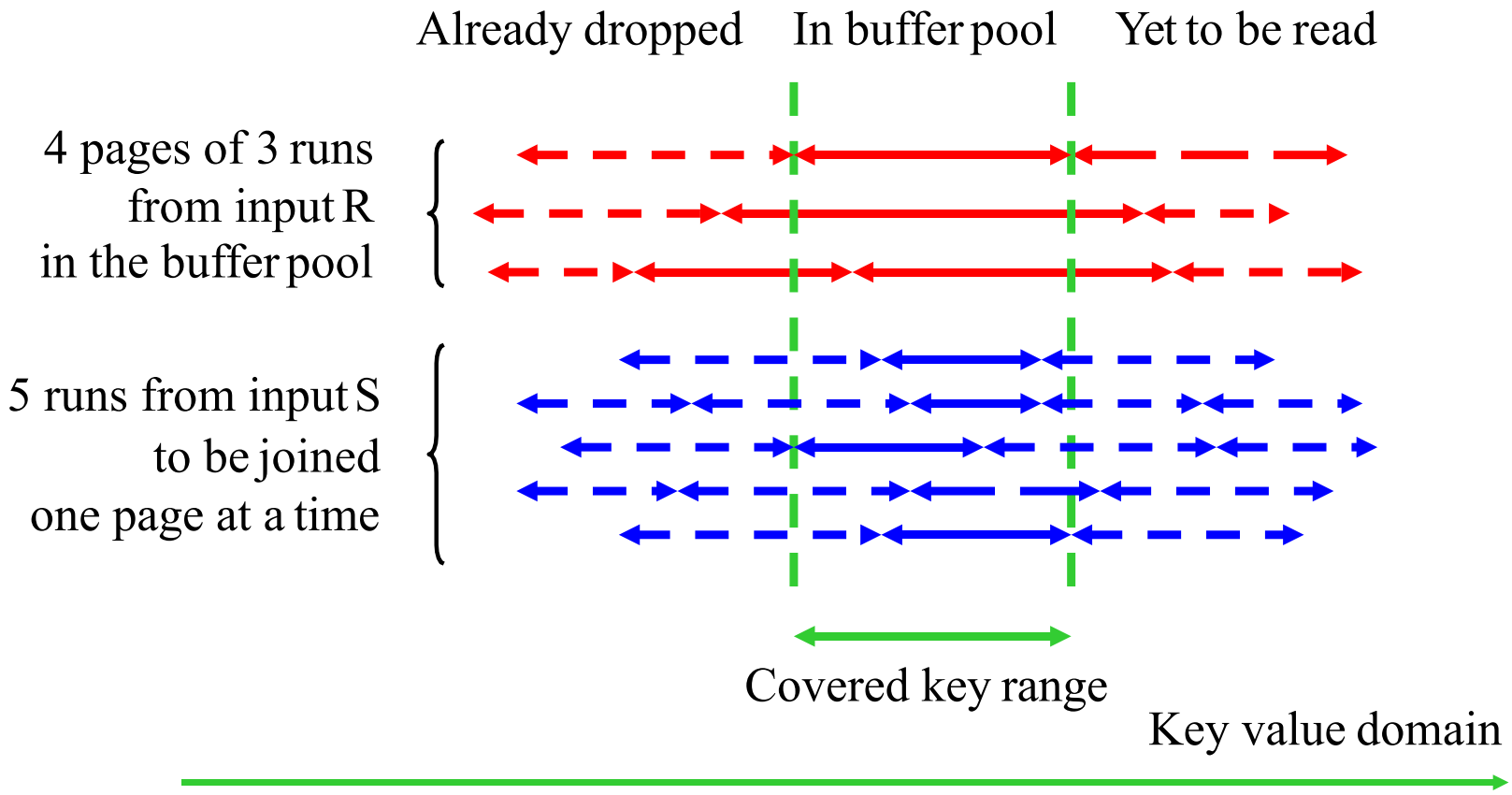
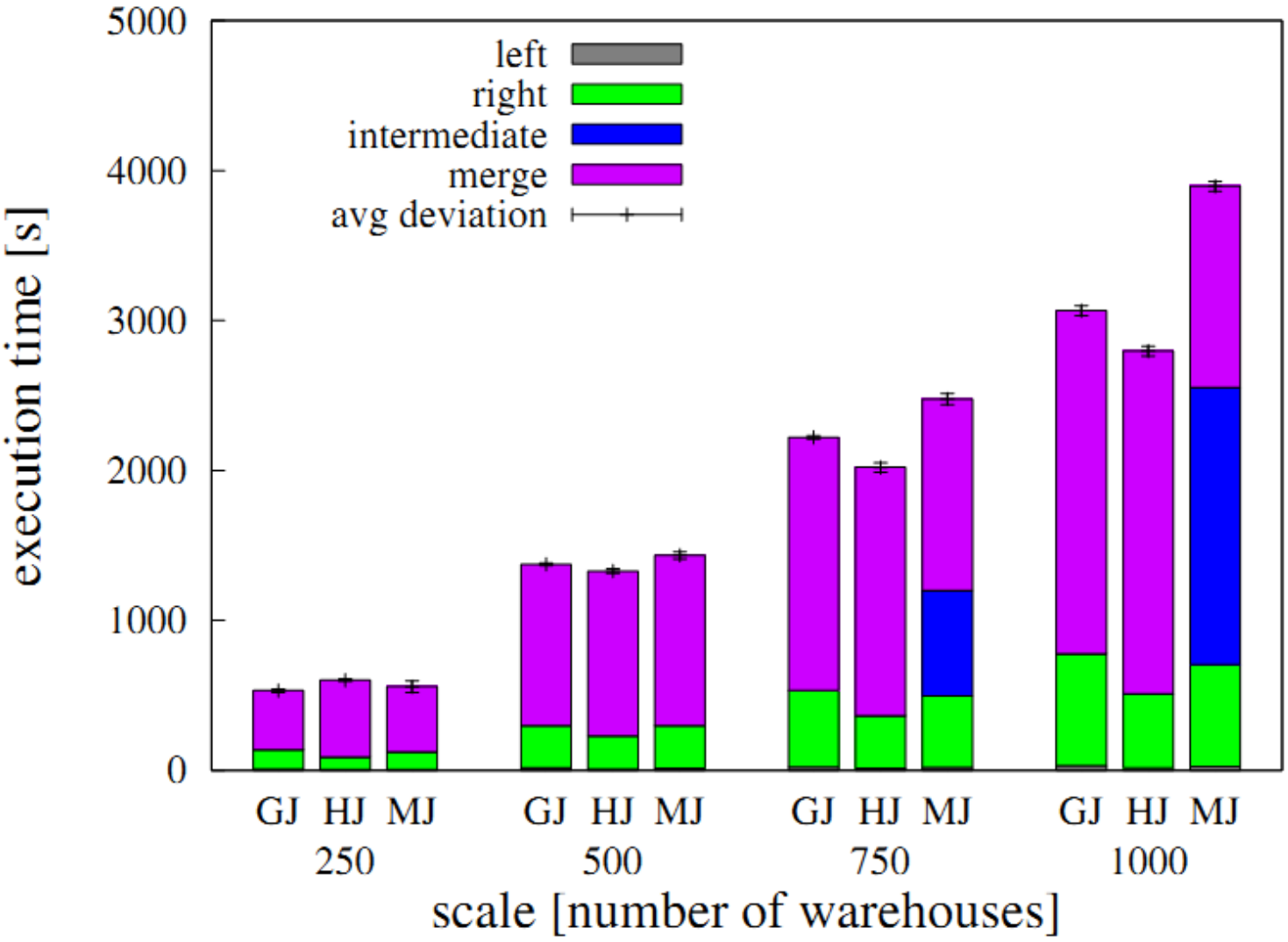


Figure 3.2 Phase 2 of G-Join

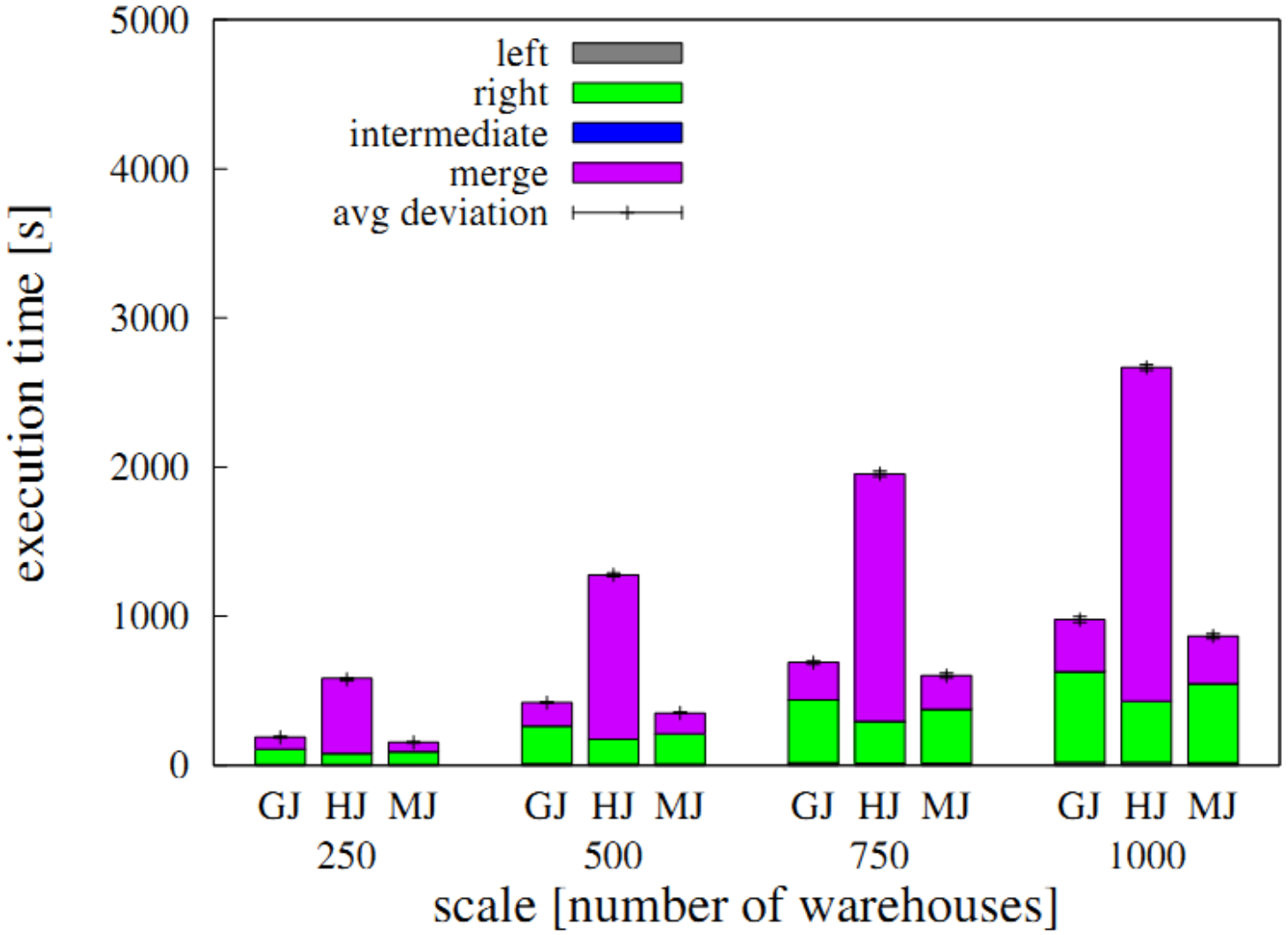
Merge algorithm illustrated



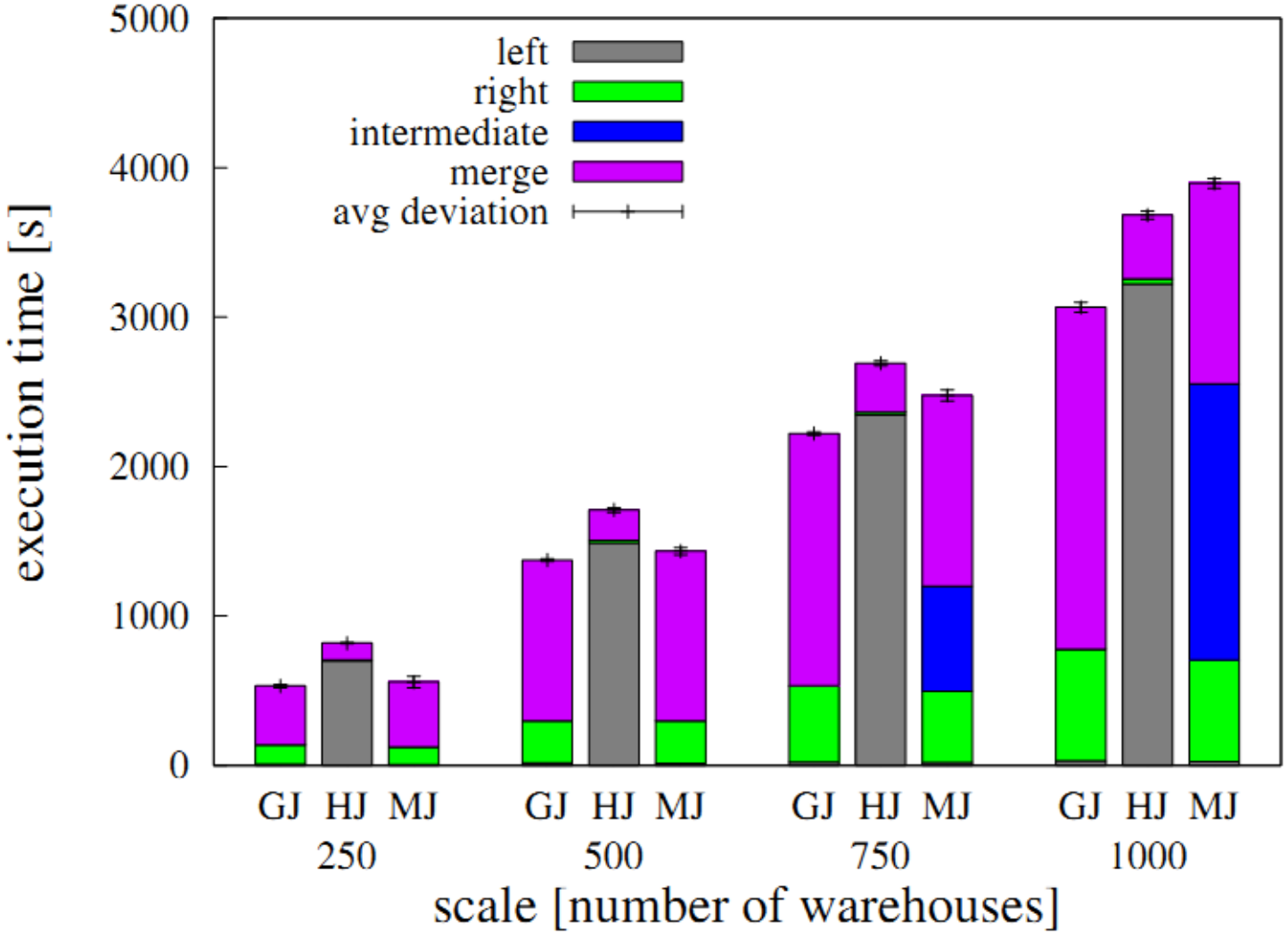
Unsorted inputs



~Sorted inputs



Erroneous optimizer choices



Performance

- Similar "unified" algorithms available for grouping and set operations. [16]

- Performance Guarantee:
  - First-cut theoretical analysis shows rough equivalence to best of existing algorithms

- Limitations:
  - Skew in sizes of runs and skew in key value distribution can adversely impact performance

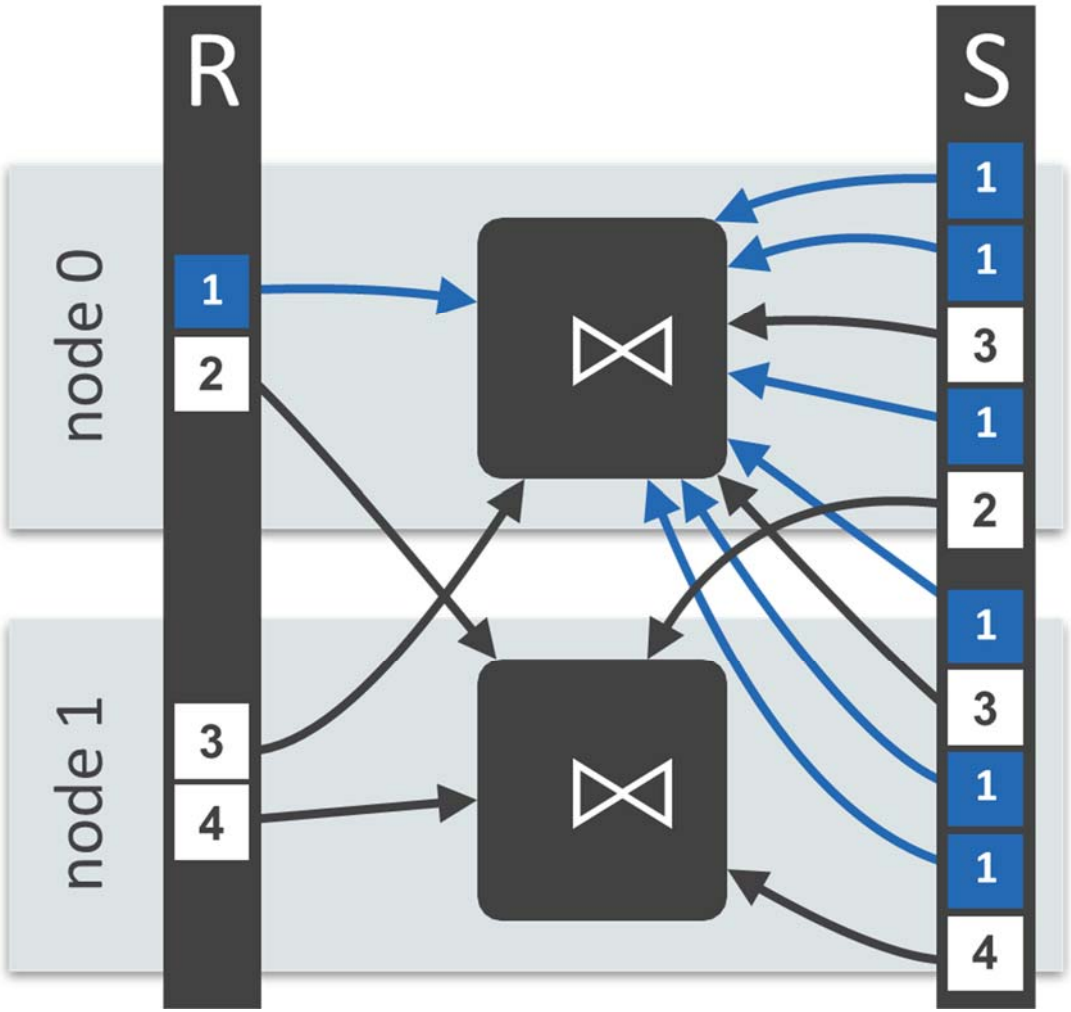
Flow Join

Flow Join

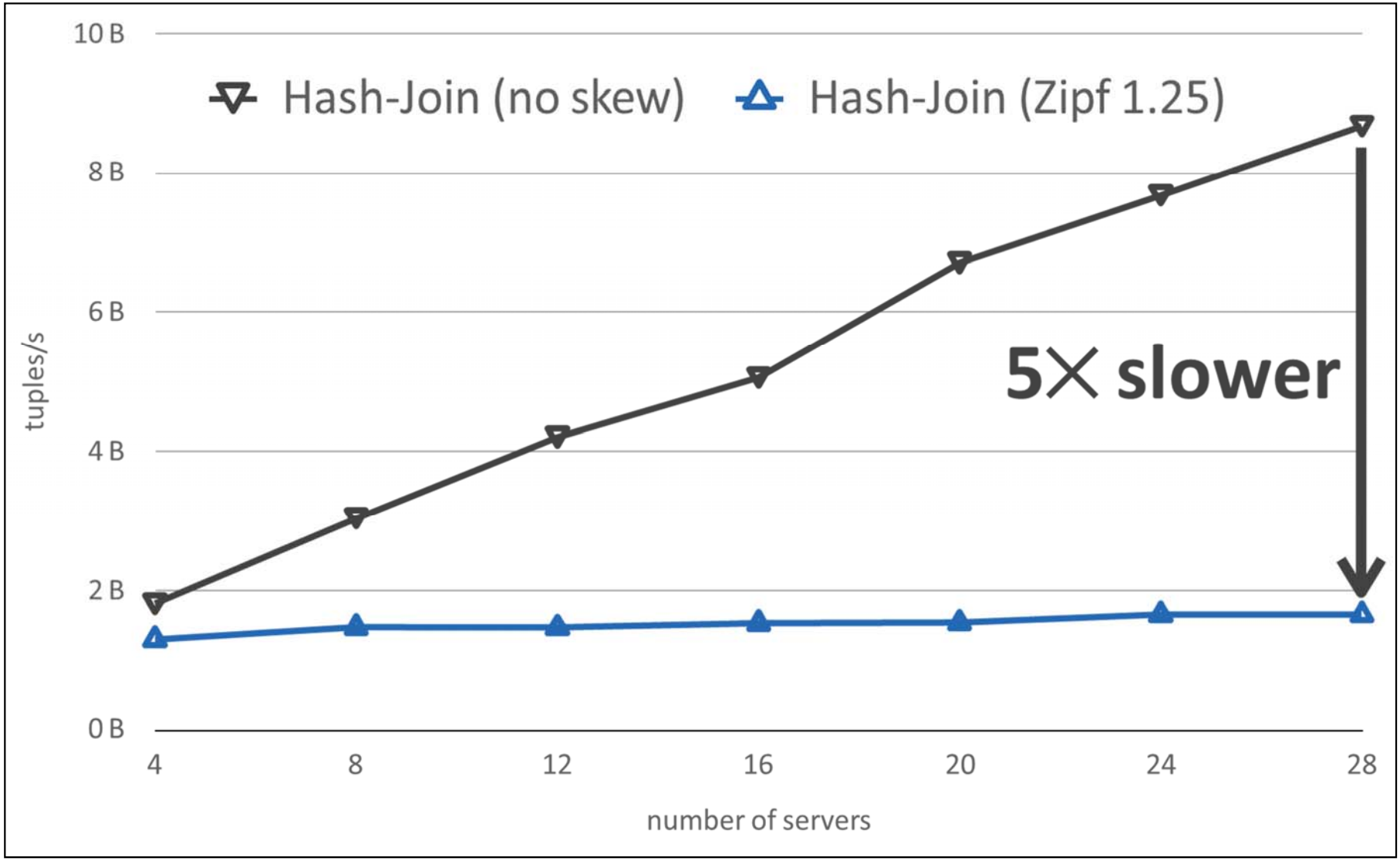
- Data Skew is a critical problem on distributed machines connected over a high-speed network.
- Performance determined by slowest server.

Problem with Skew

- Foreign key join with skewed probe relation
- Attribute value skew can cause severe imbalances
- One server receives more tuples than the others
- Join duration determined by slowest server

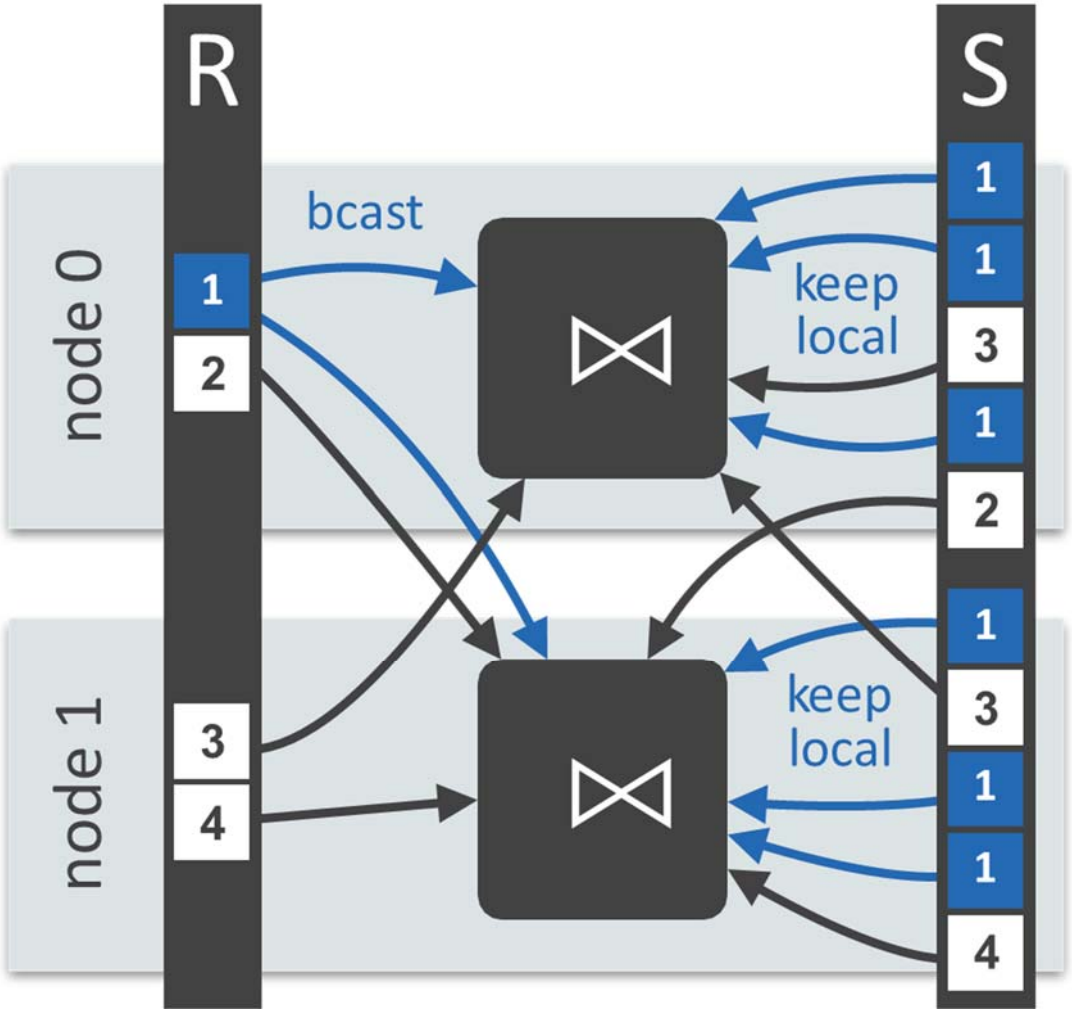


Skew Threatens Scalability



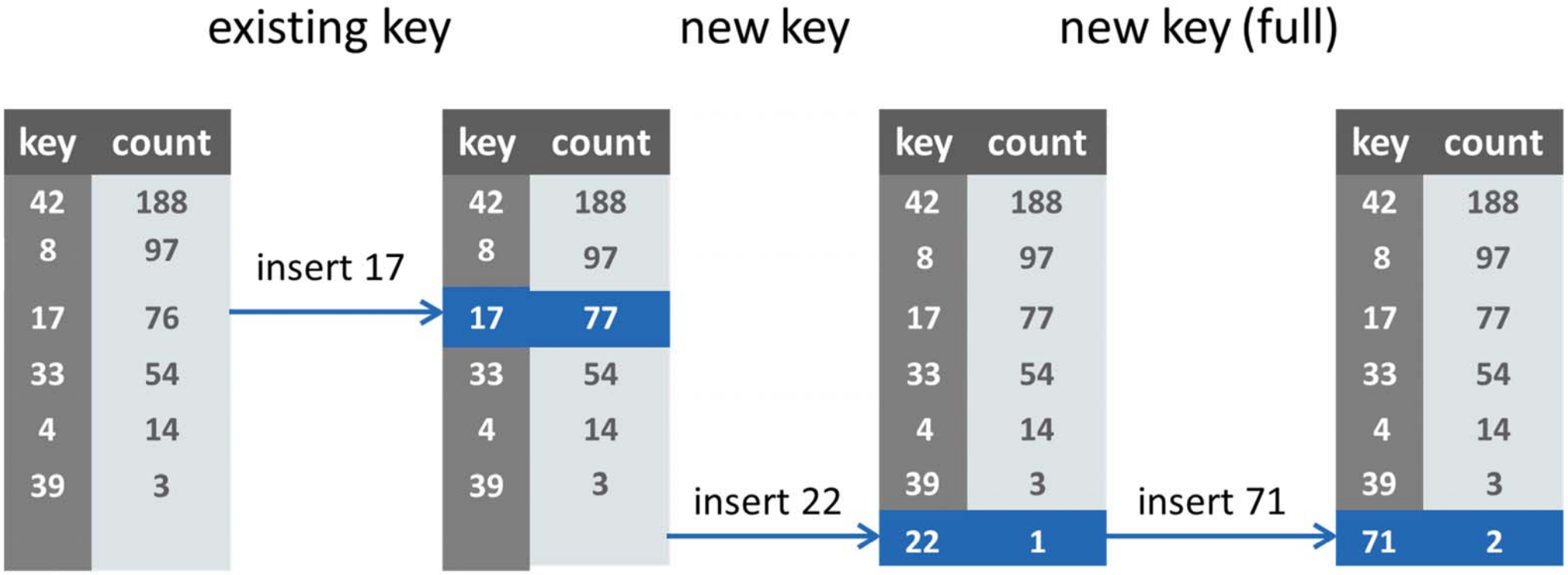
Ideal Solution: Selective Broadcast

- **Assumption:** Skewed values are known beforehand
- Broadcast the skewed build tuples
- Keep skewed probe local
- Avoids load imbalance
- Subset-Replicate [VLDB 92], Partial Redistribution & Duplication [SIGMOD 08]



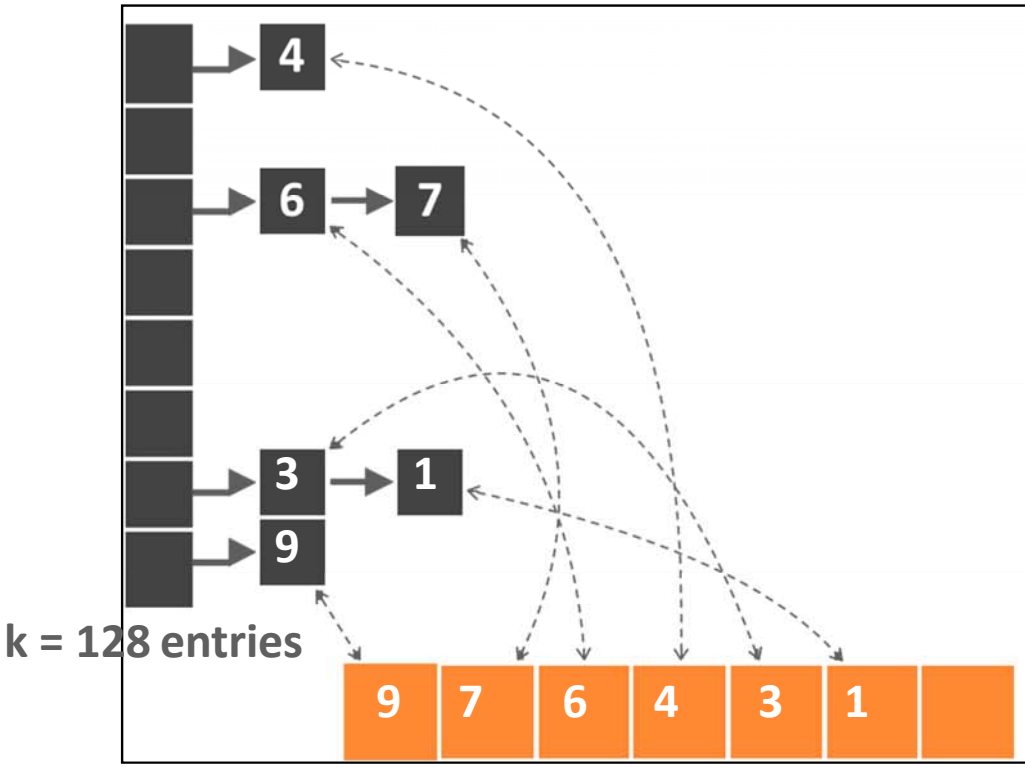
Dynamic Discovery of Heavy Hitters

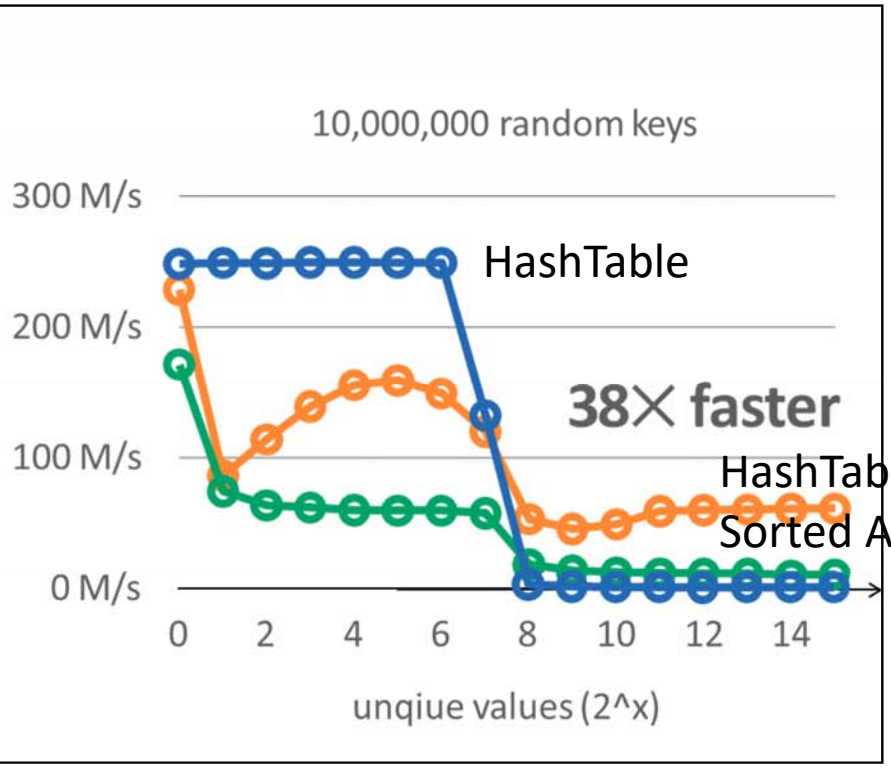
- Space Saving Algorithm (ICDT 05)



fast access by key vs. **replace minimum when full**

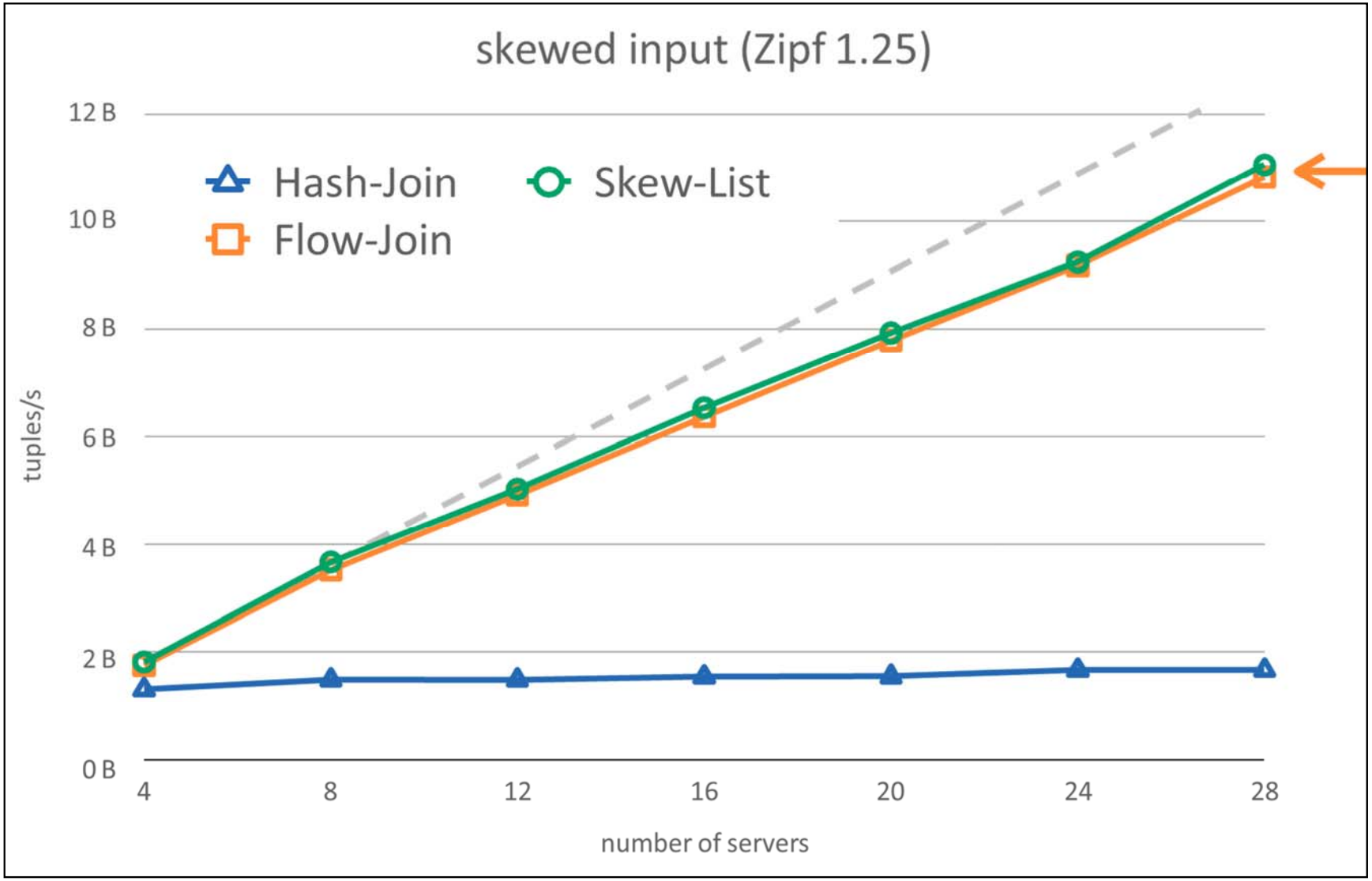
Data Structures: Hash Table + Sorted Array





- Sorted array for minimum
- Update only sometimes shifts an element to the left
- Increment key – worst case **O(k)**
- Remove minimum **O(1)**

Evaluation


skewed input (Zipf 1.25)
12 B
10 B
8 B
6 B
4 B
2 B
0 B
tuples/s
Hash-Join
Skew-List
Flow-Join
4
8
12
16
20
24
28
number of servers

Summary

- Designing a 1:1 mapping between logical operators and physical implementations is a viable proposition
- Tuning of these implementations is a function of schema information and not statistical estimates

- However, the operator-implementation mapping is only the first part of the problem, the sequencing of these operators still remains …

Stage 2: Robust Plans

APPROACHES

- Least Expected Cost (PODS 99 [11], PODS 02 [12])
  - estimate Distributions instead of Values for parameters
- Cost-Greedy (VLDB 2007 [17])
  - reduce parametric optimal set of plans (POSP) space into low-cardinality ("anorexic") approximation featuring relatively stable plans
- SEER (VLDB 2008 [18])
  - reduce POSP space into anorexic approximation that can handle arbitrary estimation errors

LEC Basic Principle

- Typically optimize for either the estimated values of execution parameters (data/query/system), or for "magic constants" assigned to them.

- An alternative approach is to assume that an approximate probability distribution of the parameter values is known (e.g. through histograms or learnt models)

Least Expected Cost (LEC) Plan

- This allows, in principle, identification of plan with least expected cost over the parameter space.

- But, significantly increases optimization overheads, especially with large number of POSP plans
  - Addressable by "anorexic reduction" (VLDB 2007 [17])
  - Significant uncertainty in the distributions
  - Restricted to left-deep plans
  - Assumes LEC plan = some plan from POSP space

LEC = LSC plan

- Standard optimizer – optimizes for the expected value of the parameter distribution (i.e. the Least Specific Cost (LSC) plan)
- When is LSC = LEC ?
  - If the cost of a plan is linear in its input parameters
  - If the cost of a plan is a sum of products of independent parameters
- Guarantee:
  - For an n-way left-deep join, the LSC plan is within a factor of n of the LEC plan (if plan cost includes variance)

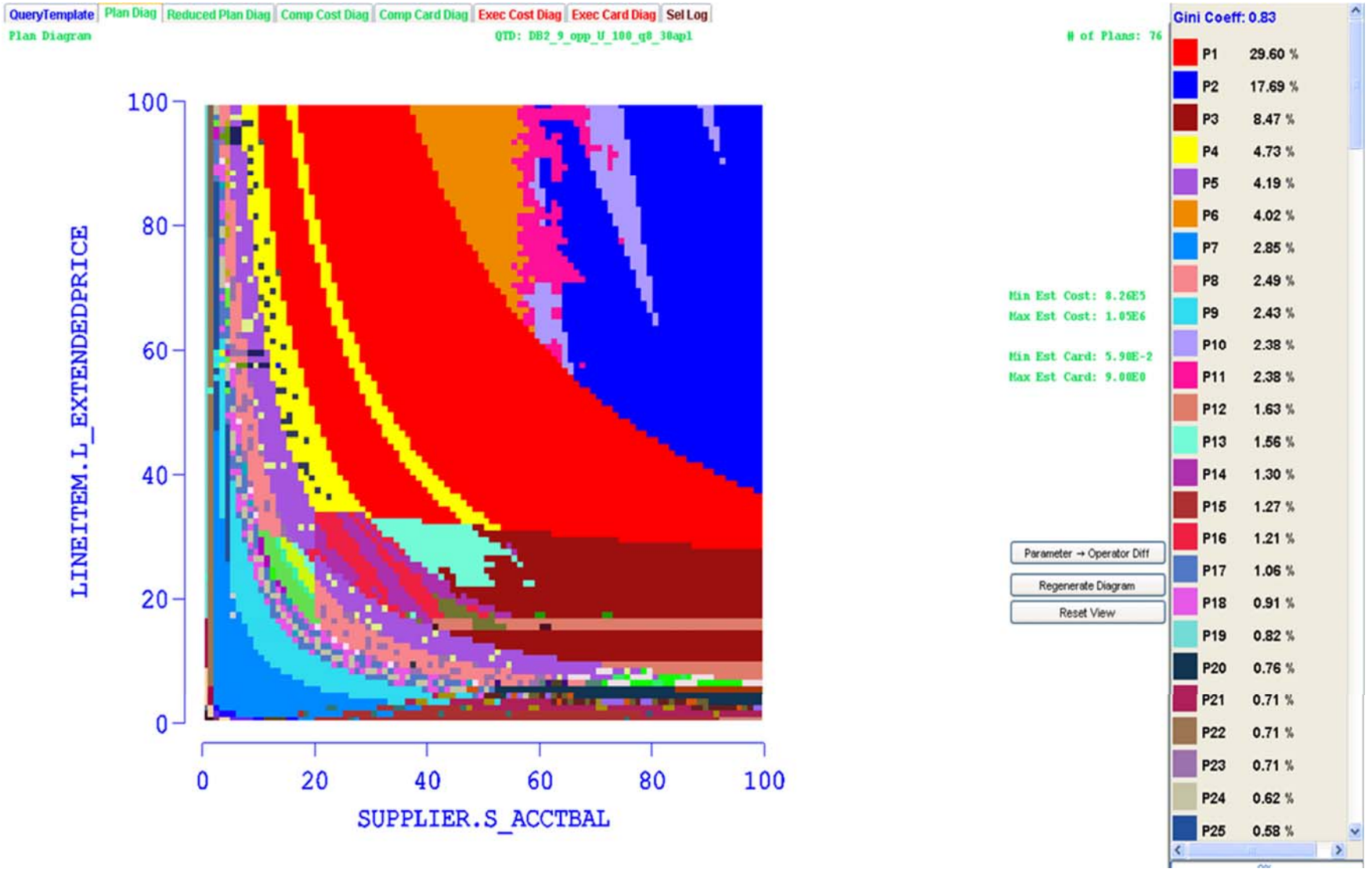
Cost-Greedy

Query Template [Q8 of TPC-H]

Determines how the market share of Brazil within America has changed over 1995-1996 for Steel parts

```
select o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume)
from   ( select   YEAR(o_orderdate) as o_year,
                  l_extendedprice * (1 - l_discount) as volume, n2.n_name as nation
         from     part, supplier, lineitem, orders, customer, nation n1, nation n2, region
         where    p_partkey = l_partkey and s_suppkey = l_suppkey and
                  l_orderkey = o_orderkey and o_custkey = c_custkey and
                  c_nationkey = n1.n_nationkey and n1.n_regionkey = r_regionkey  and
                   r_name = 'AMERICA' and s_nationkey = n2.n_nationkey and
                  o_orderdate between '1995-01-01' and '1996-12-31' and
                  p_type = 'ECONOMY ANODIZED STEEL'
                  and s_acctbal ≤ C1 and l_extendedprice ≤ C2
       ) as all_nations
group by o_year
order by o_year
```

POSP Plan Diagram



Problem Statement

Can the plan diagram be recolored with a smaller set of colors (i.e. some plans are "swallowed" by others), such that

Guarantee:

No query point in the original diagram has its estimated cost increased, post-swallowing, by more than λ percent (user-defined)

Analogy: (with due apologies to Macanese in the audience)
Macau agrees to be annexed by China if it is assured that the cost of living of each Macanese citizen is not increased by more than λ percent

CostGreedy

- Optimal plan diagram reduction (w.r.t. minimizing the number of plans/colors) is NP-hard
  - through problem-reduction from classical Set Cover

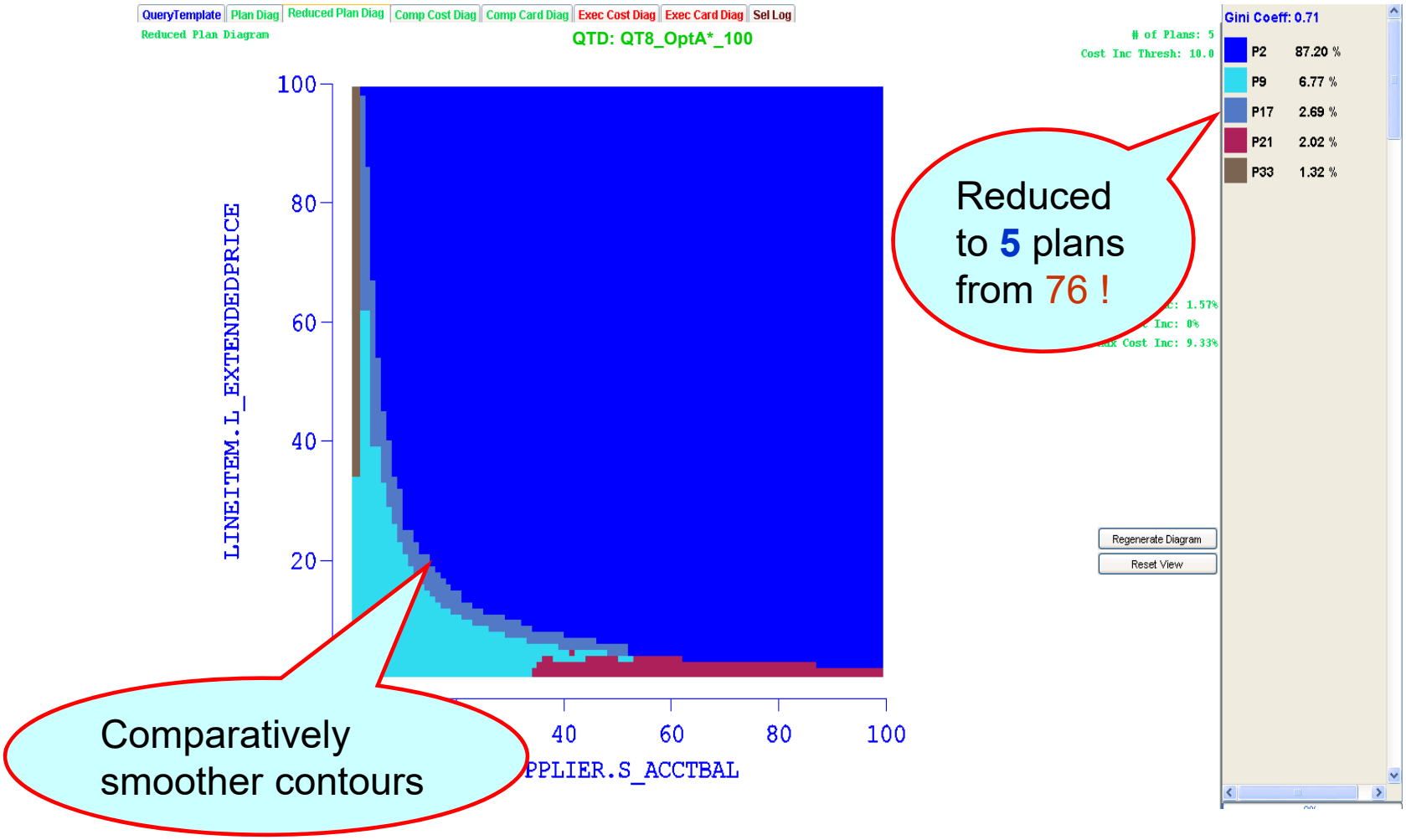
- CostGreedy is a greedy heuristic-based algorithm with following properties:
  [m is number of query points, n is number of plans in diagram]

  - Time complexity is O(mn)
    - linear in number of plans for a given diagram resolution
  - Approximation Factor is O(ln m)
    - bound is both tight and optimal
    - in practice, closely approximates optimal

Reduced Plan Diagram [λ=10%]

Complex Plan Diagram

[QT8, OptA*, Res=100]



Applications of Plan Diagram Reduction

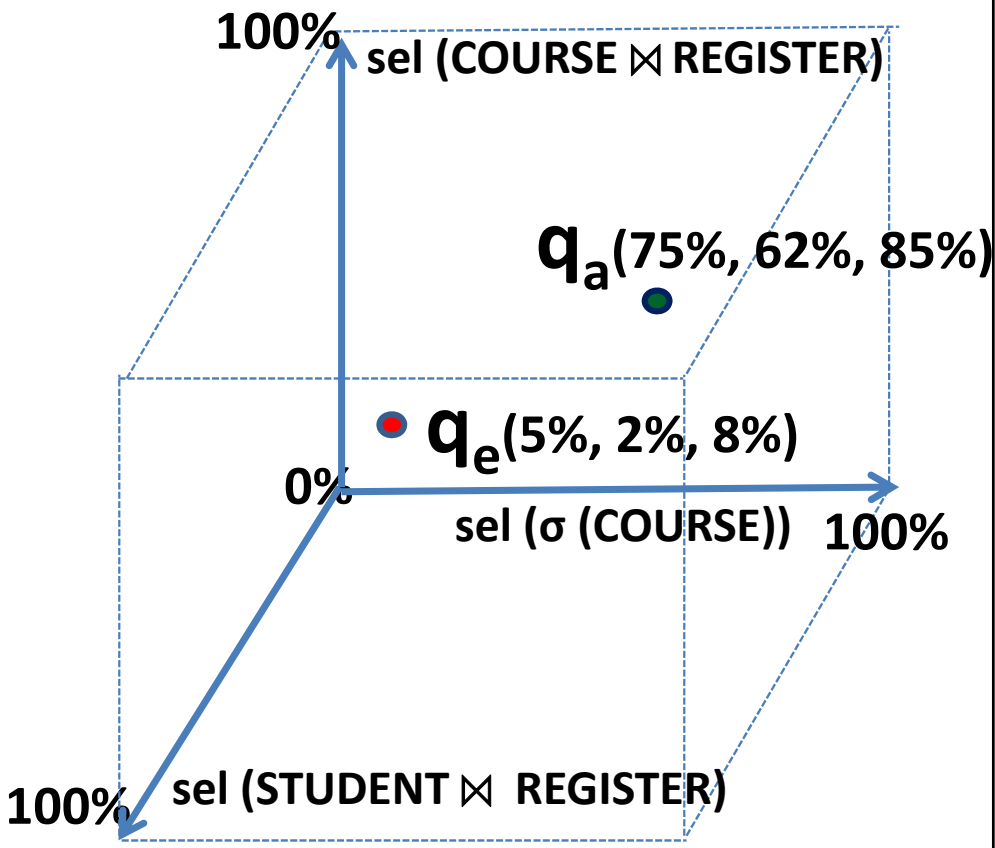
- Quantifies redundancy in plan search space
- Provides better candidates for plan-cacheing
- Enhances viability of Parametric Query Optimization (PQO) techniques
- Improves efficiency/quality of LEC plans
- Identifies selectivity-error resistant plan choices
  - retained plans are robust choices over larger selectivity parameter space
- Minimizes overheads of multi-plan approaches (e.g. Adaptive Query Processing)

Limitation

Cost Greedy can cause arbitrarily poor performance if the selectivity error is large enough that the actual location of the query falls outside the swallowing region of the estimated location.

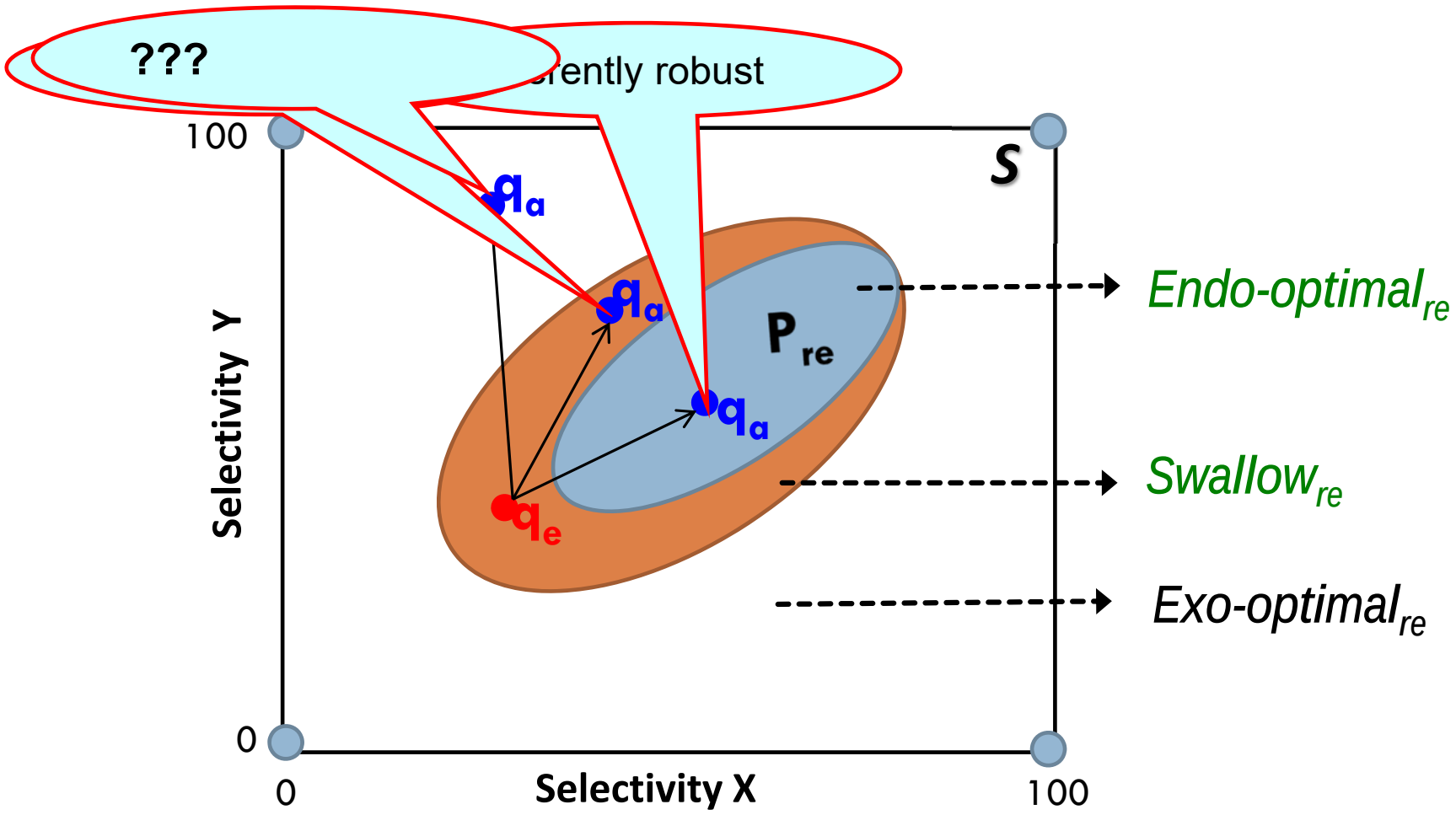
Notation

```
select  *
from    STUDENT, COURSE, REGISTER
where   S.RollNo = R.RollNo and
        C.CourseNo = R.CourseNo and
        C.fees < 1000
```



- q_e – estimated selectivity location in SS (Selectivity Space)
- q_a – actual run-time location in SS
- P_{oe} – optimal plan for q_e
- P_{oa} – optimal plan for q_a
- P_{re} – replacement plan for q_e

Error Locations wrt Plan Replacement Regions

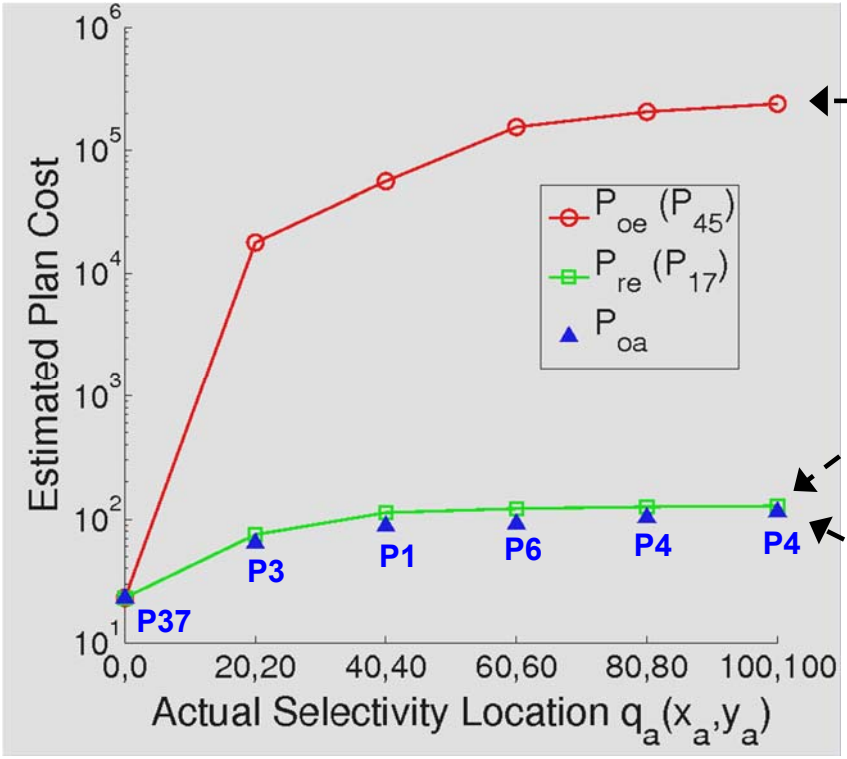


Positive Impact of Reduction

In most cases, replacement plan provides robustness to selectivity errors even in exo-optimal region

QT5

q_e = (0.36, 0.05)



Original Plan (P_{oe})

Replacement Plan (P_{re})

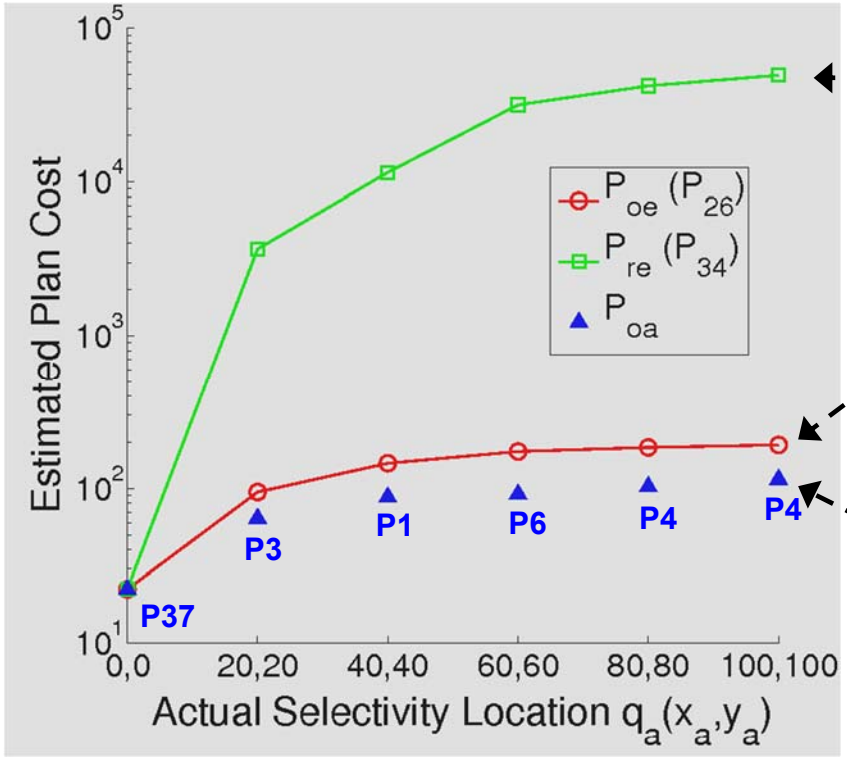
Local Optimal Plan (P_{oa})

Negative Impact of Reduction

But, occasionally, the replacement is much worse than the original plan !

QT5

q_e = (0.03, 0.14)



Replacement Plan (P_{re})

Original Plan (P_{oe})

Local Optimal Plan (P_{oa})

SEER

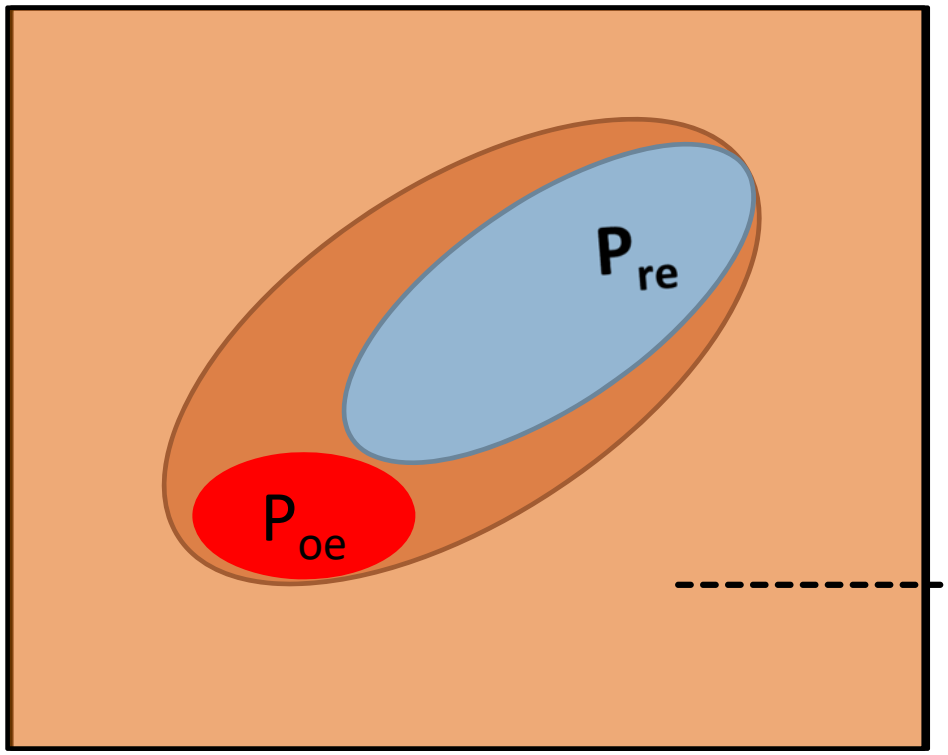
[Selectivity Estimate Error Resistance]

Globally Safe Replacement

- Earlier local constraint:
  P_{re} can replace P_{oe} if
  - ∀ points q in P_{oe}'s endo-optimality region,
    $\text{cost}(P_{re}, q) \leq (1+\lambda)\ \text{cost}(P_{oe}, q)$

- New global constraint:
  P_{re} can replace P_{oe} only if it guarantees a globally safe space
  - ∀ points q in selectivity space **S**,
    $\text{cost}(P_{re}, q) \leq (1+\lambda)\ \text{cost}(P_{oe}, q)$

Globally Safe Replacement



Safe (P_{re}, P_{oe})

Analogy Update

China can annex Macau only if the Chinese passport can guarantee cost of living of Macanese citizen is within λ of that obtained with the Chinese passport, *irrespective of the country to which the Macanese citizen emigrates*.

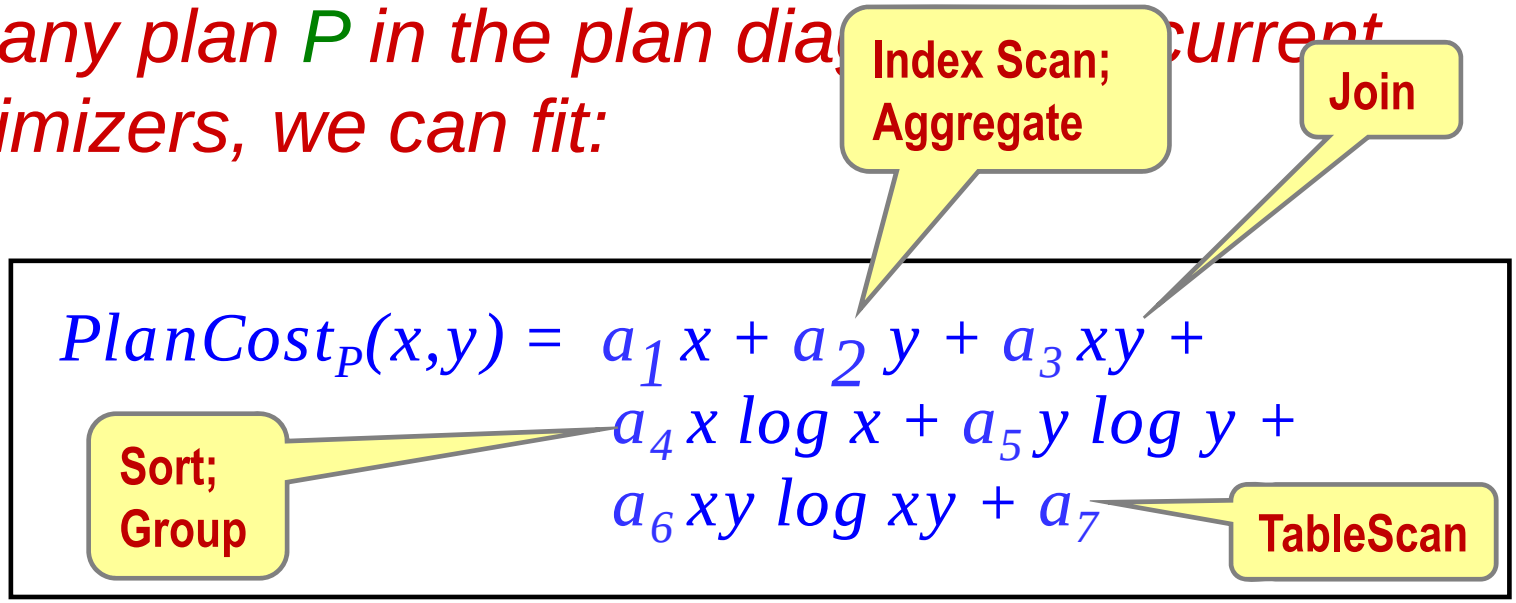
Solution Strategy

- Characterize behavior of all plans throughout the selectivity space **SS**
- Accomplished using the *Abstract Plan Costing (APC)* feature of current DBMS
  - Supports costing of plans in their exo-optimal regions
- But, not a viable solution in practice
  - Requires $O(mn)$ APC to be performed (exp in SS dimensionality)
    - m: number of query points; n: number of optimal plans
  - Although costing cheaper than optimization (1:100), the sheer number makes it prohibitively expensive

Can we reduce the number of APC invocations to a manageable extent?

Plan Cost Model (2D)

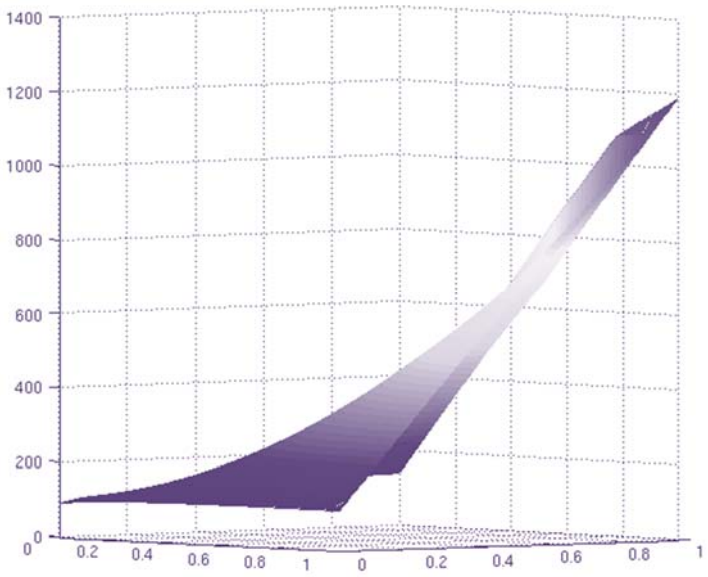
Given selectivity variations x and y, for any plan P in the plan dia current optimizers, we can fit:



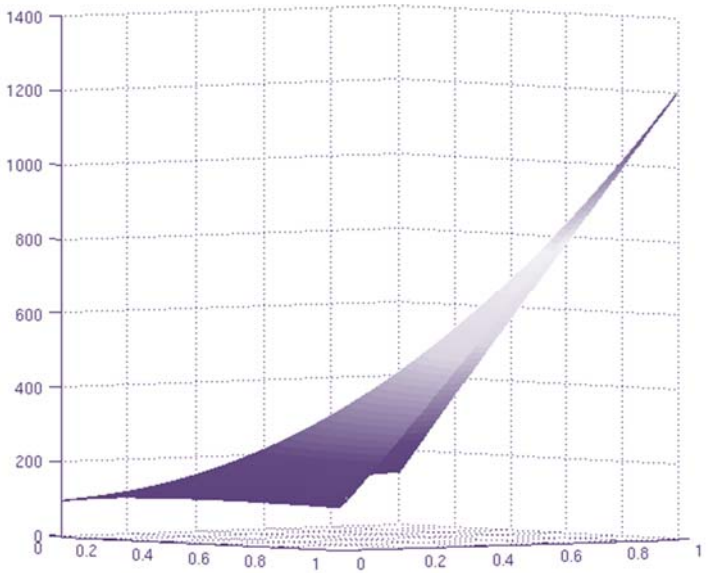
The specific values of a_1 through a_7 are a function of P .

Extension to n-dimensions is straightforward.

Cost Model Fit Example



Original Cost Function



Fitted Cost Function

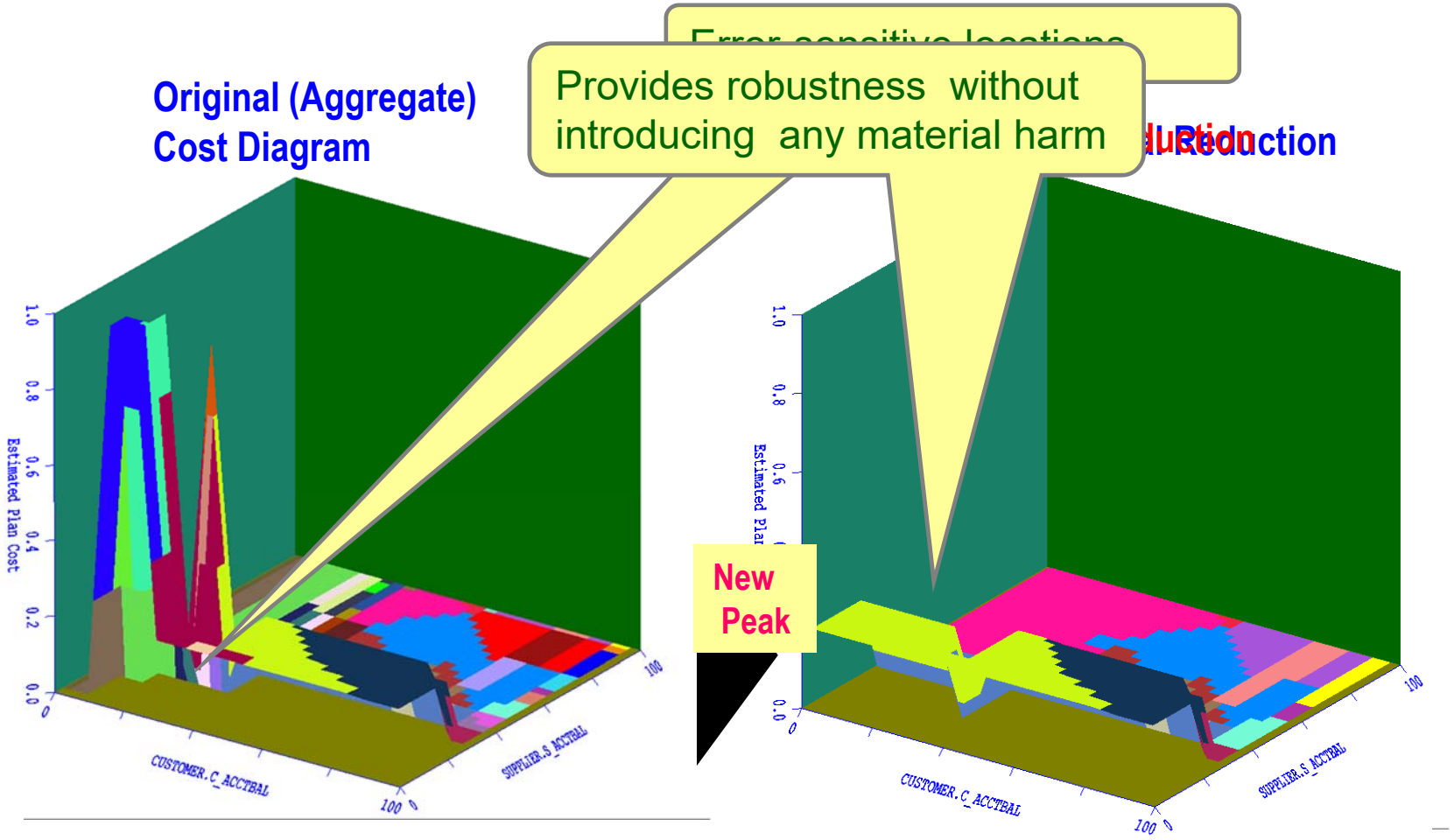
$$Cost(x,y) = 17.9x + 45.9y + 1046xy - 39.5x\log x + 4.5y\log y + 27.6xy\log xy + 97.3$$

Main Result

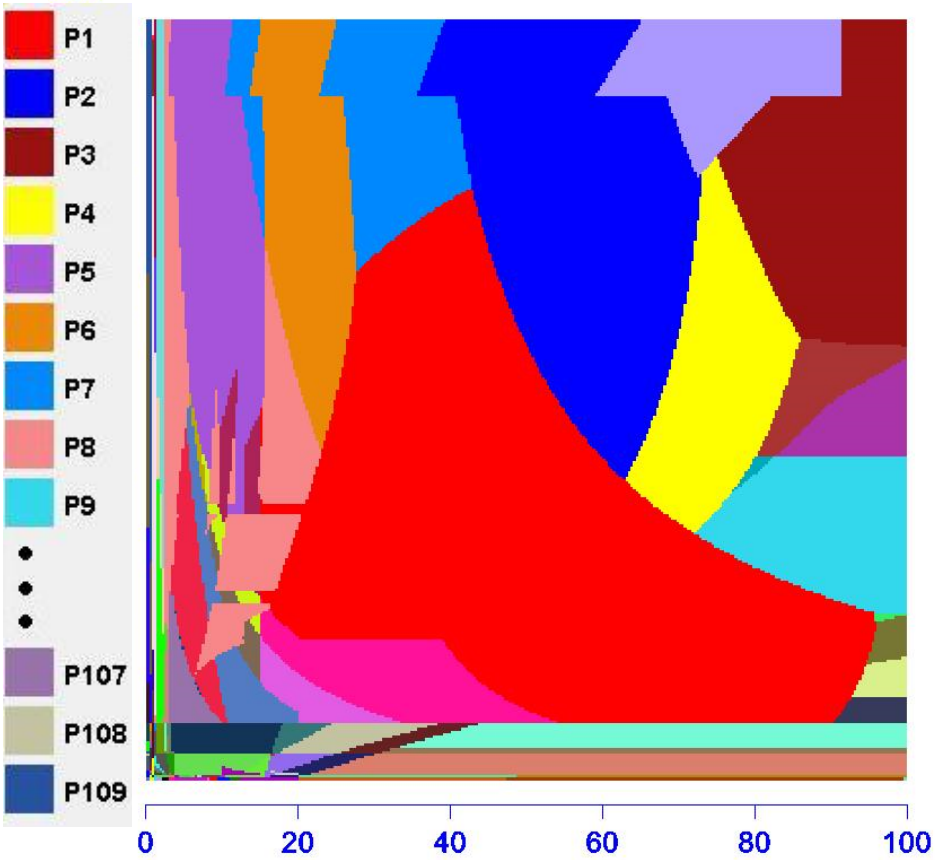
Given the 7-coefficient plan cost model, need to perform APC at only the perimeter of the selectivity space to determine global safety

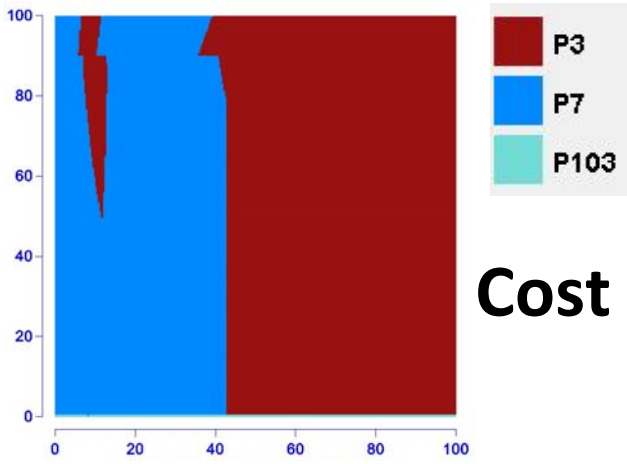
i.e. Border Safety ⇒ Interior Safety !

Error Resistance Example

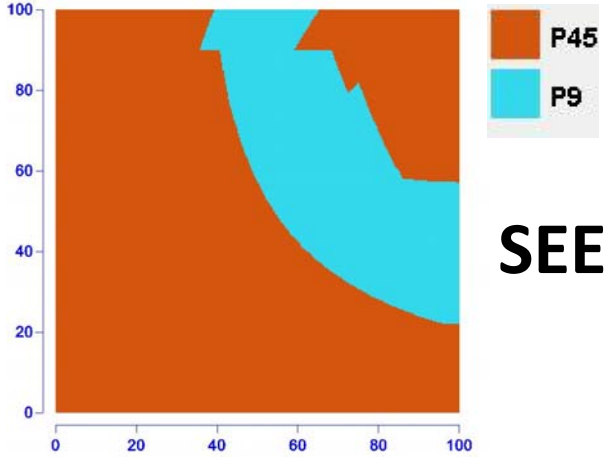


Anorexic Reduction Remains!





Cost Greedy



SEER

Limitation

- Although SEER introduces stability into the plan choices, its performance guarantees are with respect to $\mathbf{P_{oe}}$, the optimal plan at the estimated location.

- Ideally, we would like performance guarantees to be with respect to $\mathbf{P_{oa}}$, the optimal plan at the actual location (this is the "God's plan").

Stage 3: Robust Execution

APPROACHES

- Bounded Impact (PVLDB 2009 [33])
  - performance guarantee with quartic dependency on error magnitude
- Plan Bouquet (SIGMOD 14 / TODS 16 [14])
  - discovery-based approach to selectivities
  - error-independent guarantees with linear dependency on plan diagram density
- Spill-Bound (ICDE 16 / TKDE 19 [22])
  - platform-independent guarantee with quadratic dependency on error dimensionality
- Frugal Spill-Bound (PVLDB 2018 [23])
  - extension to ad-hoc queries

Measuring Cardinality Estimation Errors

Popular error metrics (= optimization goals)

$$l_2 = \sqrt{(f_e(x) - f_a(x))^2}$$

$$l_\infty = \max |(f_e(x) - f_a(x)|$$

Minimizing these error metrics can lead to **arbitrarily bad plans**!

Q(uotient) Error

- Errors propagate multiplicatively, so metric should also be multiplicative
- It should be symmetric wrt over- and under-estimation

q-error is defined as:

$$l_q = \max \frac{max\ (f_e(x), f_a\ (x))}{min\ \ (f_e(x), f_a\ (x))}$$

- actual cardinality 10, estimation 100 $\Rightarrow$ l_q = 10
- actual cardinality 10, estimation 1 $\Rightarrow$ l_q = 10
- note that l_q is the maximum over the whole domain of x

Knowing *q*-error provides **bounds on resulting plan performance**!

Cost Bounds Implied by Q-error

- **Theorem:**

  Let joins be all Sort-Merge joins or all Grace-Hash joins. Then

$$Cost(P_{oe}, q_a) \leq q^4 \, Cost(P_{oa}, q_a)$$

  where q is the maximum q-error taken over all intermediate results.

  Problem: q can be arbitrarily large!

Plan Bouquet

Approach

- Plan Bouquet is a new query processing technique, that completely abandons estimating operator selectivities
- Instead, run-time selectivity discovery using compile-time selected bouquet of plans
  - provides worst case performance guarantees wrt ideal that magically knows the correct selectivities
    e.g. for single error-prone selectivity, relative guarantee of 4
  - empirical performance well within guaranteed bounds on industrial-strength environments

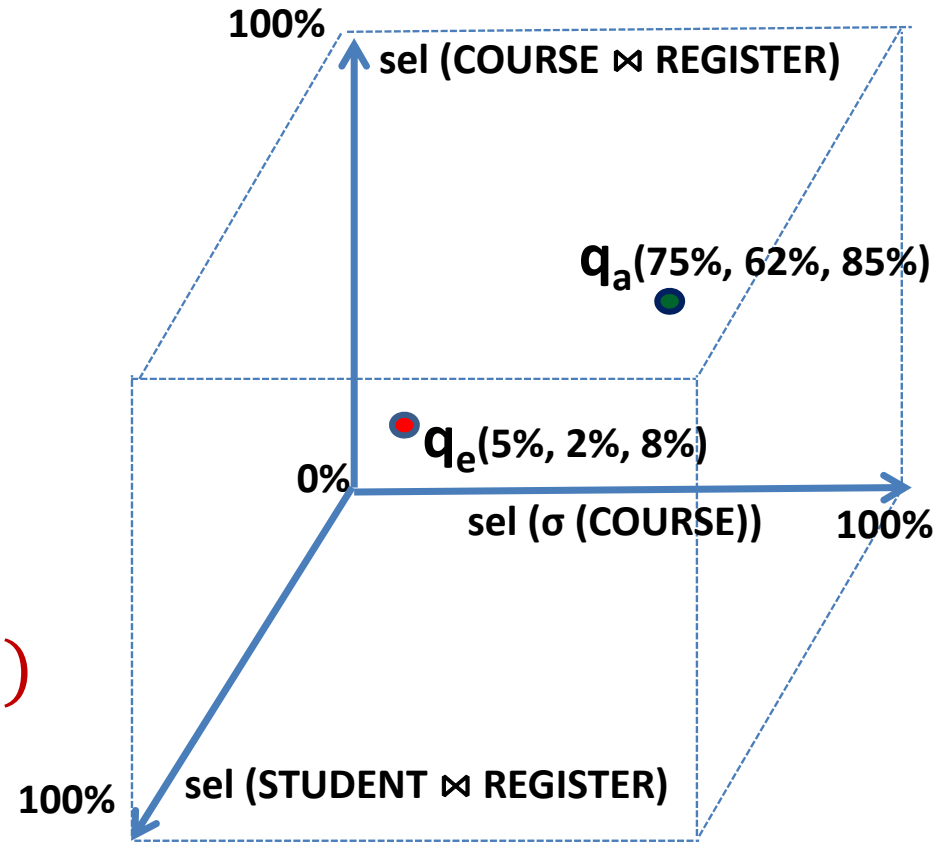
Problem Framework

Performance Metrics

- q_e – estimated selectivity location in SS
- q_a – actual run-time location in SS

- P_{oe} – optimal plan for q_e
- P_{oa} – optimal plan for q_a

$$SubOpt(q_e, q_a) = \frac{cost(P_{oe}, q_a)}{cost(P_{oa}, q_a)} \quad [1, \infty)$$



$$MaxSubOpt\ (MSO) = MAX[SubOpt(q_e, q_a)] \quad \forall q_e, q_a \in SS$$

Note: Metric is now with respect to the "God's plan"

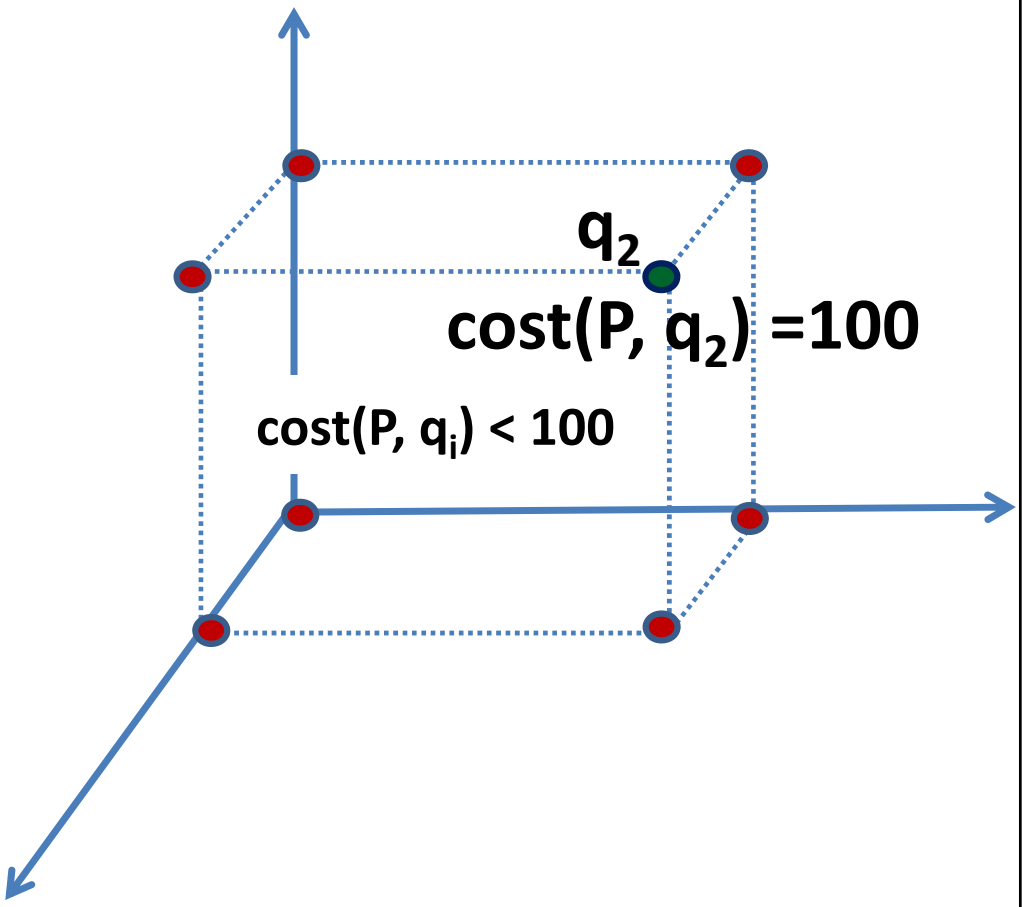
Main Assumption

- Plan Cost Monotonicity

For any plan P and distinct locations q_1 and q_2

$$\text{Cost}(P, q_1) < \text{Cost}(P, q_2) \quad \text{if } q_1 \prec q_2$$

(i.e. q_1 is dominated by q_2)

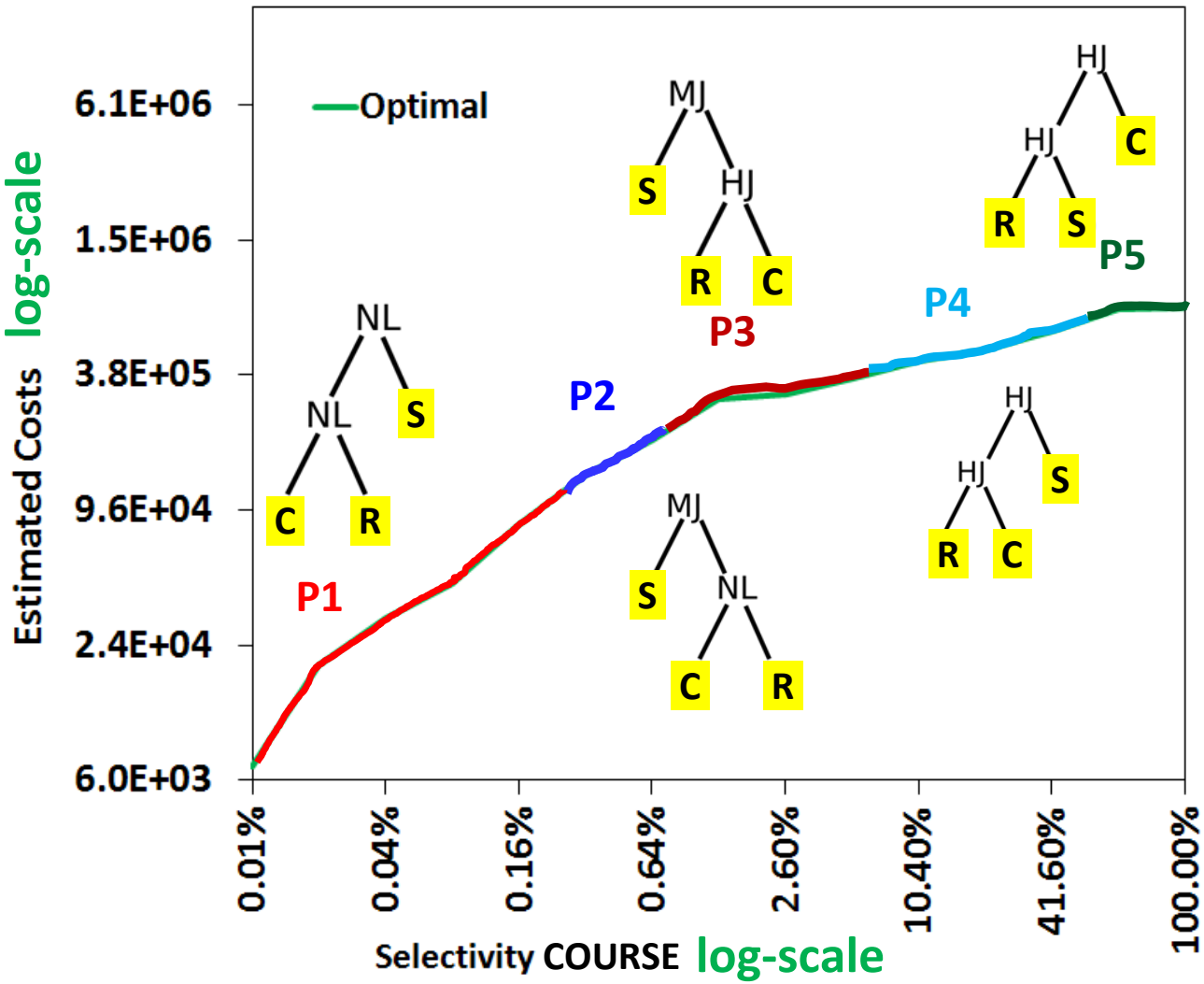


Contemporary Optimizer Behavior on 1D Selectivity Space

Parametric Optimal Set of Plans (POSP)

(Parametric version of Example Query)

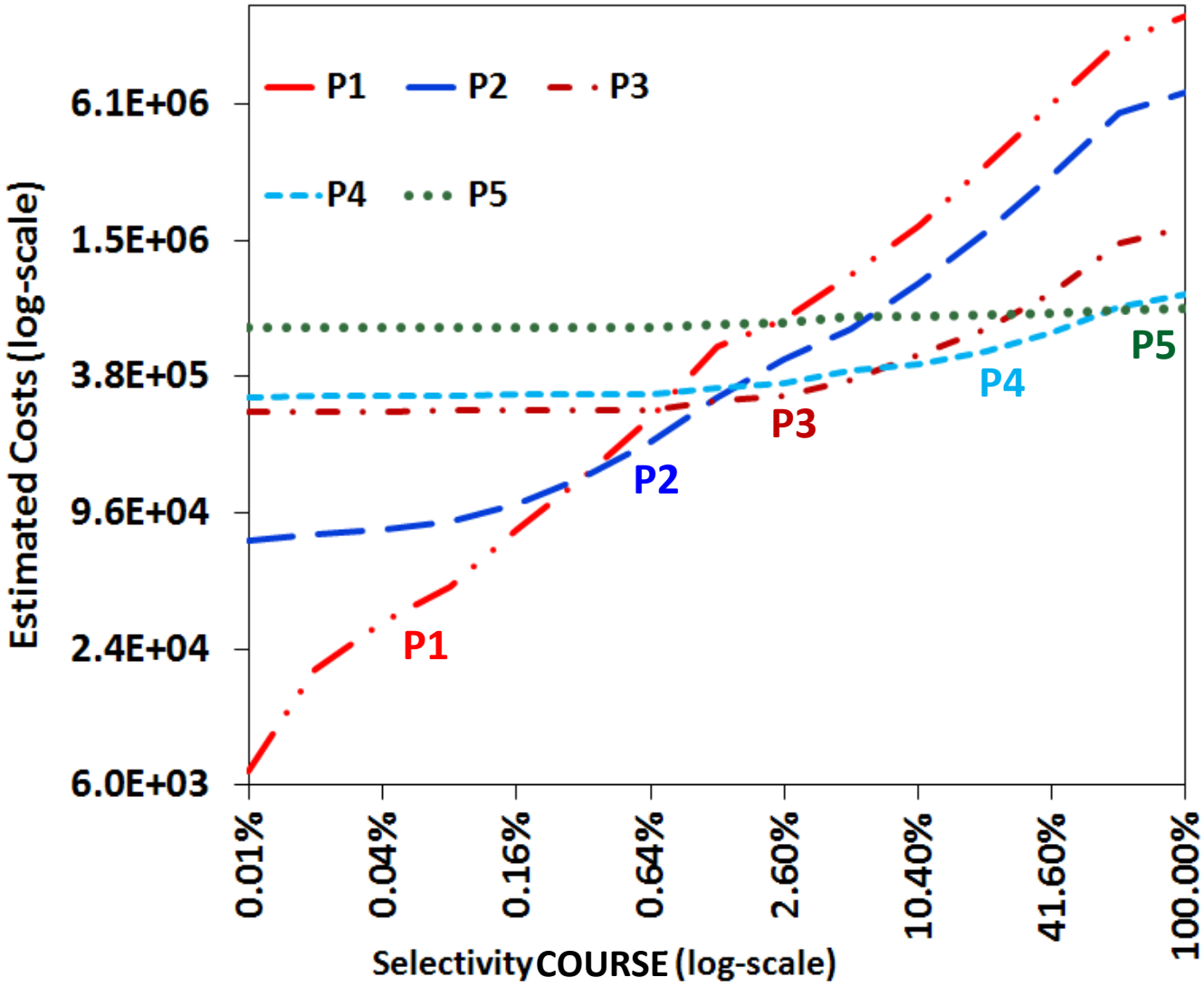
```
select  *
from    STUDENT, COURSE, REGISTER
where   S.RollNo = R.RollNo and
        C.CourseNo = R.CourseNo and
        C.credits < $1
```



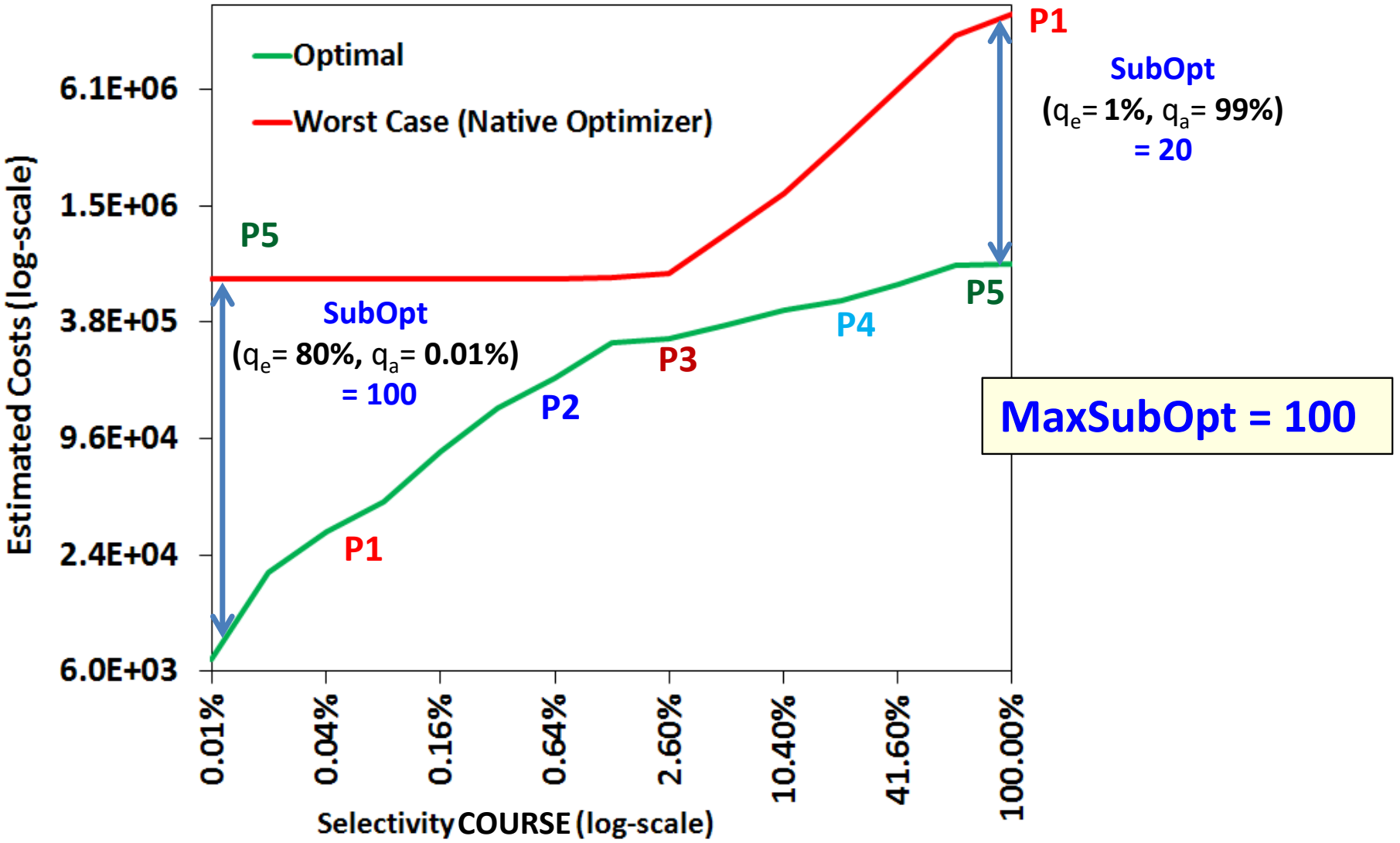
S: Student
C: Course
R: Register

NL: Nested Loop Join
MJ: Merge Join
HJ: Hash Join

POSP Performance Profile (across SS)

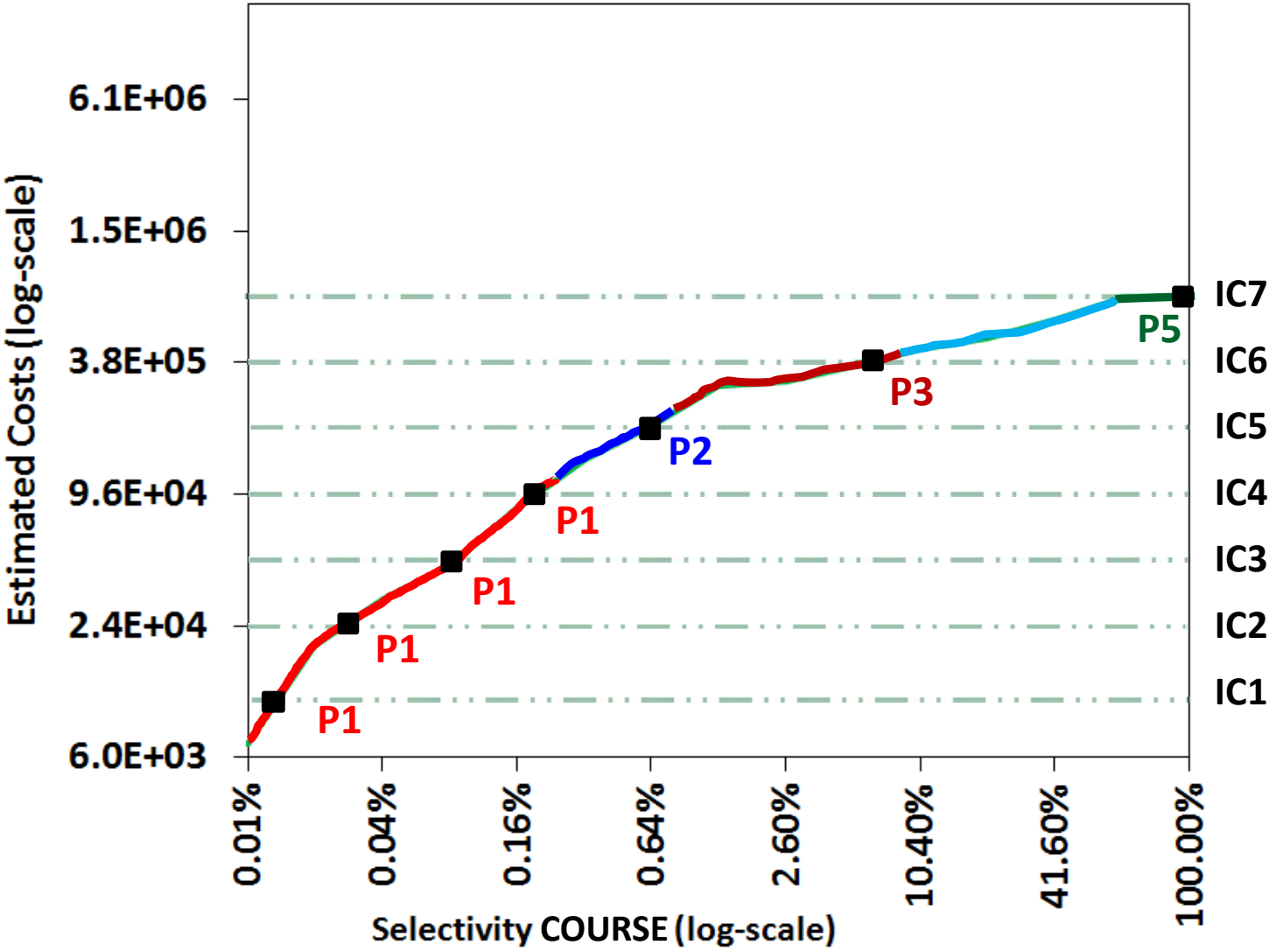


Sub-optimality Profile (across SS)



Plan Bouquet Behavior on 1D Selectivity Space

Bouquet Identification

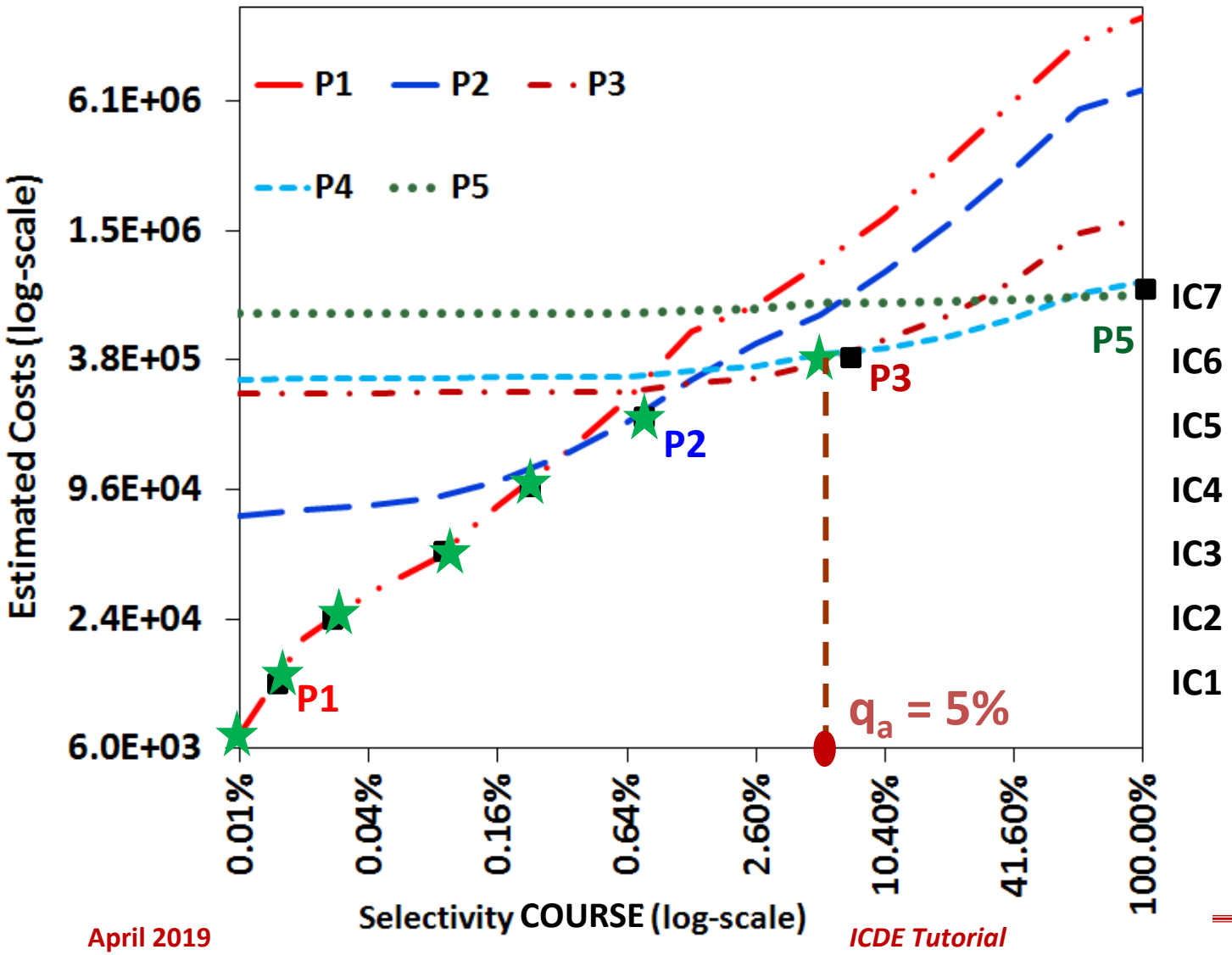


Step 1: Draw cost steps with cost-ratio **r=2** (geometric progression).

Step 2: Find plans at intersection of optimal profile with cost steps

Bouquet = {P1, P2, P3, P5}

Bouquet Execution



Let q_a = 5%

(1) Execute P1
with budget IC1(1.2E4)
Throw away results of P1

(2) Execute P1
with budget IC2(2.4E4)
Throw away results of P1

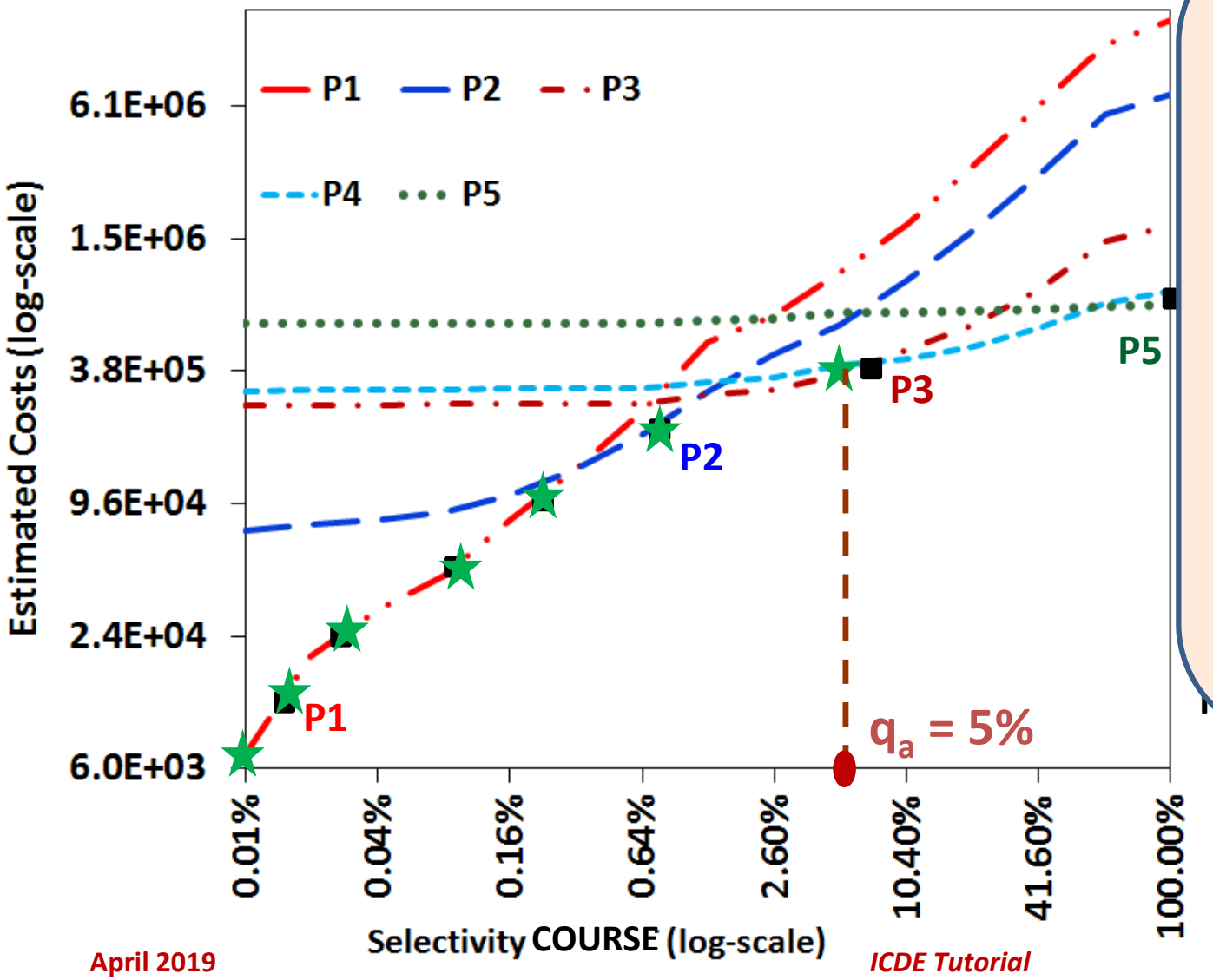
(3) Execute P1
with budget IC3(4.8E4)
Throw away results of P1

(4) Execute P1
with budget IC2(9.6E4)
Throw away results of P1

(5) Execute P2
with budget IC5(1.9E5)
Throw away results of P2

(6) Execute P3
with budget IC6(3.8E5)
P3 completes with cost 3.4E5

Bouquet Execution



Let q_a = 5%

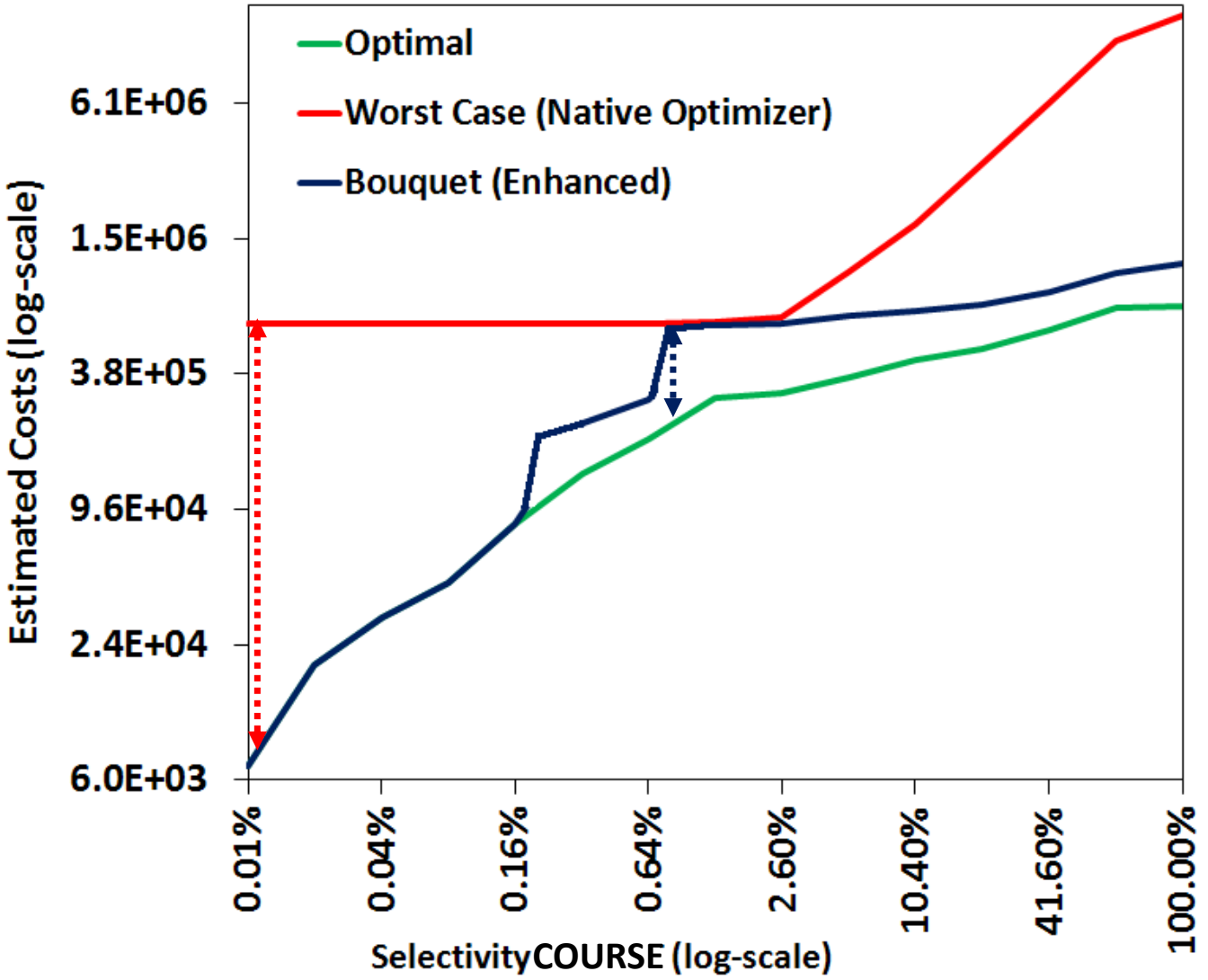
Bouquet Cost = 3.4 E5 (P3) +
1.92 E5 (P2) +
0.96 E5 (P1) +
0.48 E5 (P1) +
0.24 E5 (P1) +
0.12 E5 (P1)
= 7.1 E5

SubOpt (*, 5%) = 7.1/3.4 = 2.1

With obvious optimization
SubOpt(*, 5%) = 6.3/3.4 = 1.8

with budget IC6(3.8E5)
P3 completes with cost 3.4E5

Bouquet Performance (EQ)



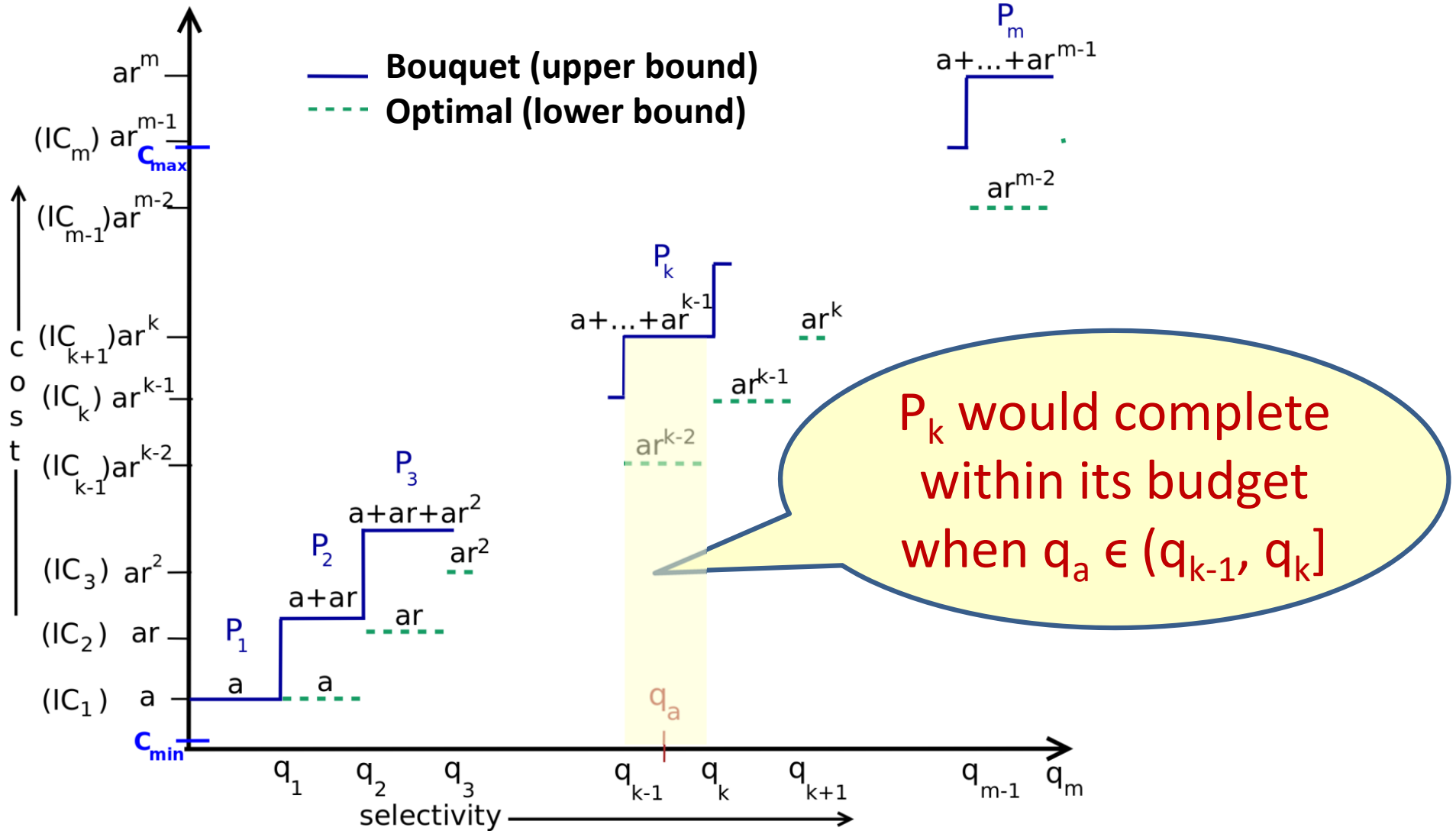
Native Optimizer

MaxSubOpt = 100

Bouquet

MaxSubOpt = 3.1

Worst Case Cost Analysis



1D Performance Bound

$$\mathbf{C_{bouquet}(q_{k-1}, q_k] = cost(IC_1) + cost(IC_2) + \ldots + cost(IC_{k-1}) + cost(IC_k)}$$

$$= \quad a \quad + \quad ar \quad + \ldots + \quad ar^{k-2} \quad + \quad ar^{k-1}$$

$$= \quad \frac{a(r^k - 1)}{(r-1)}$$

$$\mathbf{C_{optimal}(q_{k-1}, q_k] \geq ar^{k-2}} \qquad \text{(Implication of PCM)}$$

$$\mathbf{SubOpt_{bouquet}(*, q_a) \leq \frac{1}{ar^{k-2}} \times \frac{a(r^k - 1)}{(r-1)} \leq \frac{r^2}{r-1}} \qquad \forall q_a \in (q_{k-1}, q_k]$$

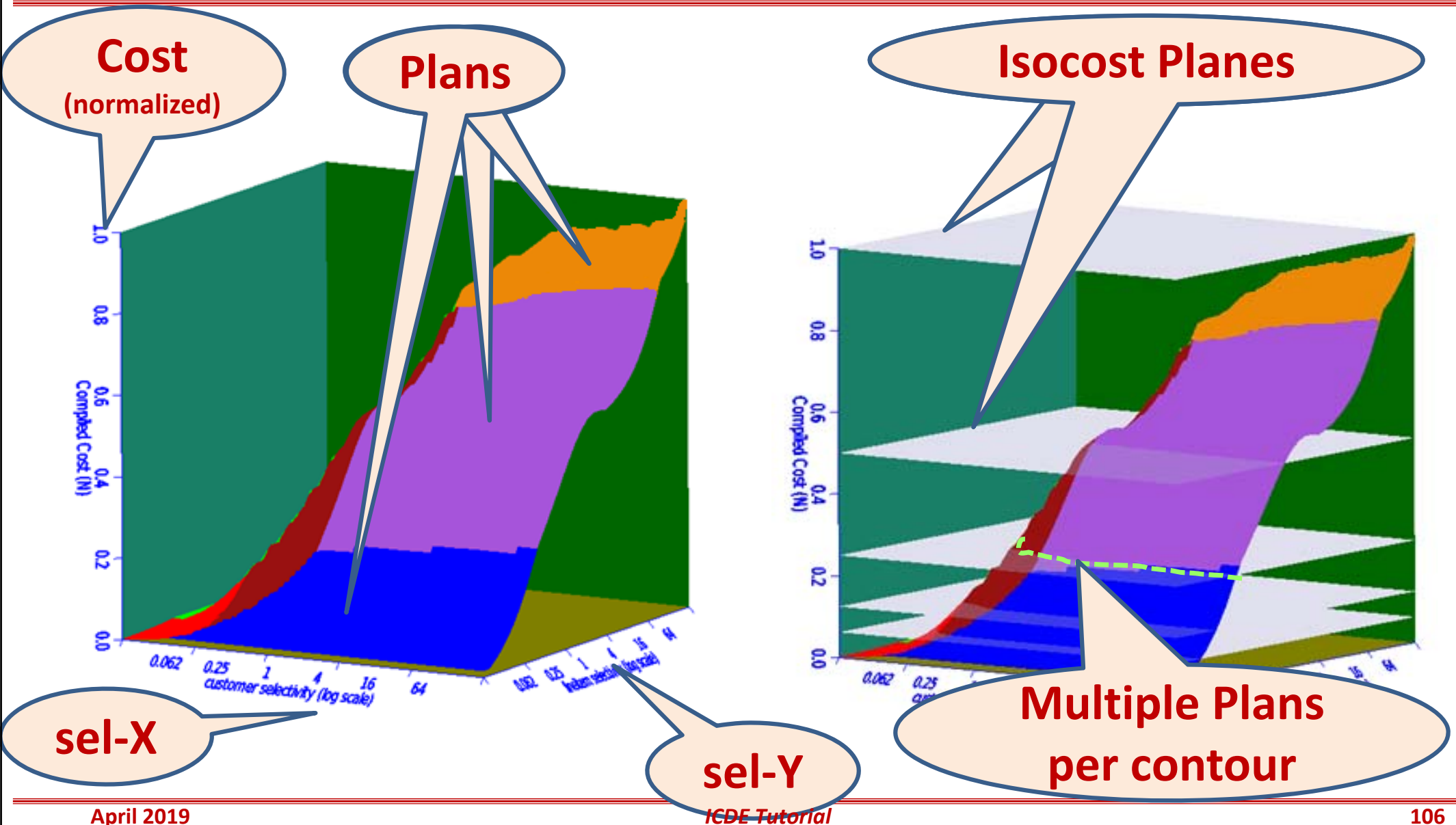
Reaches minima at r = 2

➔ MSO = 4

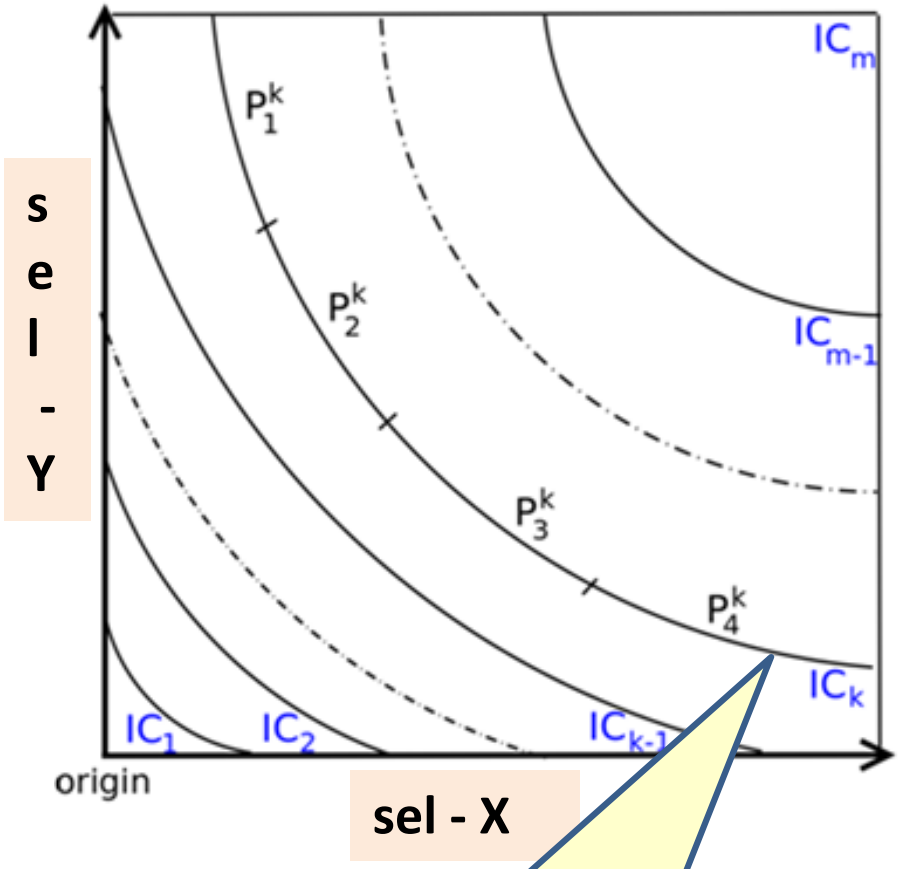
Best performance achievable by <u>any</u> deterministic online algorithm!

Bouquet Approach in 2D SS

2D Bouquet Identification



Characteristics of 2D Contours



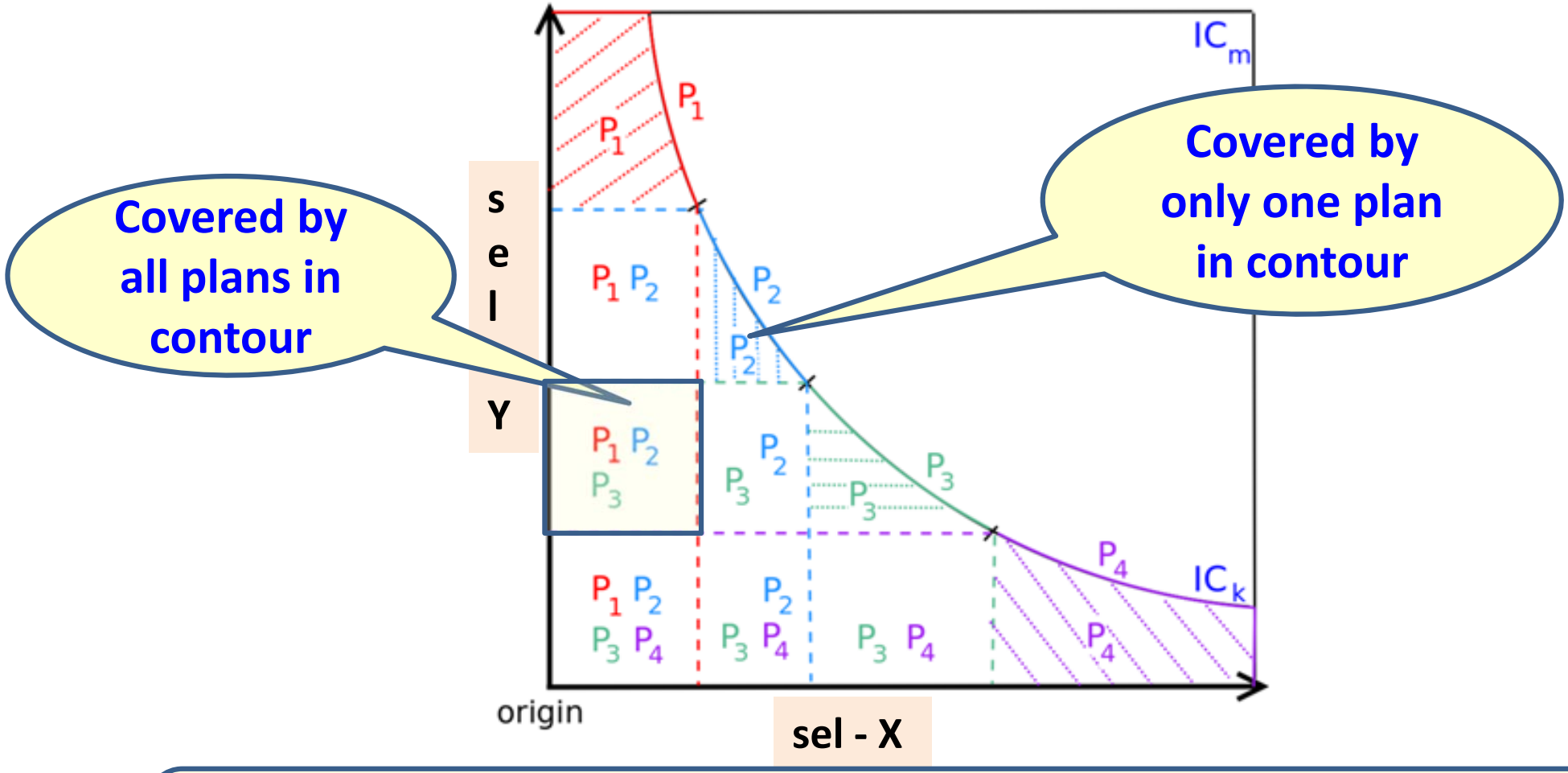
2D contours

- **Hyperbolic curves**
- **Multiple plans per contour**

Third quadrant coverage (due to PCM)

P_2^k can complete any query with actual selectivitiy (q_a) in the shaded region within cost(IC_k)

Crossing 2D Contours



⇒ **Entire set of contour plans must be executed to fully cover all locations under IC_k**

2D Performance Analysis

- When $q_a \in (IC_{k-1}, IC_k]$

Number of plans on i^{th} contour

$$C_{bouquet}(q_a) = \sum_{i=1}^{k} [n_i \times cost(IC_i)]$$

$\rho = \max(n_i)$

$$C_{bouquet}(q_a) \leq \rho \times \sum_{i=1}^{k} cost(IC_i)$$

$$SubOpt_{bouquet}(q_a) = 4\rho \quad \text{(Using 1D Analysis)}$$

Bound for N-dimensions: $SubOpt_{bouquet}(q_a) = 4 \times \rho_{ICsurface}$

Dealing with large ρ

- In practice, ρ can often be large, even in 100s, making the performance guarantee of 4ρ impractically weak

- Reducing ρ:

  Anorexic POSP reduction
  (from CostGreedy)

MSO guarantees (compile time)

	Query (dim)	MSO Bound
TPC-H	Q5 (3D)	14.4
	Q7 (3D)	14.4
	Q8 (4D)	33.6
	Q7 (5D)	43.2
TPC-DS	Q15 (3D)	14.4
	Q96 (3D)	14.4
	Q7 (4D)	19.2
	Q19 (5D)	38.4
	Q26 (4D)	24.0
	Q91 (4D)	43.2

Empirical Evaluation

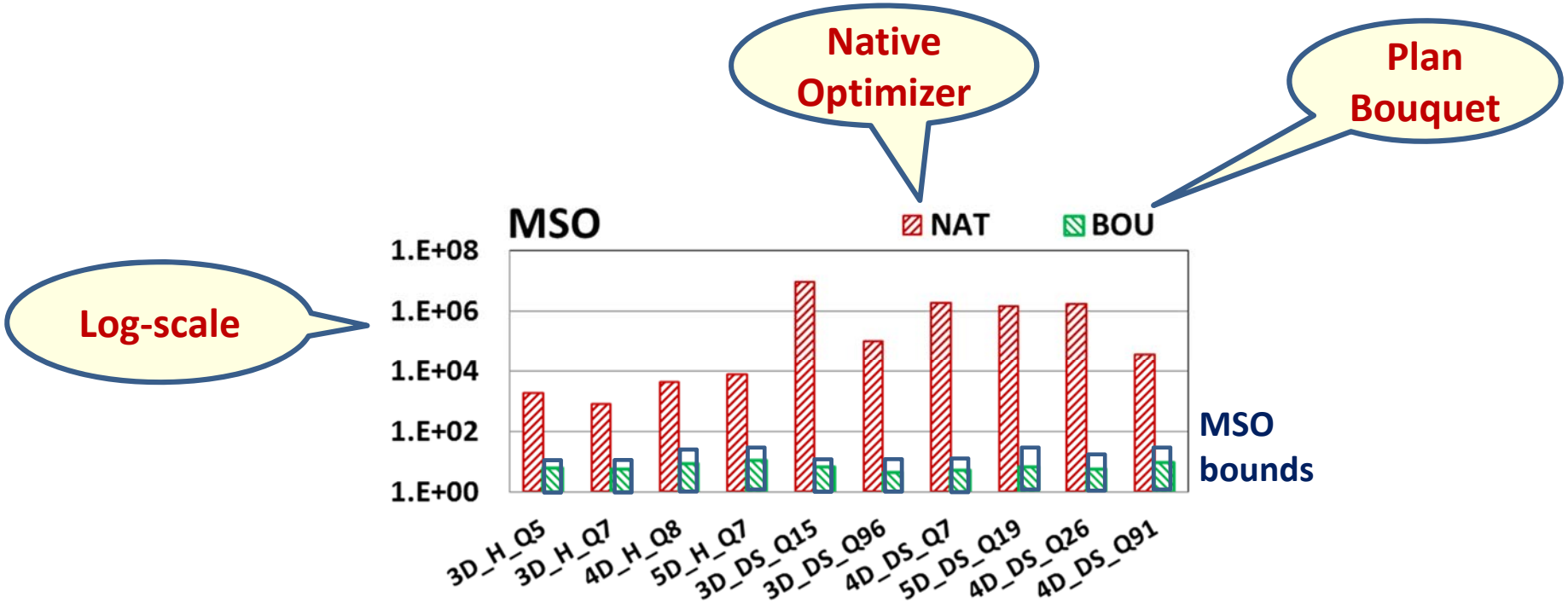
Experimental Testbed

- Database Systems: PostgreSQL and COM (commercial engine)
- Databases: TPC-H and TPC-DS (standard benchmarks)
- Physical Schema: Indexes on all attributes present in query predicates

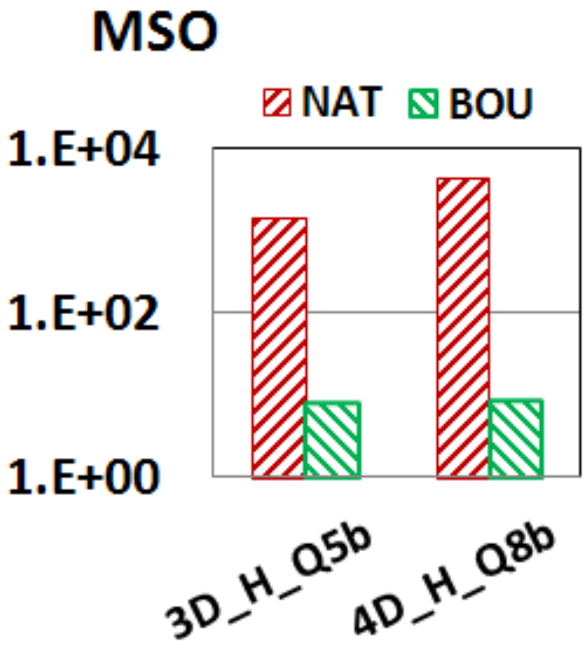
- Workload: 10 complex queries from TPC-H and TPC-DS
  - with SS having upto **5 error dimensions** (join-selectivities)

- Metrics: Computed MSO using Abstract Plan Costing over SS

Performance on PostgreSQL



Performance with Commercial System



Summary

- Plan bouquet approach achieves
  - bounded performance sub-optimality
    - using a (cost-limited) plan execution sequence guided by isocost contours defined over the optimal performance curve
  - robust to changes in data distribution
    - only q_a changes – bouquet remains same
  - easy to deploy
    - bouquet layer on top of the database engine
  - repeatability in execution strategy (important for industry)
    - q_e is always zero, depends only on q_a
    - independent of metadata contents

Limitations of PlanBouquet

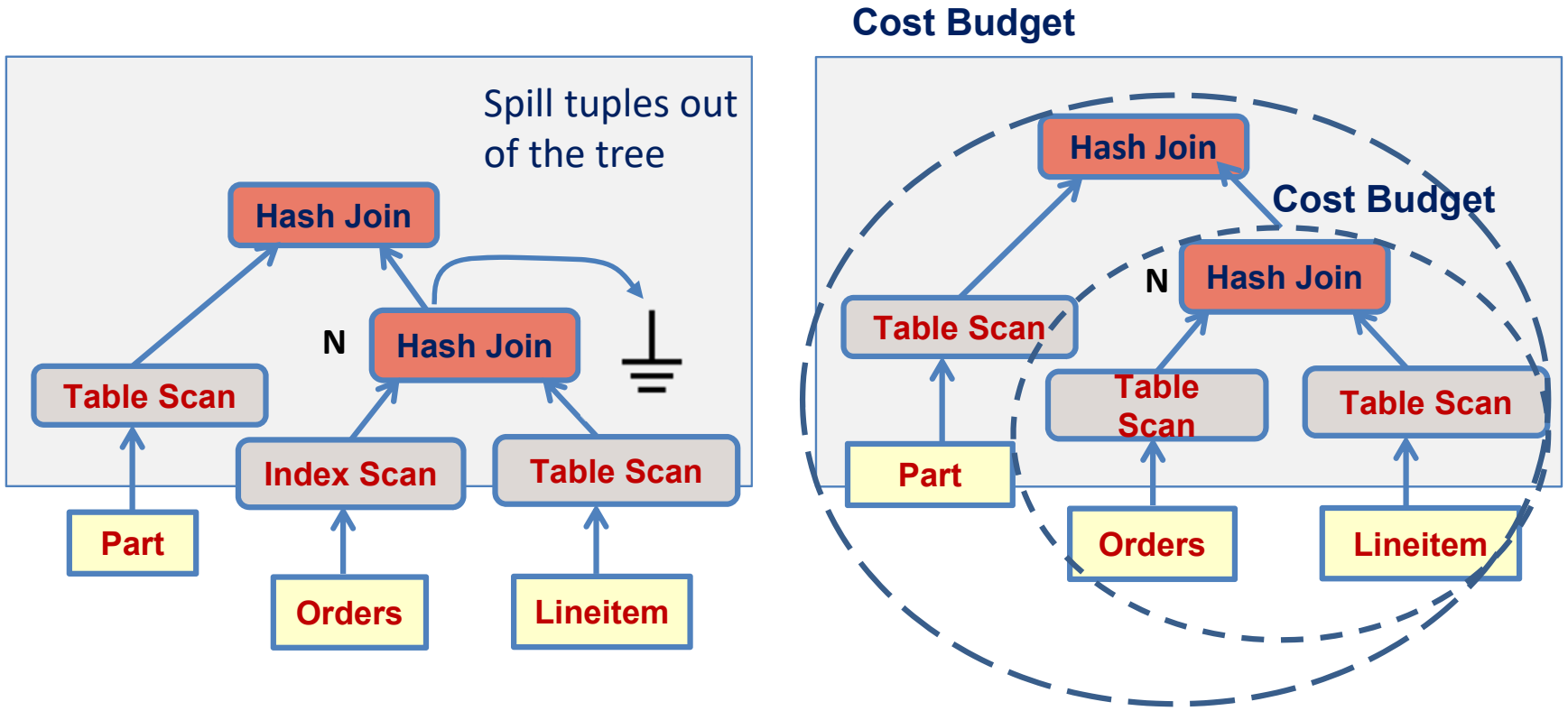
- Enormous offline computational effort to produce the plan diagram, suitable only for "canned" queries
  - Partially addressed by enumerating only the contours, not the entire diagram
- Practical guarantee values are predicated on anorexic reduction holding true
- Guarantee of 4ρ depends on plan diagram complexity, making it not portable across query optimizers, databases and hardware systems

Spill Bound

Tackling the Platform and Reduction Dependencies

Spilling

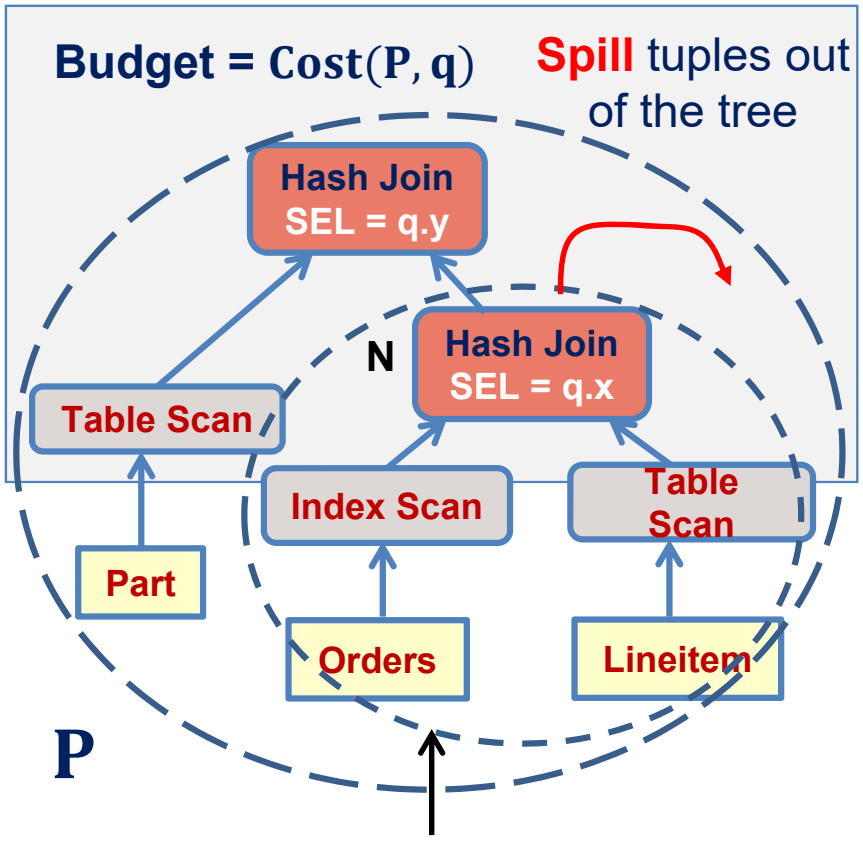
The output tuples of a node are dropped without forwarding to the downstream tree.

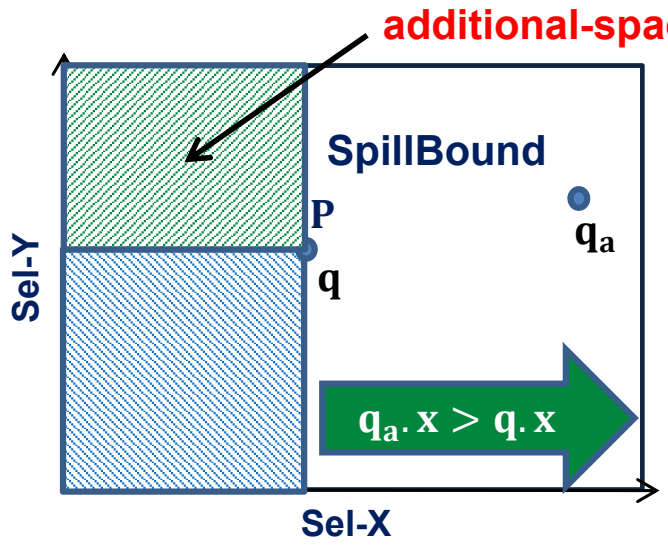


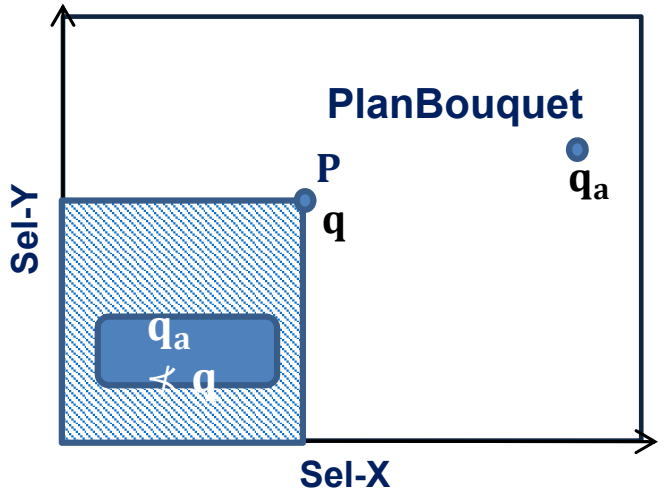
Bottom up execution

Learn single selectivity instead of combinations

When plan P is executed with cost budget $Cost(P, q)$, spilled plan P is executed with cost budget $Cost(P, q)$, and does not complete ⇒





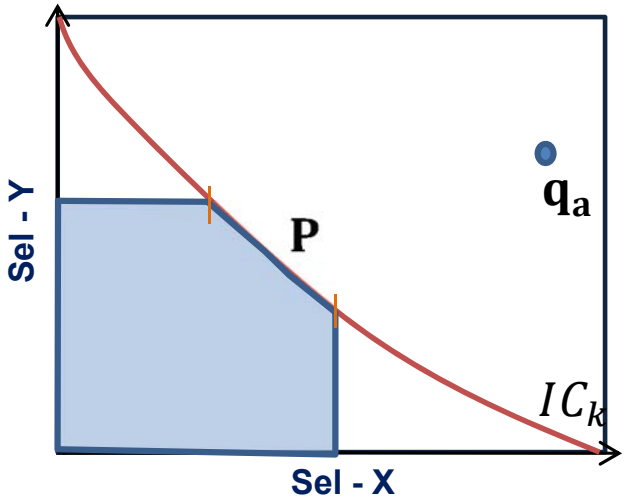


In practice, spill-mode execution can learn more than q.x selectivity

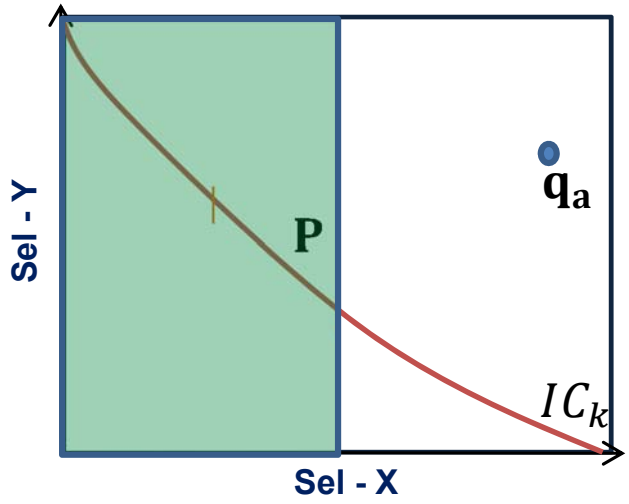
Generalizing to Contours: Hypograph to Half-space Pruning

When plan $\boldsymbol{P}$ is executed (in spill-mode) with budget equal to **Cost(IC_k)**, and does **not** complete

Hypograph Pruning (PlanBouquet)

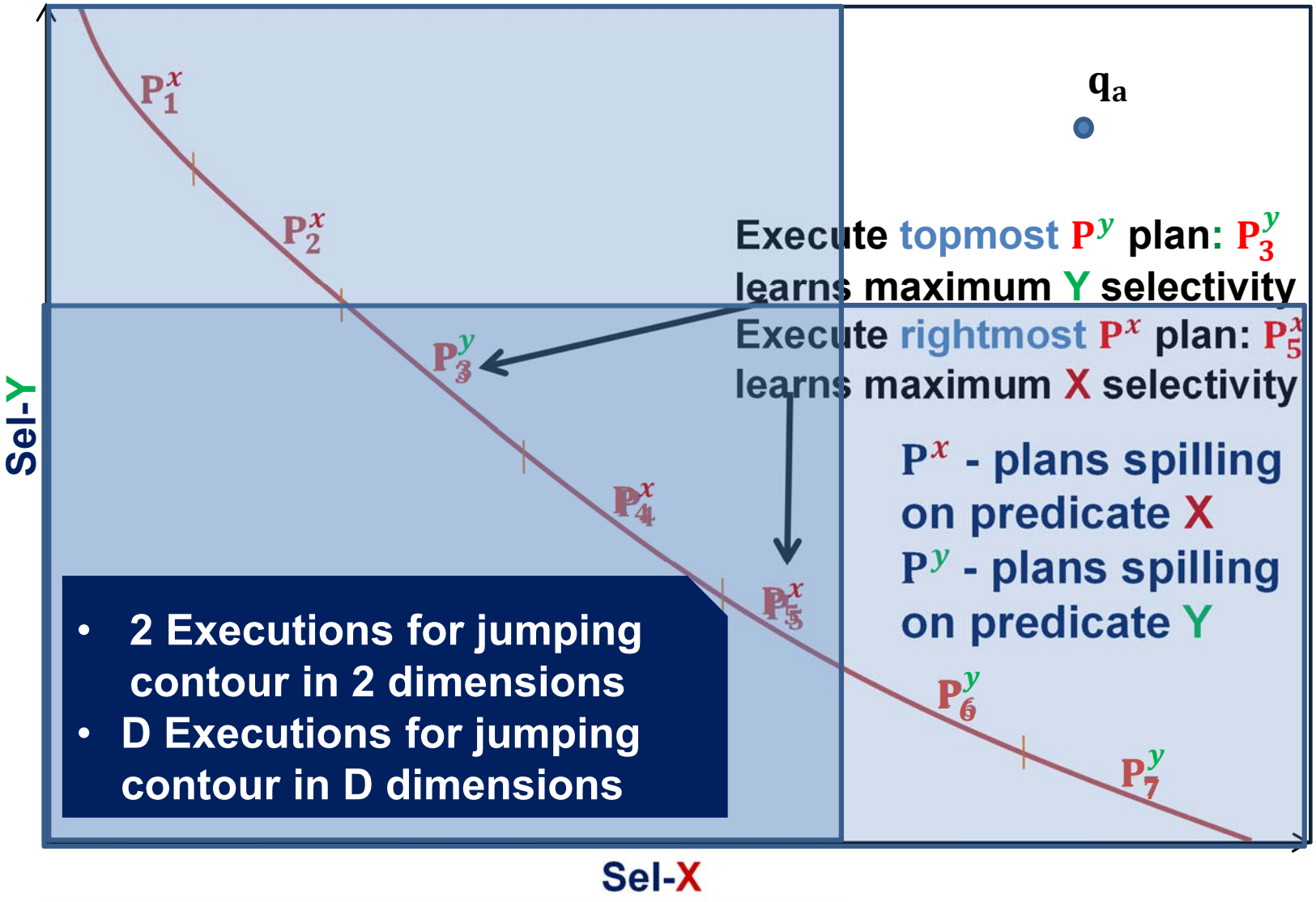


Half-space Pruning (SpillBound)



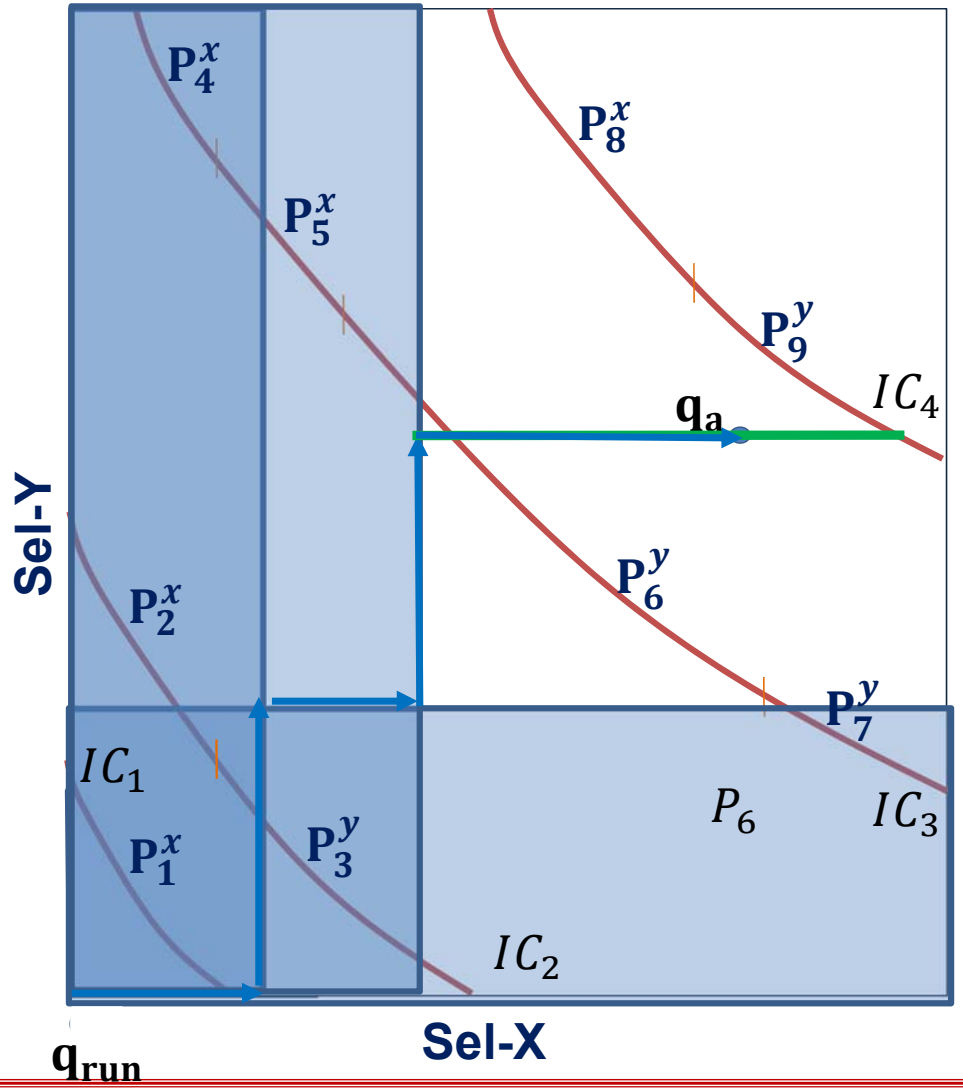
Spilling is done on the currently lowest unknown predicate in the plan tree

Benefits of Half-space pruning



Only 2 (not ρ) executions for jumping a contour!

SpillBound Execution



Procedure

In every contour starting from IC_1, execute in ***spill-mode***

1. rightmost P^x plan
2. topmost P^y plan

Repeat until all the selectivities are learnt

2D ESS Executions (spill-mode)

- IC_1: Plan P_1 (only available plan)
- IC_2: Plan P_3 (P_2 has been pruned)
- IC_3: Plan P_5 (choices: P_5, P_6)

  IC_3: Plan P_6 (complete)

1D ESS Executions (full-plan)

- IC_3: Plan P_6
- IC_4: Plan P_9 (complete)

q_{run} represents selectivity inferred during execution across error-prone predicates

Proof of 2D MSO Guarantee

Lemma: If $\mathbf{q_a}$ lies between contour IC_k and IC_{k+1}, then plan executions in IC_{k+1} will take the query to completion

Proof:

- Case 1: Right-most $\mathbf{P}^x$ and top-most $\mathbf{P}^y$ in IC_{k+1} both don't complete
  - But this contradicts $\mathbf{q_a}$ lying below contour IC_{k+1}

- Case 2: One of $\mathbf{P}^x$ and $\mathbf{P}^y$ in IC_{k+1} reach completion
  - The only dominating plan's execution in IC_{k+1} will take query to completion

Contd …

Lemma: At most one contour in $IC_1, \ldots, IC_{k+1}$, has atmost 3 executions while the rest have at most 2 executions each.

- For simplicit
- Hence, optir

For 2 dimensions, MSO guarantee is **10**

$$\textit{Total cost}$$

$$\leq \sum_{i=1}^{k} \left(2 * Cost(IC_i)\right) + 3 * Cost(IC_{k+1})$$

The guarantee is better than PlanBouquet when $\rho \geq 3$ which is 12
For instance, $\rho = 4$ for Q91 from TPC-DS

$$\leq 10 * 2^{k-1}$$

Extending to Higher Dimensions

For D dimensions,
MSO guarantee is $D^2 + 3D$

Platform-independent for a query

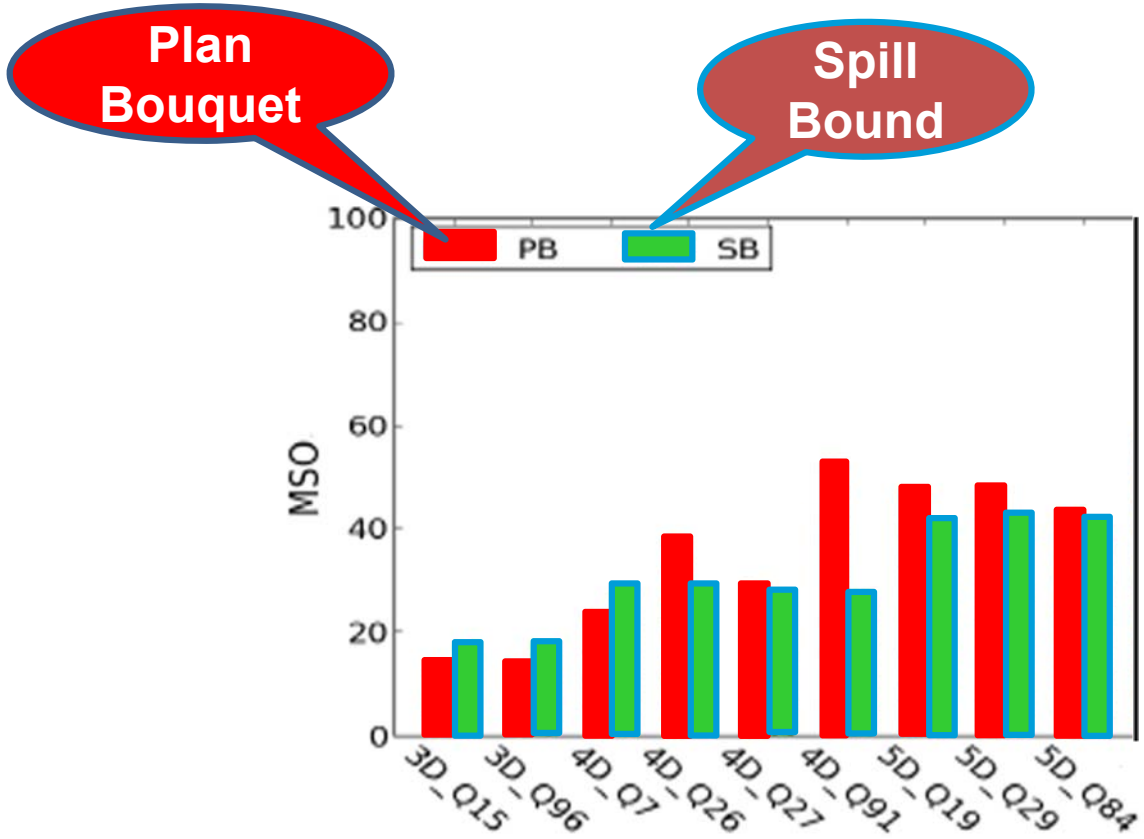
Empirical Evaluation

- Does platform-independence cause an increase in the MSO guarantee?
- Is SpillBound empirically worse than PlanBouquet for queries?

Experimental Setup

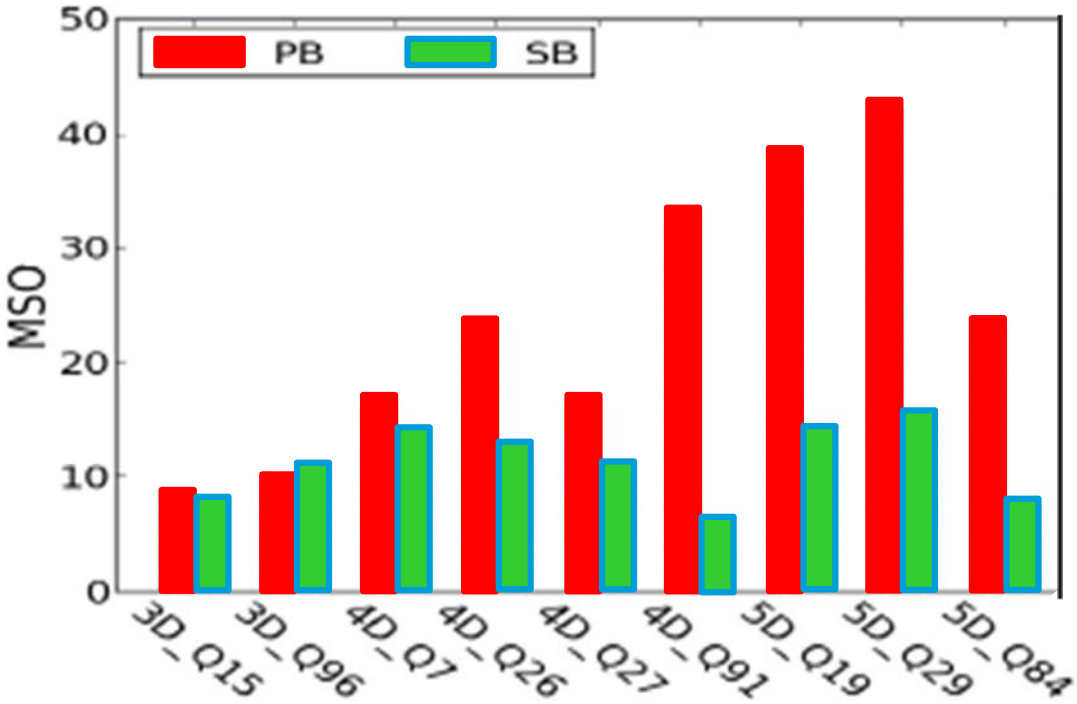
- System
  - workstation with 8 GB memory, 1TB Hard disk
- Queries
  - chosen from 100 GB TPC-DS benchmark
  - number of error-prone predicates range from 2 to 5
- Database
  - PostgreSQL: implemented spilling inside the executor module of the engine
  - Physical Schema: indexes on all columns

MSO Guarantees



SpillBound provides platform-independent MSO guarantees without quality deterioration

MSO Empirical Values



MSO empirical: SpillBound ≪ PlanBouquet !

Lower Bound on MSO

Can prove lower bound of $\Omega(D)$ among class of half-space pruning algorithms.

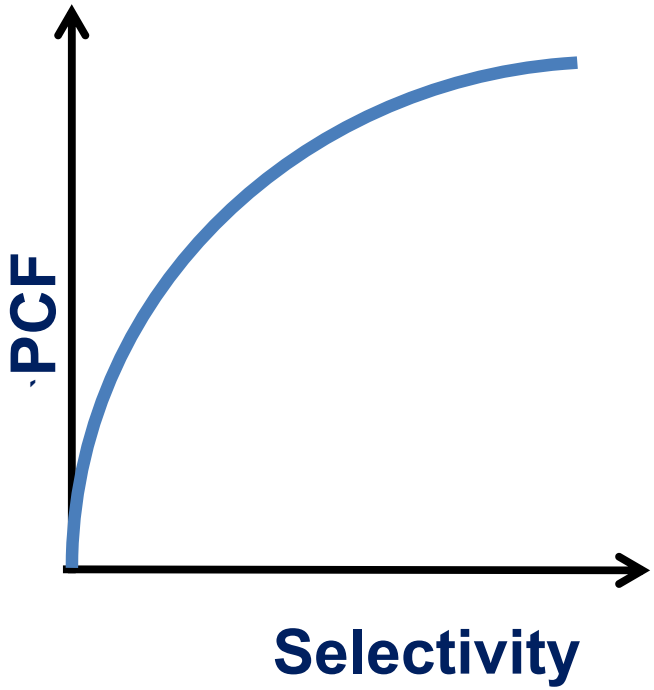
Open Problem:
Can we bridge the quadratic-to-linear MSO gap?

Frugal Spill Bound

Tackling the Compilation Overheads

Frugal SpillBound

- Apart from PCM, additional assumption that Plan Cost Functions satisfy concave property



Extensively validated the assumption on TPCDS benchmark

Results

- Using concavity assumption, for upto twice increase in MSO guarantees, the overheads reduction factor is

$$\Omega\left(\frac{r^D}{(D \log_2 r)^{D-1}}\right)$$

- In theory, the reduction factor is two orders of magnitude. Empirically, it often reaches three orders of magnitude.

- For D=5 and r =100, the compilation efforts reduces from few days to few minutes on contemporary servers

Dimensionality Reduction

- Both the MSO guarantees and the compilation overheads of SpillBound, are prey to the curse of dimensionality
- For queries with large number of dimensions, practical experience suggests that only some of these dimensions are relevant wrt robustness
- Open Problem: Identify such dimensions and remove them, thereby resulting in improved MSO guarantees and reduced compilation overheads

Stage 4: Robust Cost Models

Approaches

- Learning-based approaches (ICDE 2009, ICDE 2012 [4])
  - received Influential Paper Award in this conference!
  - Several papers on Arxiv in recent months
    - **Plan-Structured Deep Neural Network Models for Query Performance Prediction (arXiv:1902.00132, Jan 2019)**

- Statistical approaches (ICDE 2013 [40], PVLDB 2013 [41], SIGMOD 2016 [42])

Optimizer's Cost Estimates: Unusable

Direct Scaling:
Predict the *execution time T*
by *scaling* the *cost estimate C*,
i.e., $T = a \cdot C$

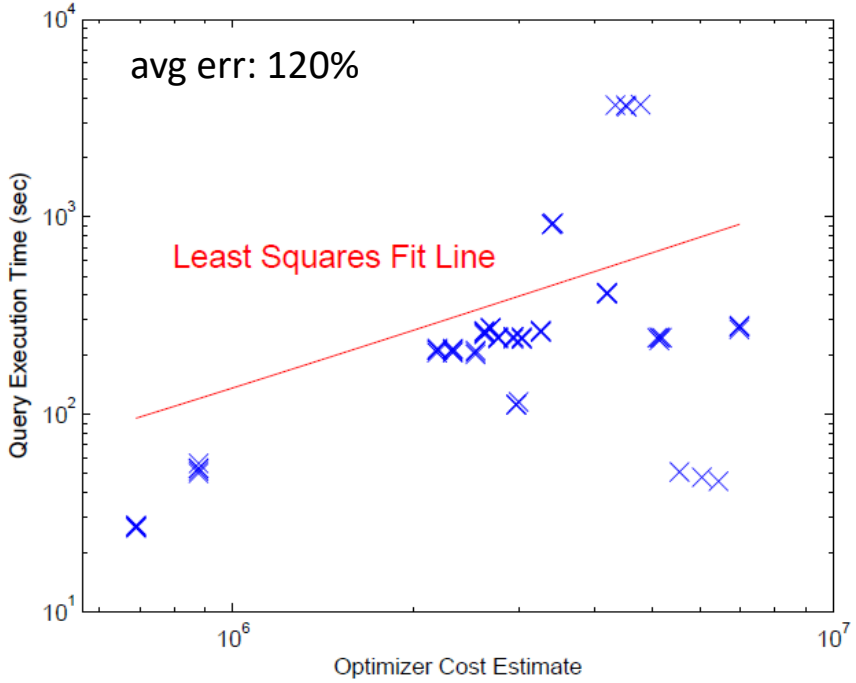


Fig. 5 of [4]

Why Does Direct Scaling Fail?

- PostgreSQL's cost model

$$C = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o$$

Scaling

$$T = a \cdot C = c_s' \cdot (n_s + n_r \frac{c_r}{c_s} + n_t \frac{c_t}{c_s} + n_i \frac{c_i}{c_s} + n_o \frac{c_o}{c_s})$$

$$c_s' = a \cdot c_s = a \cdot 1.0 = a$$

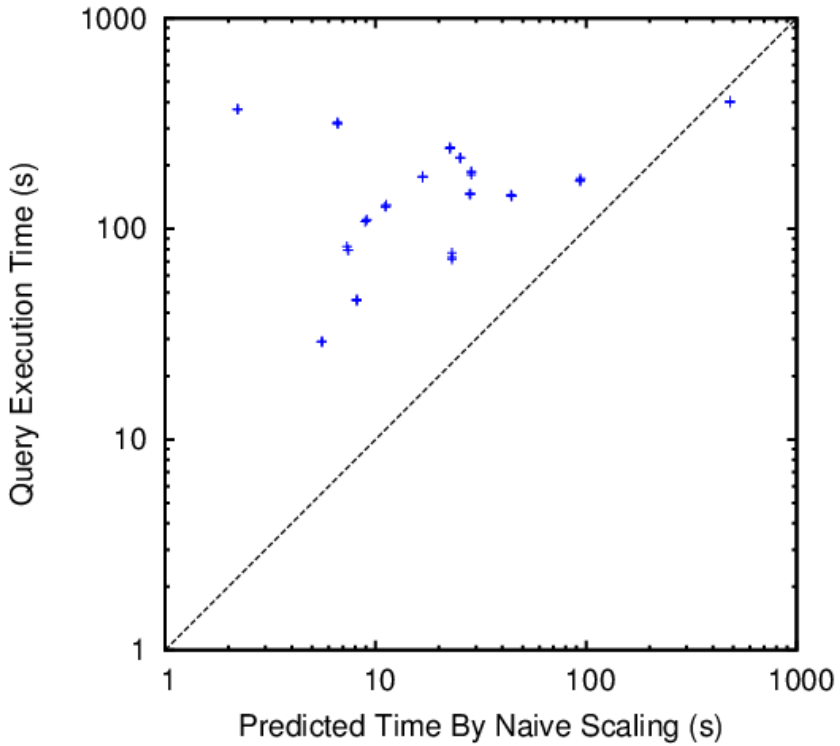
Should be correct!

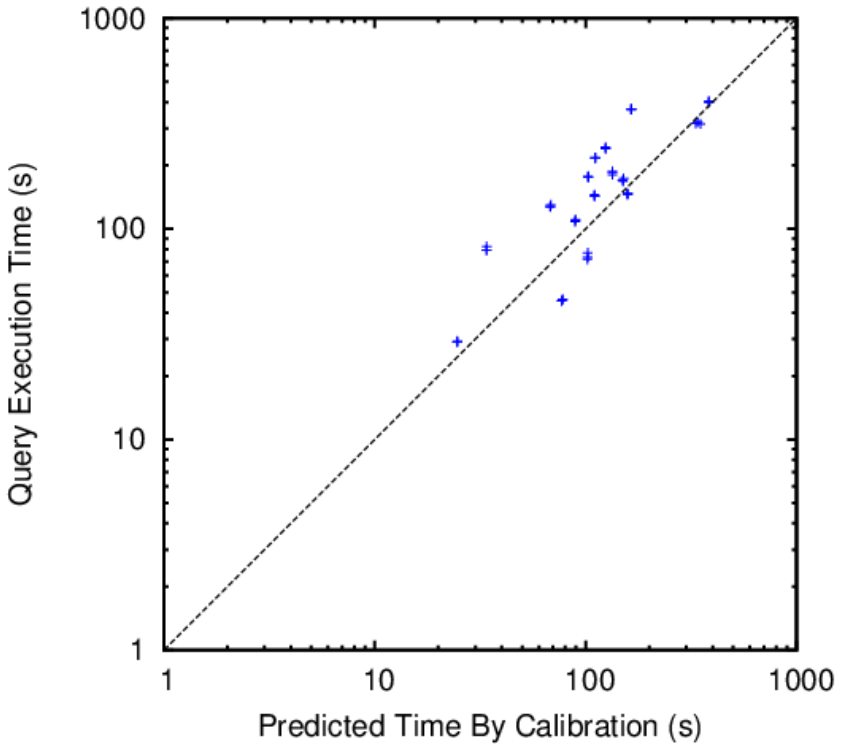
Cost Unit	Value
c_s: seq_page_cost	1.0
c_r: rand_page_cost	4.0
c_t: cpu_tuple_cost	0.01
c_i: cpu_index_tuple_cost	0.005
c_o: cpu_operator_cost	0.0025

- Assumptions for scaling fail in practice
  - *Ratios* between the *c* values are incorrect.
  - *n* values are incorrect.
- Solution: Proper calibration

Calibrated *c* and *n*

– Cost models become much more effective.





Prediction by Scaling:

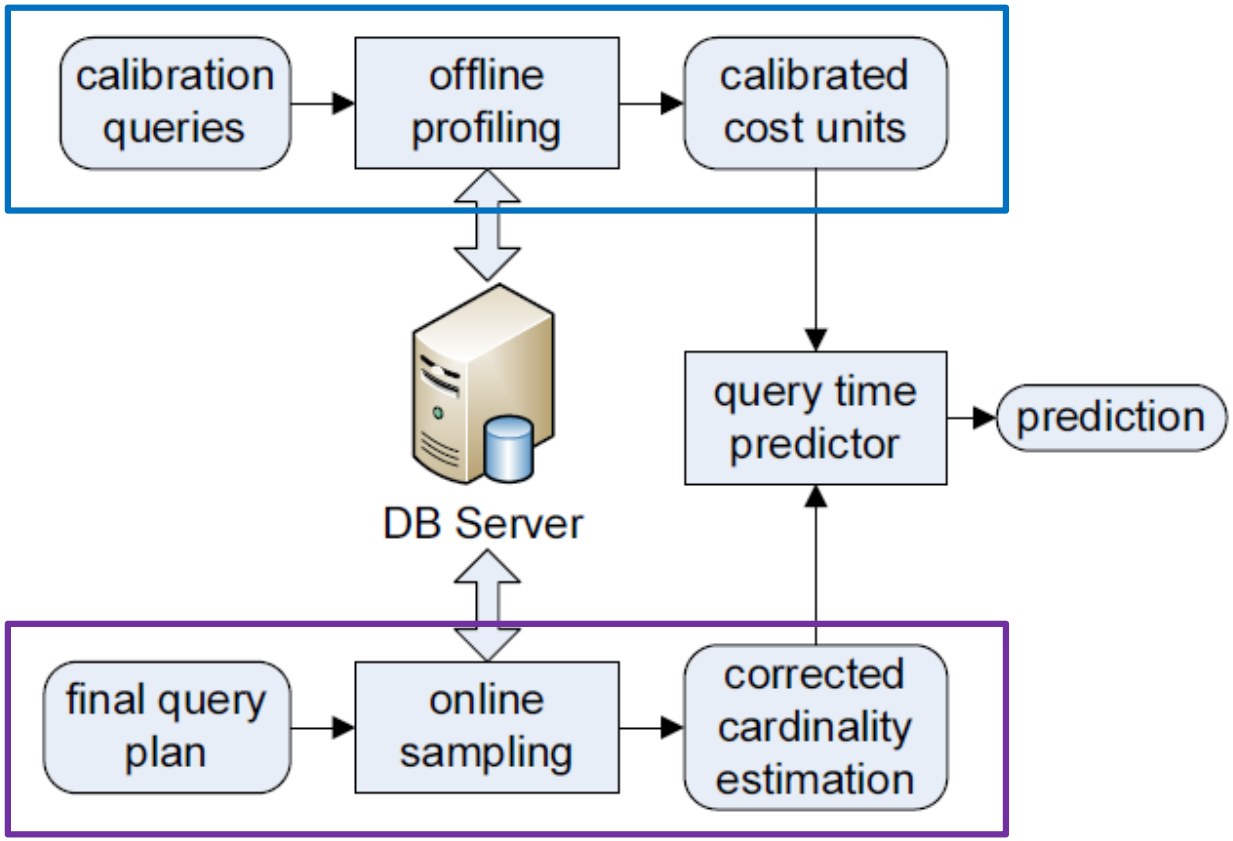
$$T_{pred} = a \cdot (\sum c \cdot n)$$

Prediction by Calibration:

$$T_{pred} = \sum c' \cdot n'$$

Main Idea

- Calibrate *c*: *use profiling queries*
- Calibrate *n*: *refine cardinality estimates*



Calibrating the *c values*

Cost Unit
c_s: seq_page_cost
c_r: rand_page_cost
c_t: cpu_tuple_cost
c_i: cpu_index_tuple_cost
c_o: cpu_operator_cost

- Basic idea (an example)
  - Want to know the true c_t and c_o

q₁: select * from R
q₂: select count(*) from R

R in memory

$$t_1 = c_t \cdot n_t$$
$$t_2 = c_t \cdot n_t + c_o \cdot n_o$$

- General case
  - *k* cost units (i.e., *k* unknowns) ⇒ *k* queries (i.e., *k* equations)
  - *k* = 5 in the case of PostgreSQL

How to Pick Profiling Queries?

- Completeness
  - Each *c* should be covered by at least one query.
- Conciseness
  - Set of queries is incomplete if any query is removed.
- Simplicity
  - Each query should be as simple as possible, both for efficiency and for unambiguous measurement.

Profiling Queries For PostgreSQL

Isolate the unknowns and solve them *one per equation*!

q₁: select * from R	R in memory	$t_1 = c_t \cdot n_{t1}$
q₂: select count(*) from R	R in memory	$t_2 = c_t \cdot n_{t2} + c_o \cdot n_{o2}$
q₃: select * from R where R.A < a (R.A with an index)	R in memory	$t_3 = c_t \cdot n_{t3} + c_i \cdot n_{i3} + c_o \cdot n_{o3}$
q₄: select * from R	R on disk	$t_4 = c_s \cdot n_{s4} + c_t \cdot n_{t4}$
q₅: select * from R where R.B < b (R.B *unclustered* index)	R on disk	$t_5 = c_s \cdot n_{s5} + c_r \cdot n_{r5} + c_t \cdot n_{t5} + c_i \cdot n_{i5} + c_o \cdot n_{o5}$

Calibrating the *n values*

- The *n* values are *functions* of *N* values (i.e., input cardinalities).
  - Calibrating the *n values* ⇒ Calibrating the *N* values

Example 1 (In-Memory Sort)

$$sc = \lceil 2 \cdot N_t \cdot \log N_t \rceil \cdot c_o + tc\ of\ child$$

$$rc = c_t \cdot N_t$$

n_o

Example 2 (Nested-Loop Join)

$$sc = sc\ of\ outer\ child + sc\ of\ inner\ child$$

$$rc = c_t \cdot N_t^o \cdot N_t^l + N_t^o \cdot rc\ of\ inner\ child$$

n_t

sc: start-cost rc: run-cost $tc = sc + rc$: total-cost

N_t: # of input tuples

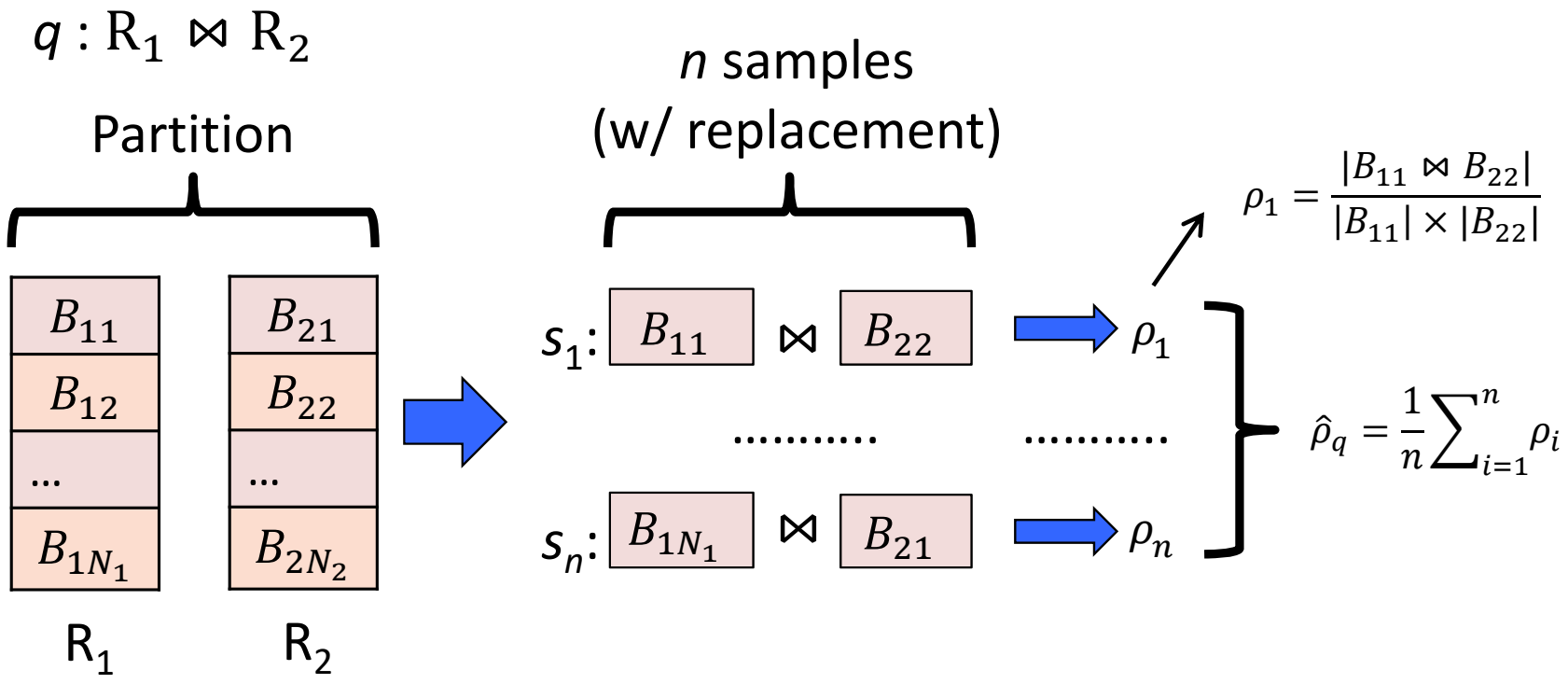
Refine Cardinality Estimates

- Different perspective than the norm

	Traditional Role (Query Optimization)	**Our Case (Execution Time Prediction)**
# of Plans	Hundreds/Thousands of	1
Time per Plan	Must be very short	Can be a bit *longer*
Precision	Important	*Critical*
Approach	Histograms (dominant)	*Sampling* (one option)

A Sampling-Based Estimator

- Estimate the *selectivity* ρ_q of a select-join query q.
  [Haas et al., J. Comput. Syst. Sci. 1996]



The estimator $\hat{\rho}_q$ is *unbiased* and *strongly consistent*!

Cardinality Refinement Algorithm

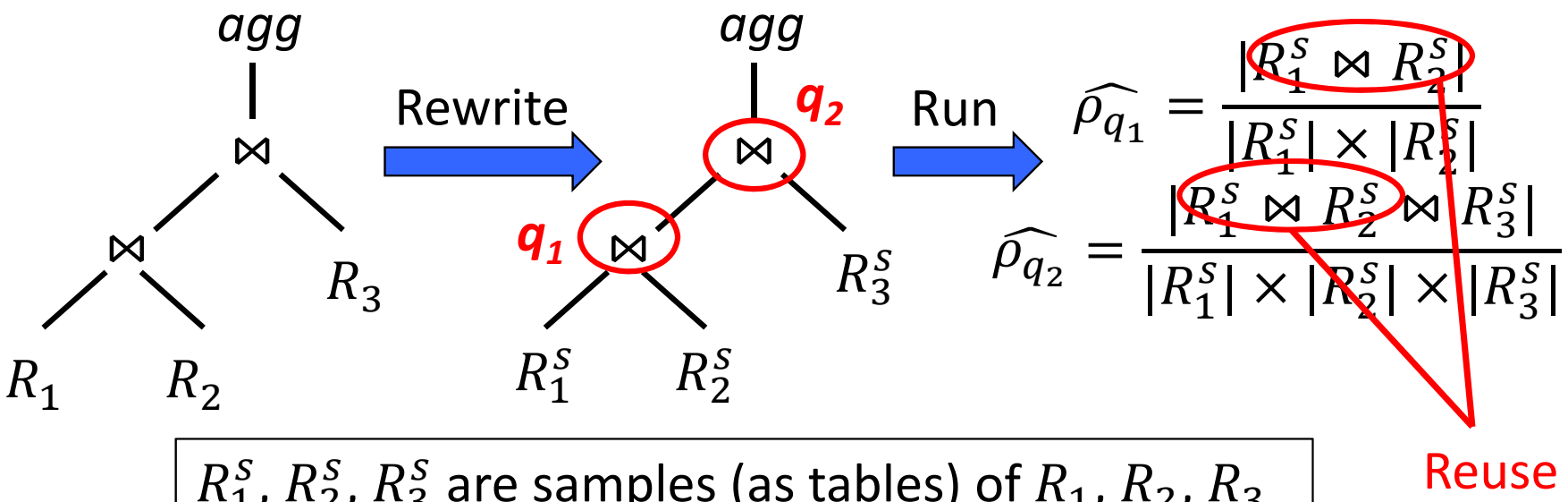
- Design the refinement algorithm based on the previous sampling formula.

Problem	Solution
The estimator needs *random* I/Os at *runtime* to take samples.	Take samples *offline* and store them as tables in the database.
Query plans usually contain *more than one* operator.	Estimate multiple operators in a *single* run, by *reusing* partial results.
The estimator only works for *select/join* operators.	Rely on PostgreSQL's cost models for *aggregates*.

Cardinality Refinement Algorithm (Example)

Plan for q:

$$q_1 = R_1 \bowtie R_2$$
$$q_2 = R_1 \bowtie R_2 \bowtie R_3$$



R_1^s, R_2^s, R_3^s are samples (as tables) of R_1, R_2, R_3

For *agg*, use PostgreSQL's estimates based on the *refined* input estimates from q_2.

Experimental Settings

- PostgreSQL 9.0.4, Linux 2.6.18
- TPC-H 1GB and 10GB databases
  - Both uniform and skewed data distribution
- Two different hardware configurations
  - PC1: 1-core 2.27 GHz Intel CPU, 2GB memory
  - PC2: 8-core 2.40 GHz Intel CPU, 16GB memory

Calibrating Cost Units

PC1:

Cost Unit	Calibrated (ms)	Calibrated (normalized to c_s)	Default
c_s: seq_page_cost	5.53e-2	1.0	1.0
c_r: rand_page_cost	6.50e-2	1.2	4.0
c_t: cpu_tuple_cost	1.67e-4	0.003	0.01
c_i: cpu_index_tuple_cost	3.41e-5	0.0006	0.005
c_o: cpu_operator_cost	1.12e-4	0.002	0.0025

PC2:

Cost Unit	Calibrated (ms)	Calibrated (normalized to c_s)	Default
c_s: seq_page_cost	5.03e-2	1.0	1.0
c_r: rand_page_cost	4.89e-1	9.7	4.0
c_t: cpu_tuple_cost	1.41e-4	0.0028	0.01
c_i: cpu_index_tuple_cost	3.34e-5	0.00066	0.005
c_o: cpu_operator_cost	7.10e-5	0.0014	0.0025

Prediction Precision

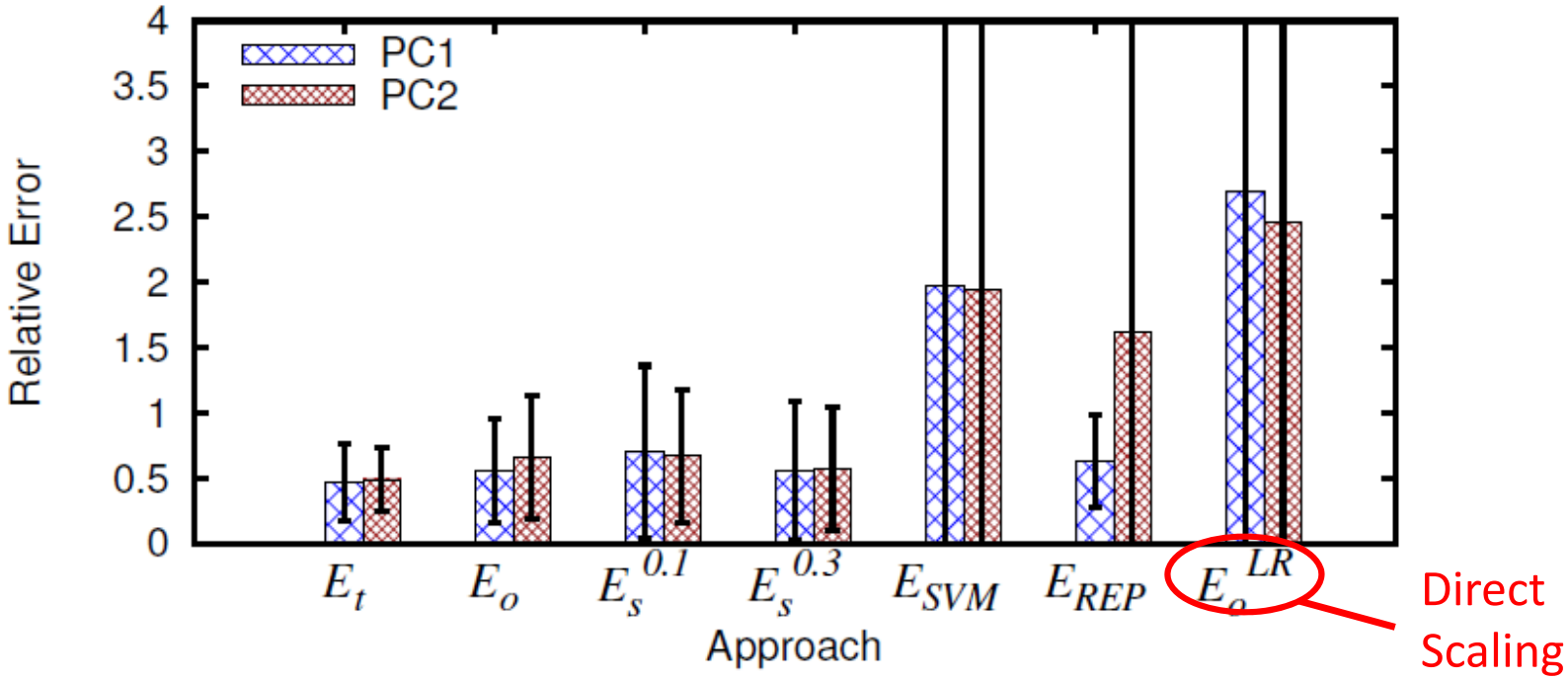
- Metric of precision
  - Mean Relative Error (MRE)
  - (questionable as compared to q-error)

$$\frac{1}{M}\sum_{i=1}^{M}\frac{|T_i^{pred} - T_i^{act}|}{T_i^{act}}$$

- Dynamic database workloads
  - Unseen queries frequently occur
- Compare with existing approaches
  - Direct scaling
  - Machine learning approaches

Precision on TPC-H 1GB DB

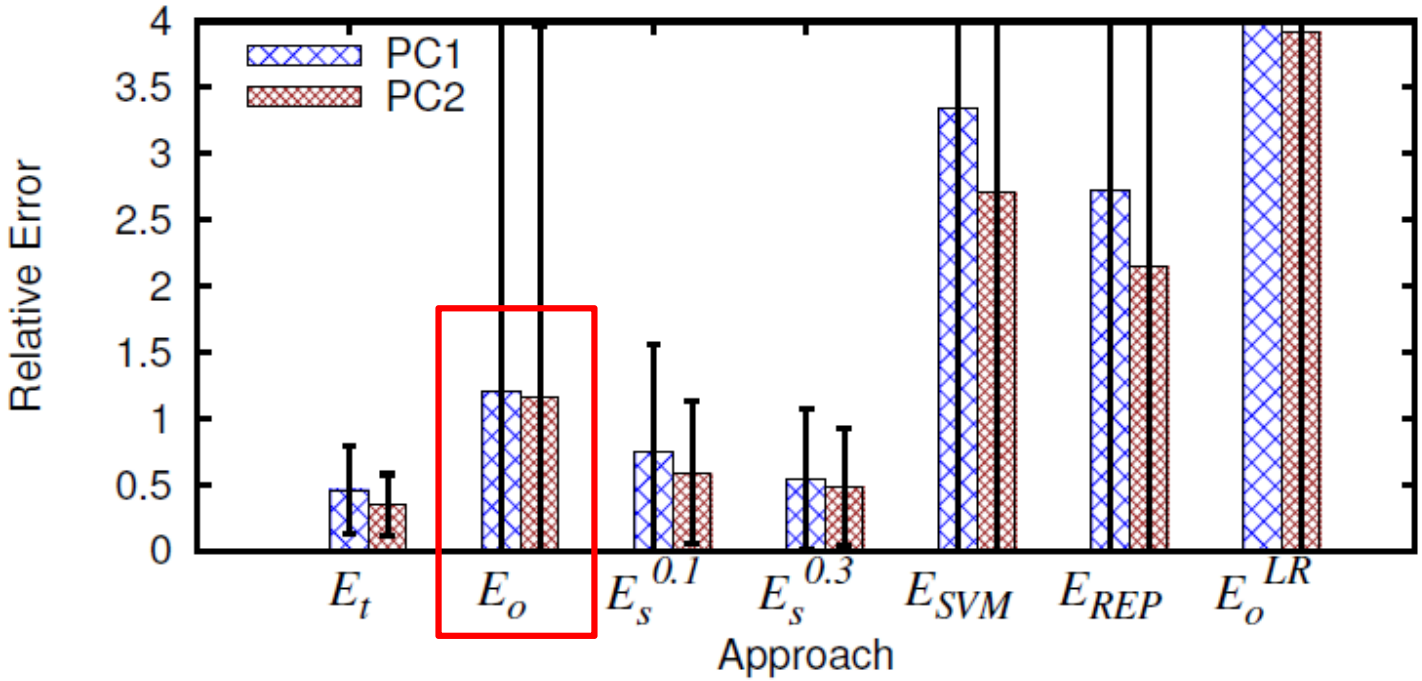
Uniform data:



E_t: c's (calibrated) + n's (*true* cardinalities)
E_o: c's (calibrated) + n's (cardinalities by *optimizer*)
E_s: c's (calibrated) + n's (cardinalities by *sampling*)

Precision on TPC-H 1GB DB (Cont.)

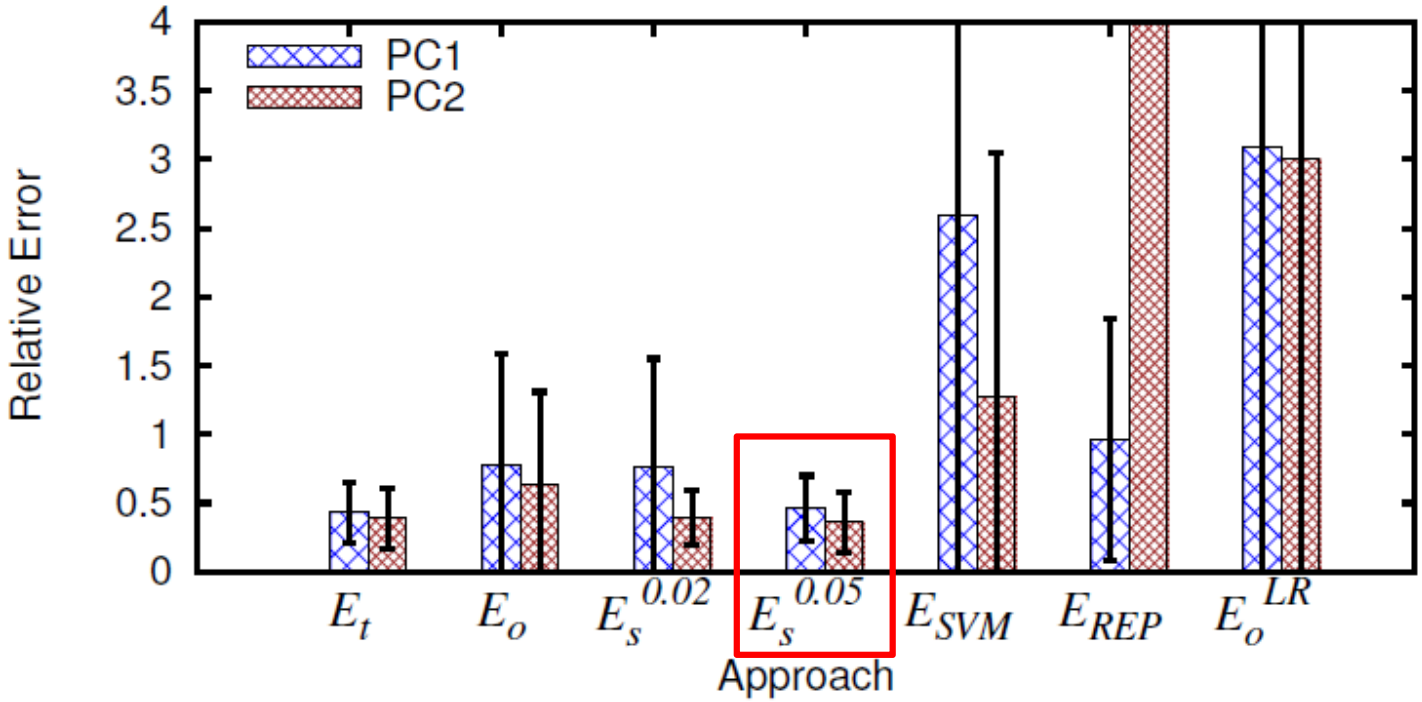
Skewed data:



E_t: *c*'s (calibrated) + *n*'s (*true* cardinalities)
E_o: *c*'s (calibrated) + *n*'s (cardinalities by *optimizer*)
E_s: *c*'s (calibrated) + *n*'s (cardinalities by *sampling*)

Precision on TPC-H 10GB DB

Uniform data (similar results on skewed data):



E_t: c's (calibrated) + n's (*true* cardinalities)
E_o: c's (calibrated) + n's (cardinalities by *optimizer*)
E_s: c's (calibrated) + n's (cardinalities by *sampling*)

Overhead of Sampling

- Additional overhead is measured as $\frac{t_{sampling}}{t_{query}}$
- More samples mean higher additional overhead
- For close-to-ideal prediction on 1GB DB
  - 30% samples (0.3GB) ⇒ 20% additional overhead
- For close-to-ideal prediction on 10GB DB
  - 5% samples (0.5GB) ⇒ 4% additional overhead

Conclusion

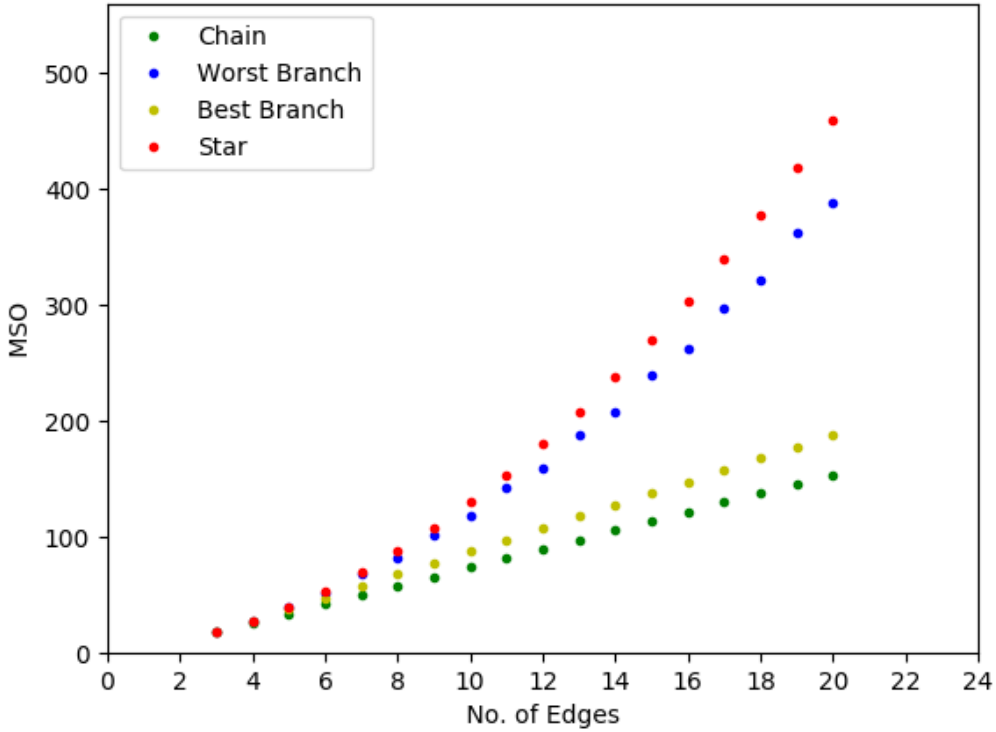
- Systematic framework to *calibrate* the *cost units* and *refine* the *cardinality estimates* used by current cost models.

- Showed that current cost models are much more *effective* in query execution time prediction after *proper calibration*, and the *additional overhead* is *affordable* in practice.

Stage 5: Future Research

1) Structure of Query Graphs

- Graph structure (chain, star, cycle, etc.) has significant impact on robustness guarantees
  - Tighter guarantees for chain (8D – 6) as compared to star (D^2+3D)



Open Problem: MSO derivations based on query graph type

2) Graceful Performance Degradation

- i.e. prevent Boeing 737 Max situations!
- Like SmoothScan or CliffGuard (SIGMOD 2015) for physical design
- Performance "walls", not cliffs, occur with Plan Bouquet, between queries located just below a contour and just above the contour.
- **Open Problem:** Design a smooth version of Plan Bouquet

3) Refined Cost Model Calibration

- Calibration assumed the Postgres basic 5-parameter model as a given for the entire suite of operators.

- **Open Problem:** Add operator-specific features and operator-specific calibration of the coefficients, and see if accuracy can be improved.

4) Robustness Benchmarks

- Standard industry benchmarks (e.g. TPC-DS) are oriented towards performance, not robustness.
- Recent proposals on benchmarks:
  - Optimizer Benchmark (OptMark) (CIKM 16 [28])
    - TPC-DS synthetic data, examines plan coverage and estimation of plans better than optimizer's choice; does not cover magnitude of cost differences
  - Join-Order Benchmark (JOB) (VLDBJ 18 [26])
    - Based on IMDB real data with heavy skew and correlation, and join-heavy queries, q-error
  - Optimizer Torture Test (OTT) (SIGMOD 16 [43])
    - Two-column relations, one join attribute and one selection, the two columns are highly correlated (in fact, identical values!)
- **Open Problem: Design non-pathological realistic benchmarks that highlight the same issues**

5) Machine Learning

- Estimating Join Selectivities using Bandwidth-optimized Kernel Density Models
  (TU Berlin, VLDB 2017 [24])
- Neo: A Learned Query Optimizer (Brown/MIT, arXiv:1904.03711, April 2019)
- Learned Cardinalities: Estimating Correlated Joins with Deep Learning
  (TU Munich, CIDR 2019 [25])
- Multi-attribute Selectivity Estimation Using Deep Learning
  (UT Arlington, arXiv:1903.09999, March 2019)
- Towards a Hands-Free Query Optimizer through Deep Learning (Brown, CIDR 2019)
- Plan-Structured Deep Neural Network Models for Query Performance Prediction
  (Brown, arXiv:1902.00132, Jan 2019)
- Flexible Operator Embeddings via Deep Learning (Brown, arXiv: 1901.09090, Jan 2019)
- Learning to Optimize Join Queries with Deep Reinforcement Learning
  (UC Berkeley, arXiv:1808.03196)
- Learning State Representations for Query Optimization with DRL
  (UWash, DEEM Workshop@SIGMOD 2018)

Limitations of Learning approaches

- Universality
  - Ability to handle unseen adhoc queries is suspect
- Explainability
  - Does not provide an intuitive confirmation of the approach
- Guarantees
  - Average case may be excellent, but worst-case can be arbitrarily poor
- Heavy-weight
  - Require expensive training phase

Open Problem:

Compare Algorithmic (Algebra+Geometry) vs Function-fitting approaches

RECOMMENDED SOLUTION

Good news

The proposed techniques are complementary and can work together!

Robust Query Processing

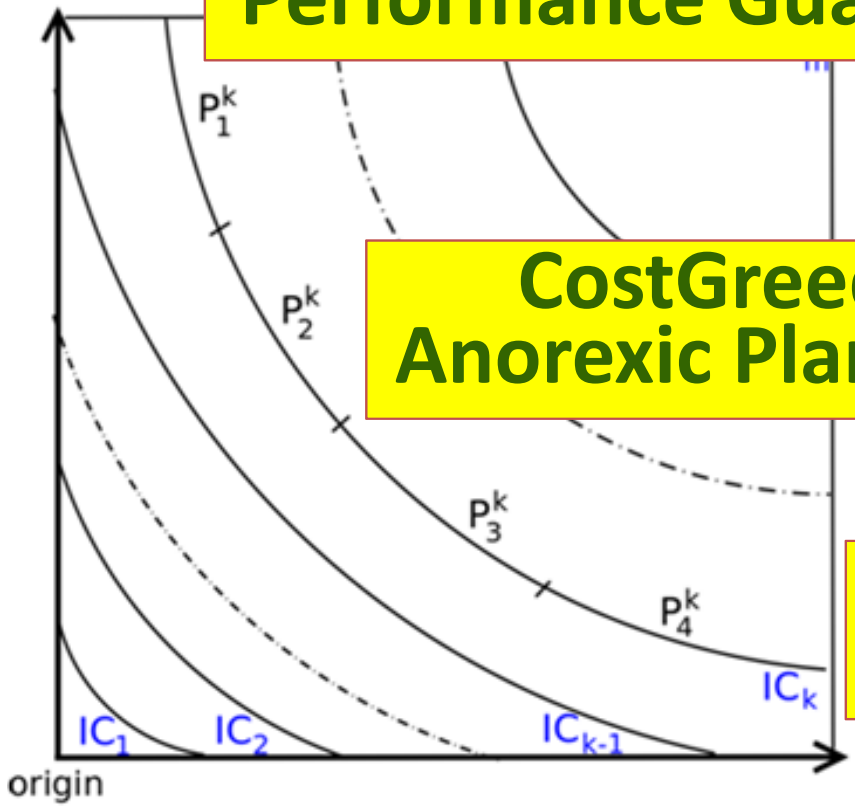
- Implement “universal” operators
  - SmoothScan, G-Join, etc.
- Implement “anorexic” reduction
  - Cost Greedy
- Implement PlanBouquet-style execution
  - SpillBound for canned, FrugalBound for ad-hoc queries
- Implement Calibration-based cost modeling
  - Augment off-line calibration with online tuning

New RQP Architecture: Plan-level

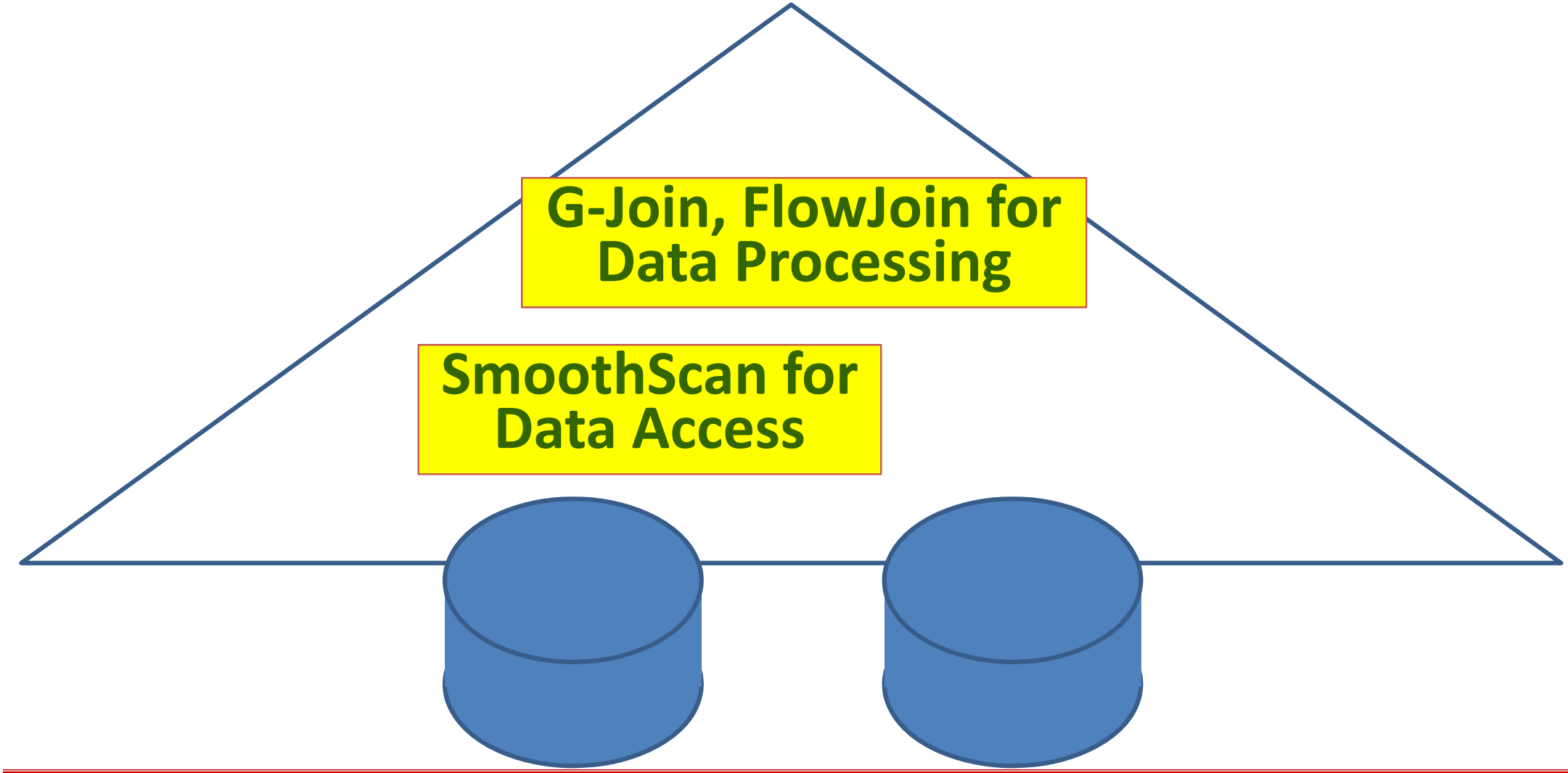
SpillBound for Performance Guarantees

CostGreedy for Anorexic Plan Density

Q-Calibrated Cost Model for Contours



New RQP Architecture: Intra-Plan



END TUTORIAL