Beyond EXPLAIN

Query Optimization From Theory To Code

Yuto Hayamizu
Ryoji Kawamichi

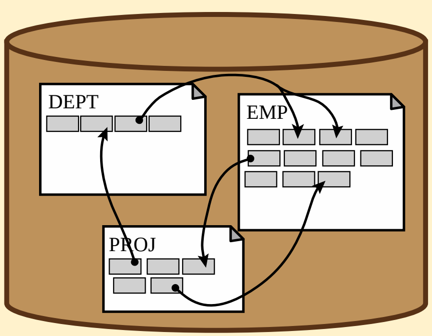
2016/5/20
PGCon 2016 @ Ottawa

Historically …

Before Relational …

- Querying was ***physical***
- Need to understand physical organization
- Navigate query execution by yourself

"Which file is this table stored in?"
"How are records linked?"
"Which access path is fast for this table?"
"What is the best order of joining tables"
…

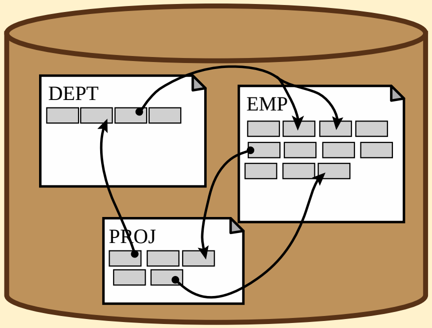


Historically …

Before Relational …

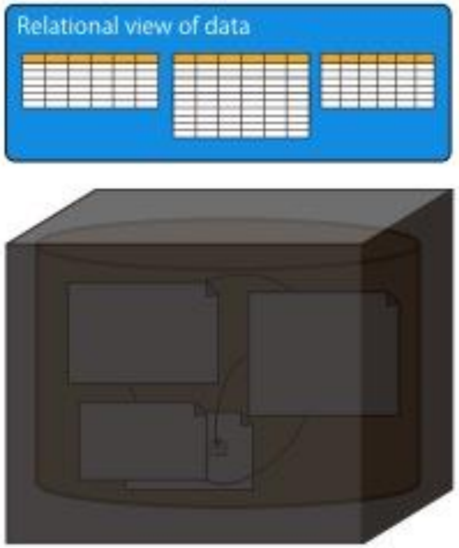
- Querying was ***physical***
- Need to understand physical organization
- Navigate query execution by yourself

"Which file is this table stored in?"
"How are records linked?"
"Which access path is fast for this table?"
"What is the best order of joining tables"
…



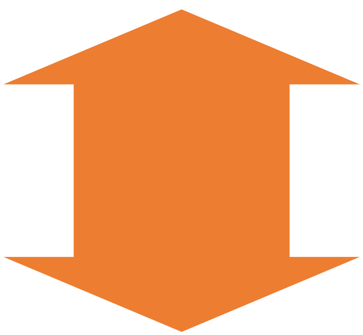
After Relational …

- Querying is logical
- Physical organization is black-boxed
- Just declare what you want

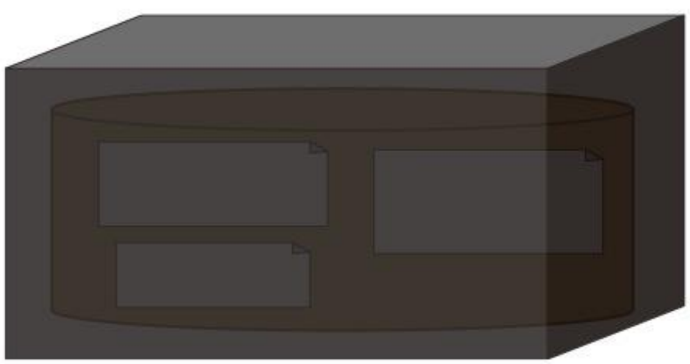


Fill the Gap: *Physical* and Logical


```
SELECT * FROM DEPT D, EMP E
  WHERE E.D_ID = D.ID AND ...
```



Query Optimizer



- Storage I/O strategy
- Access path selection
- Join method selection
- Aggregation, sorting
- Resource allocation
- ...

If optimizer perfectly fills the gap…

We don’t need EXPLAIN

Reality Is Tough

- Optimizer is NOT PERFECT
  - Generated plans are not always optimal, sometimes far from optimal
- We have to take care of physical behavior

- That's why EXPLAIN is so much explained

Go Beyond EXPLAIN

- Deeper understanding of optimization, better control of your databases

- Theoretical fundamentals of query optimization
  - From basic framework to cutting-edge technologies
- PostgreSQL Optimizer implementation
  - Focusing on basic scan and join methods
  - Behavior observation with TPC-H benchmark

Outline

- Introduction
- Theory: Query Optimization Framework
- Code: PostgreSQL Optimizer
- Theory: Cutting-Edge Technologies Overview
- Summary

Query Optimization Framework

- **Cost-based optimization**
  - Plan selection with estimated execution cost
  - Most of modern optimizers, including PostgreSQL, are cost-based

- Rule-based optimization
  - Plan selection with heuristically ranked rules
  - Easy to produce the same result
  - Hard to evaluate wide variety of plans
  - Ex) Oracle (~10g), Hive (~0.13)

Main Challenges in Cost-based Optimization

- Cost modeling is HARD
  - Overhead of CPU, I/O, memory access, network, …
- Cardinality estimation is HARD
  - Output size of scans, joins, aggregations, …
- Join ordering search is HARD
  - Combinatorial explosion of join ordering and access path
  - Exhaustive search is NP-hard

System-R optimizer (1979)

Access Path Selection
in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

IBM Research Division, San Jose, California 95193

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for

- "The standard"
  - Cost estimation with I/O and CPU
  - Cardinality estimation with table statistics
  - Bottom-up plan search
- Many of modern optimizers are "System-R style"
  - PostgreSQL, MySQL, DB2, Oracle, …

Cost/Cardinality Estimation

```
COST = [#page fetched] + W * [#storage API calls]
```

I/O cost — CPU cost — weight parameter

- `[#page fetched]`,`[#storage API calls]` are estimated with cost formula and following statistics
    - `NCARD(T)` … the cardinality of relation `T`
    - `TCARD(T)` … the number of pages in relation `T`
    - `ICARD(I)` … the number of distinct keys in index `I`
    - `NINDX(I)` … the number of pages in index `I`

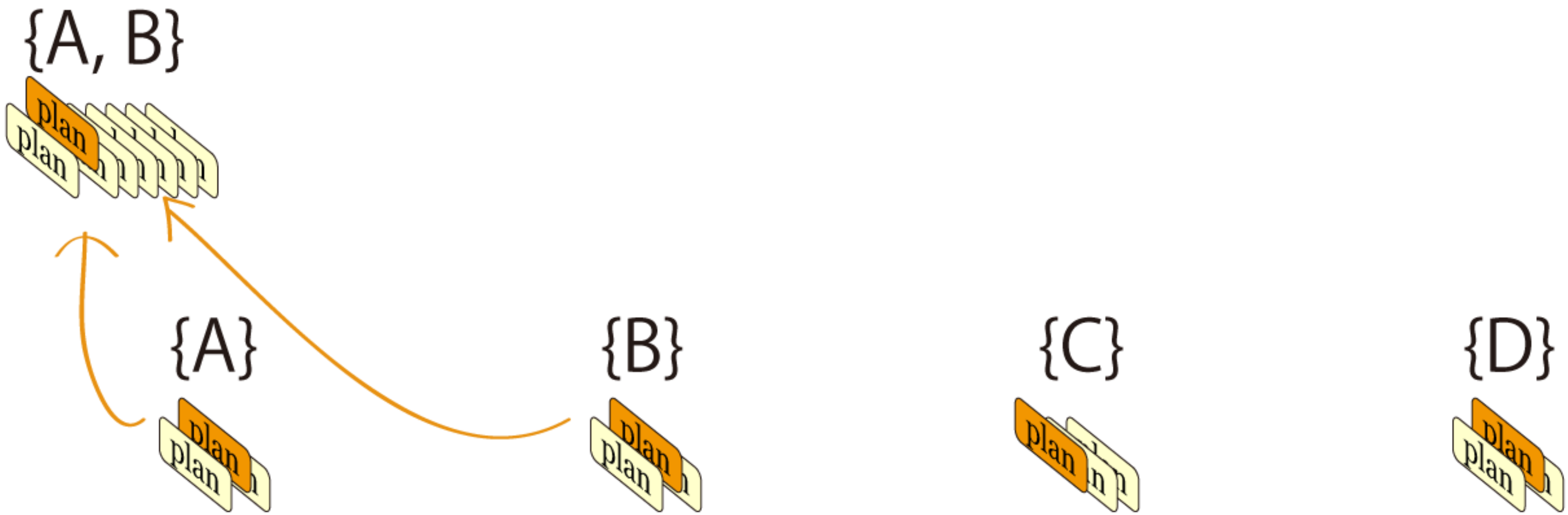
Bottom-up Plan Search

- Candidate plans for single relation
  - The cheapest access path

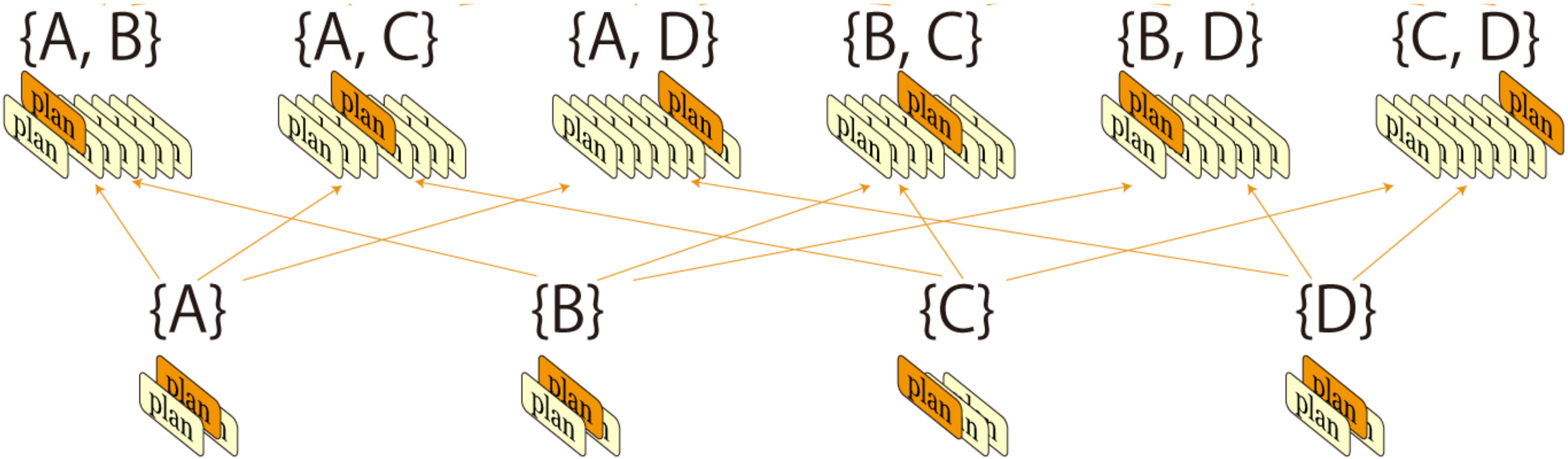
- N-relation join ordering search
  - Select the cheapest plans for each relation
  - Then, find optimal join orderings of every 2-relation join
  - Then, find optimal join orderings of every 3-relation join
    - … until N-relation

Ex) $A \bowtie B \bowtie C \bowtie D$

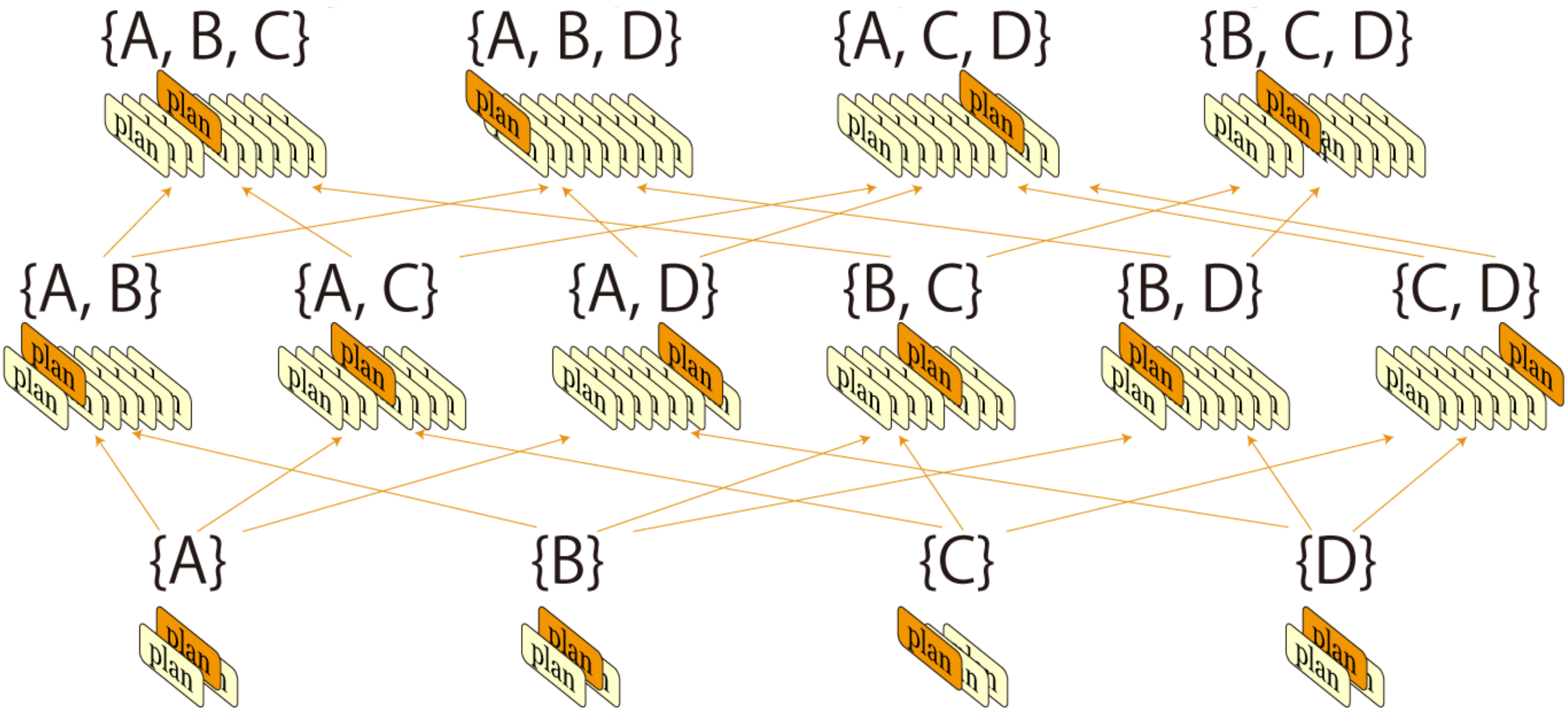
Ex) $A \bowtie B \bowtie C \bowtie D$



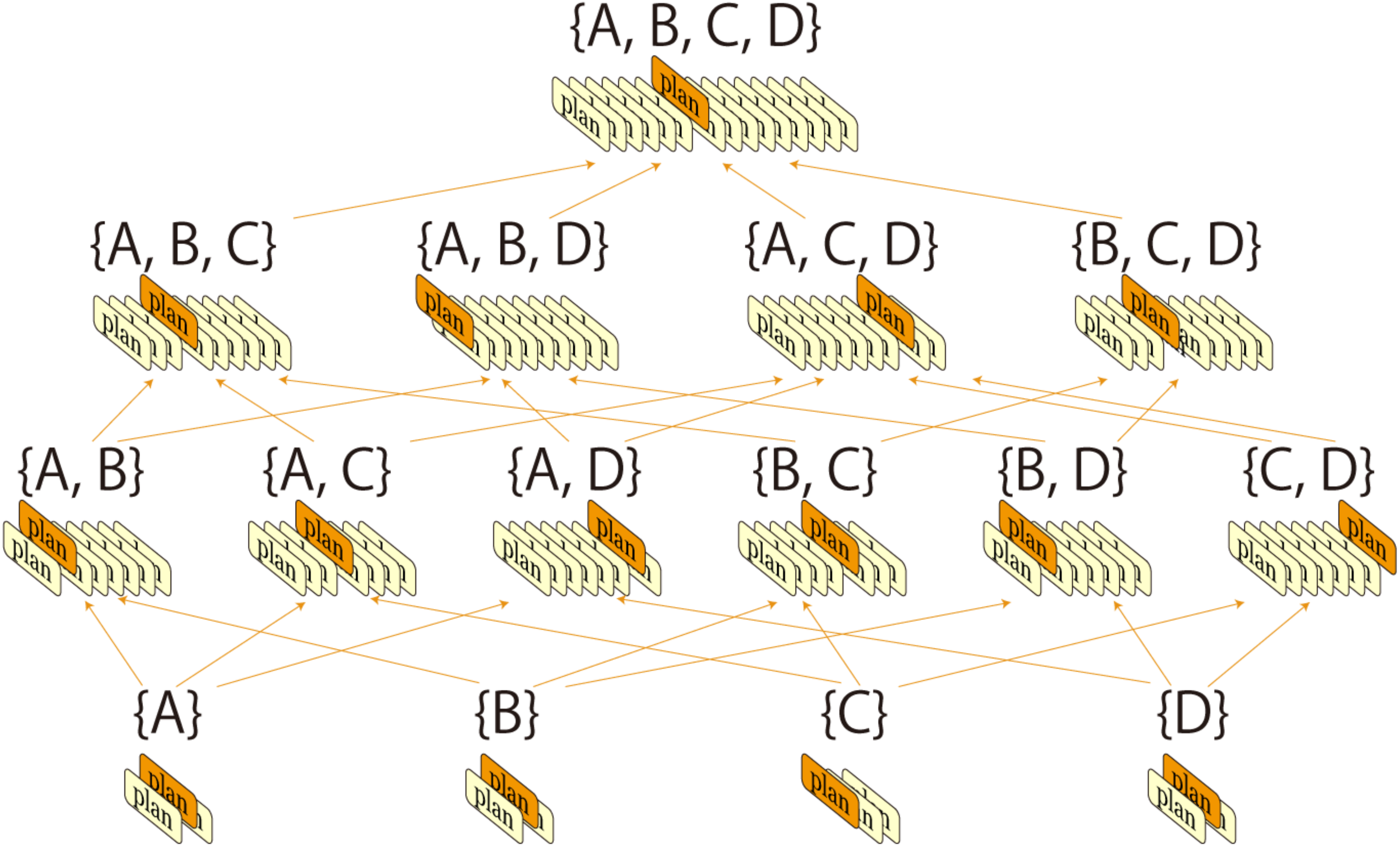
Ex) $A \bowtie B \bowtie C \bowtie D$



Ex) $A \bowtie B \bowtie C \bowtie D$



Ex) $A \bowtie B \bowtie C \bowtie D$



Volcano/Cascades (1993)

The Volcano Optimizer Generator: Extensibility and Efficient Search

Goetz Graefe
Portland State University
graefe@cs.pdx.edu

William J. McKenna
University of Colorado at Boulder
bill@cs.colorado.edu

Abstract

Emerging database application domains demand not only new functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. One of these tools is a new optimizer generator. Data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code. Compared with our earlier EX-

First, this new optimizer generator had to be usable both in the Volcano project with the existing query execution software as well as in other projects as a stand-alone tool. Second, the new system had to be more efficient, both in optimization time and in memory consumption for the search. Third, it had to provide effective, efficient, and extensible support for physical properties such as sort order and compression status. Fourth, it had to permit use of heuristics and data model semantics to guide the search and to prune futile parts of the search space. Finally, it

- Top-down transformational plan search
  - Yet another optimization approach
  - Not well known as “System-R style”, but widely used in practice
    Ex) SQL Server, Apache Hive (Apache Calcite), Greenplum Orca
- Extensible optimization framework

Extensible Optimization Framework

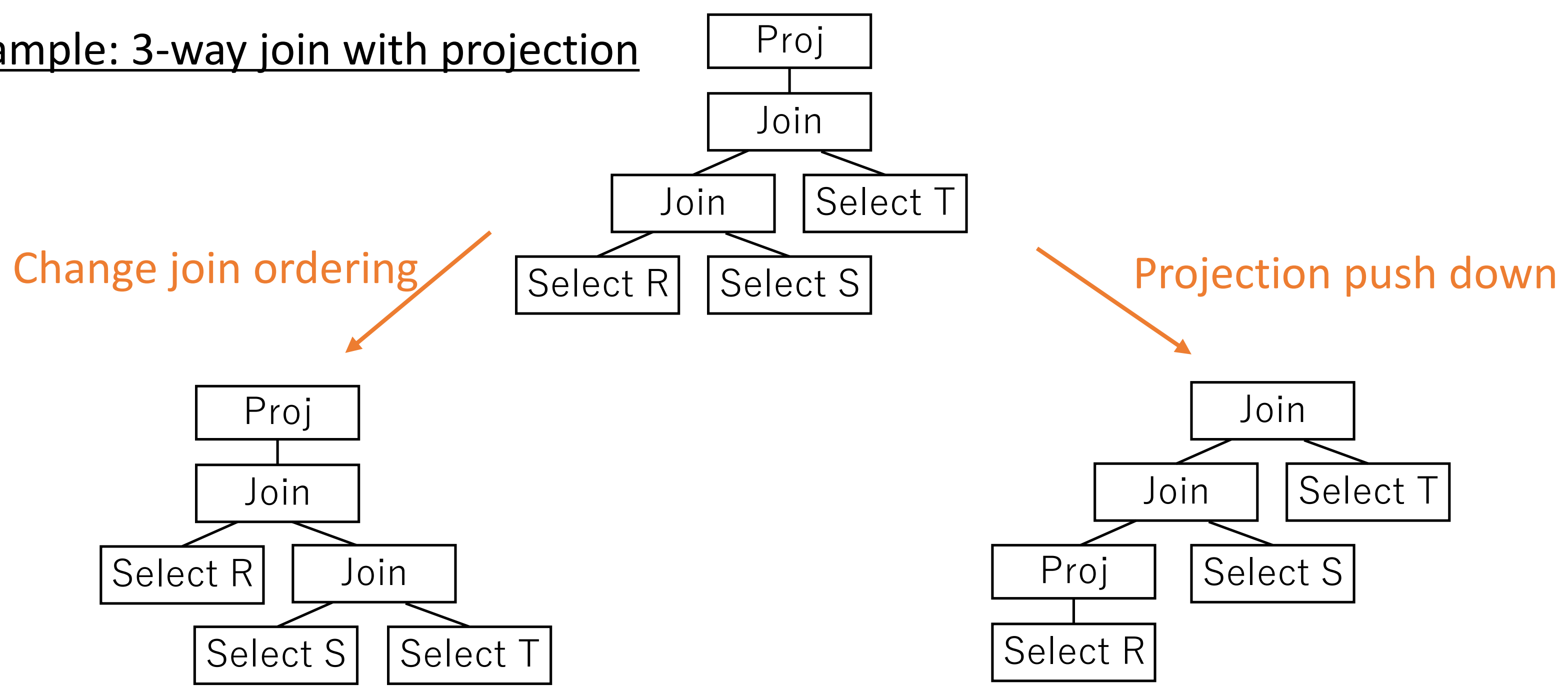
Query Optimizer Generator

- Generalized expression of query plan not limited to relational data model
- Users (optimizer developers) defines actual implementations:
  - Logical operator … corresponds to relational algebra
  - Physical algorithm … corresponds to scan & join methods such as sequential scan, index scan, hash join, nested loop join

Top-down Transformational Search

- Starts from an initial "logical plan"
- Generate alternative plans with:
  A) Logical operator transformation
  B) Physical algorithm selection
  C) Enforcing sorting order

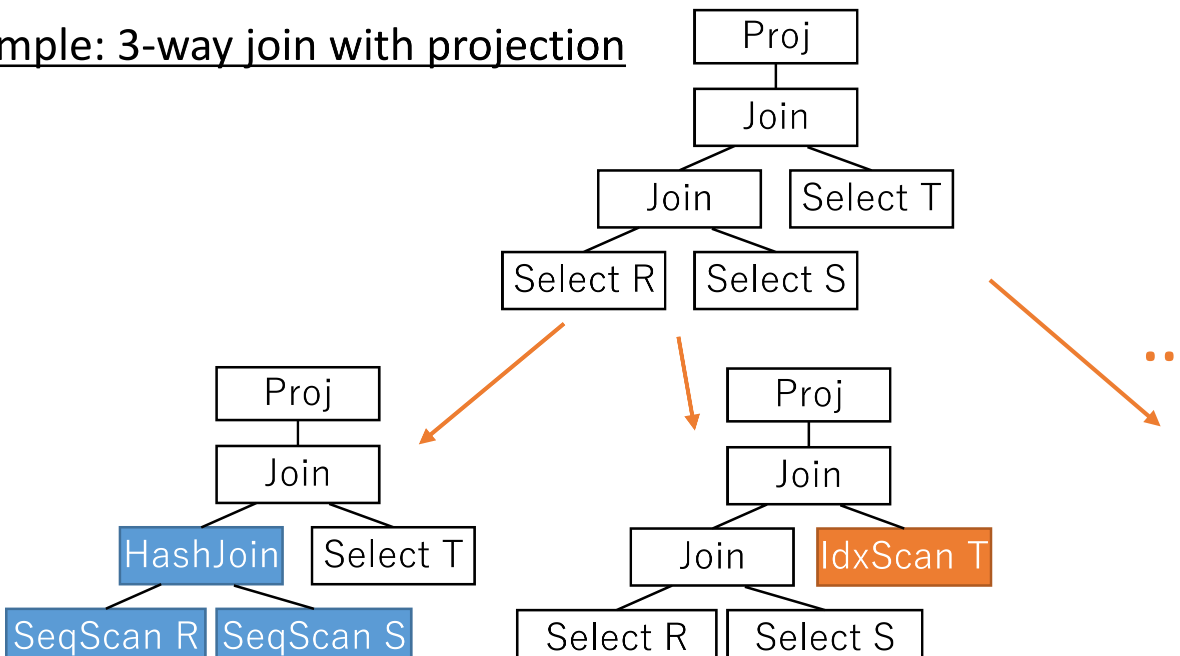
<u>Example: 3-way join with projection</u>



Top-down Transformational Search

- Starts from an initial "logical plan"
- Generate alternative plans with:
  - A) Logical operator transformation
  - B) Physical algorithm selection
  - C) Enforcing sorting order

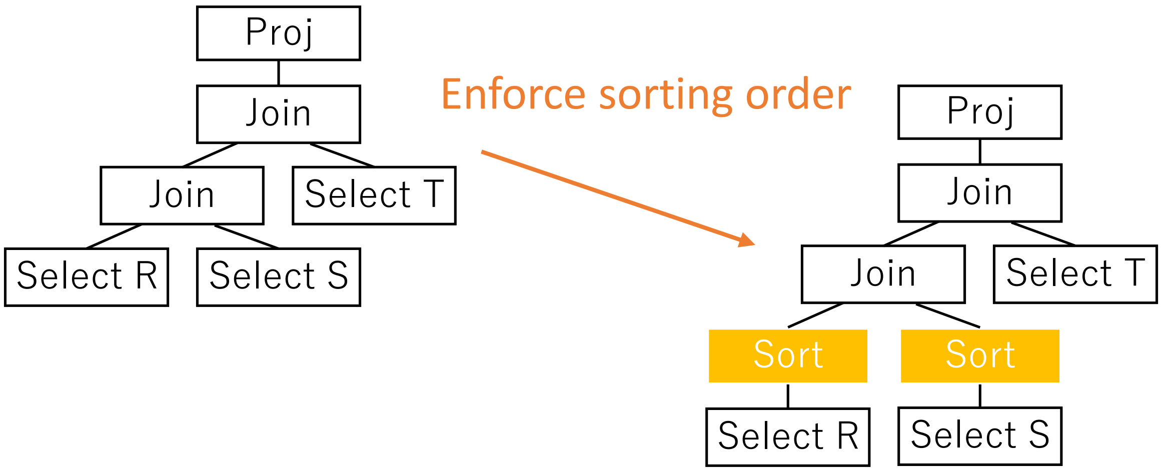
<u>Example: 3-way join with projection</u>



Top-down Transformational Search

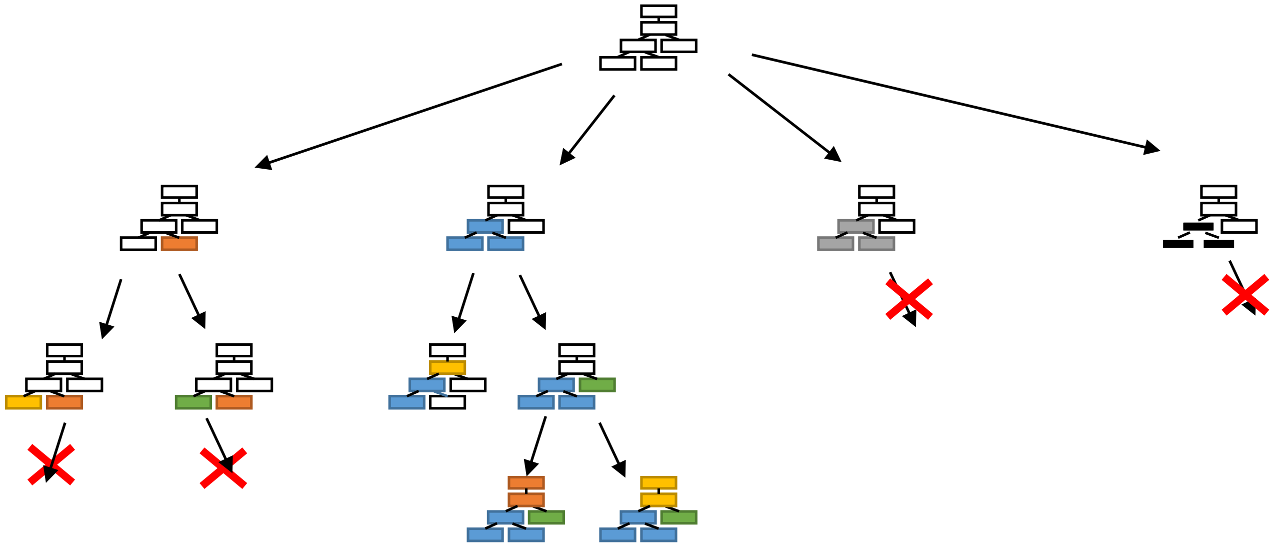
- Starts from an initial "logical plan"
- Generate alternative plans with:
  - A) Logical operator transformation
  - B) Physical algorithm selection
  - C) Enforcing sorting order

Example: 3-way join with projection



merge join of R and S is possible now

Benefits of Top-down approach



- Possible to intentionally limit search space
  - Effective pruning with branch-and-bound
  - Limit search space with search time deadline

Cost-based Optimization Basics

Two major cost-based optimization style

- System-R
  - Cost modeling with statistics
  - Bottom-up search
- Volcano/Cascades
  - Extensible optimizer generator
    - Cost estimation is user's responsibility
  - Top-down transformational search

Outline

- Introduction
- Theory: Query Optimization Framework
- Code: PostgreSQL Optimizer
- Theory: Cutting-Edge Technologies Overview
- Summary

PostgreSQL Optimizer

"System-R style" optimization

- Bottom-up plan search with dynamic programming
- CPU and I/O operation based cost modeling

$$\mathrm{Cost} = c_{\mathrm{seq}} n_{\mathrm{seq}} + c_{\mathrm{rand}} n_{\mathrm{rand}} + c_{\mathrm{tup}} n_{\mathrm{tup}} + \cdots$$

Seq. I/O *Random I/O* *CPU cost per tuple*

$$= \mathbb{C} \cdot \mathbb{N}$$

$\mathbb{C}$ Cost of single operation

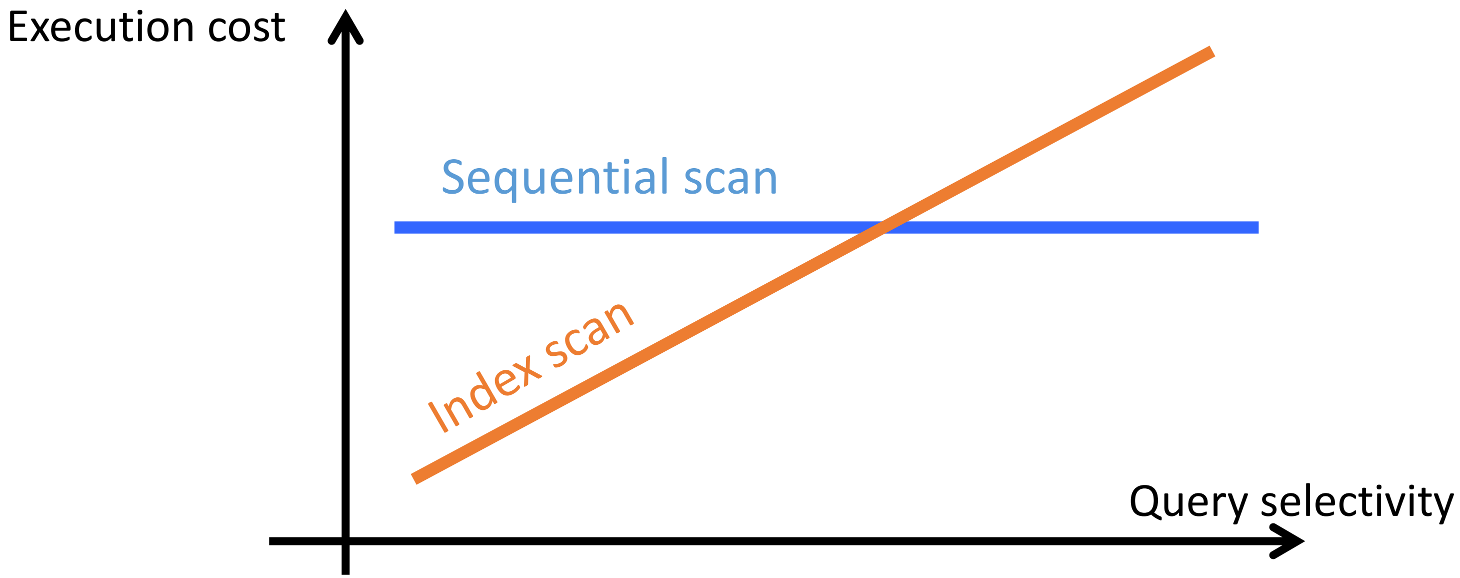
- seq_page_cost
- random_page_cost
- cpu_tuple_cost
- cpu_index_tuple_cost
- cpu_operator_cost
- (parallel_tuple_cost)

$\mathbb{N}$ Estimated number of each operation

- Cardinality estimation with statistics
- Cost formula for each plan type
  - SeqScan, IndexScan
  - NestLoopJoin, HashJoin, MergeJoin, ...

Detailed Look At Basic Scan Types

- Sequential scan
  - Efficient for accessing large potion of tables
- Index scan
  - Efficient for accessing a fraction of data



```
cost_seqscan()
@optimizer/path/costsize.c
```

$\mathbb{N}$ of **SeqScan**

n_{seq} = (# pages in a table)

n_{tup} = (# tuples in a table)

⋯ WHERE A AND B AND ⋯

n_{op} = #qual_operator

= (#tuples) × (weight factor of A)

+ (#tuples) × (weight factor of B)

+ ⋯

$\mathbb{N}$ of **IndexScan**

Consists of:

(A) CPU cost of searching B^+-tree

(B) CPU cost of scanning index tuples in leaf pages

(C) I/O cost of leaf pages

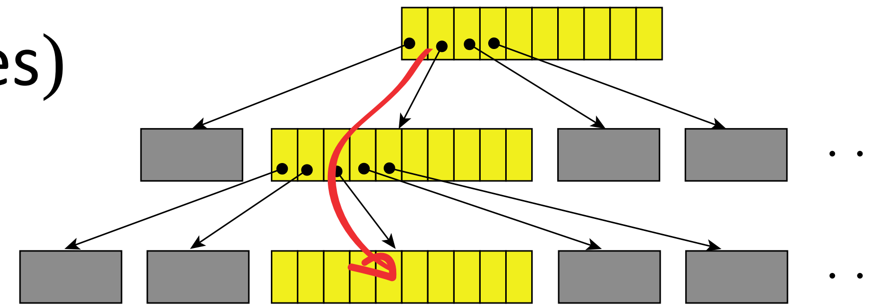
(D) I/O cost of heap pages

(E) CPU cost of scanning heap tuples

$\mathbb{N}$ of **IndexScan**

(A) B^+-tree search

n_{op} += $\log_2$(#index_tuples)



<ins>I/O cost of internal pages</ins>
Assumed to be always cached in the buffer

(B) Scanning index tuples in leaf pages

n_{itup} += #qual_operator
× #leaf_pages × #ituple_per_page × σ

<ins>Selectivity σ</ins>
Comes from statistics

$\mathbb{N}$ of **IndexScan**

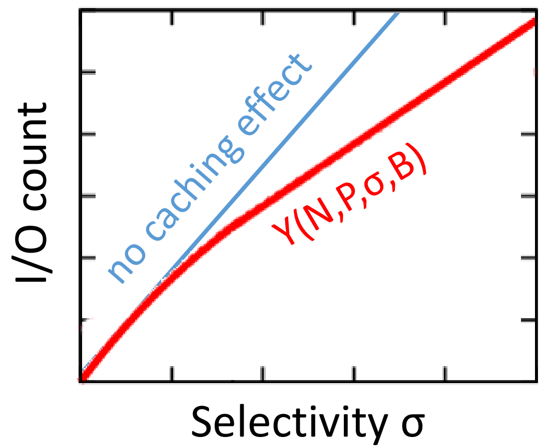
(C) I/O cost of index leaf pages

n_{rand} += Y(`effective_cache_size`, #leaf_pages)

<u>Mackert and Lohman function（Yao function）</u>

I/O count estimation with consideration of buffer caching

$$Y(N,P,\sigma,B) \equiv \begin{cases} \min\left(\frac{2PN\sigma}{2P+N\sigma}, P\right) & (P \le B) \\ \frac{2PN\sigma}{2P+N\sigma} & \left(P > B \wedge \sigma \le \frac{2PB}{N(2P-B)}\right) \\ B + \left(N\sigma - \frac{2PB}{2P-B}\right)\frac{P-B}{P} & \left(P > B \wedge \sigma > \frac{2PB}{N(2P-B)}\right) \end{cases}$$

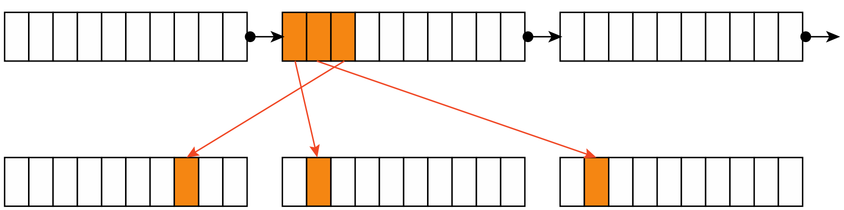


$\mathbb{N}$ of **IndexScan**

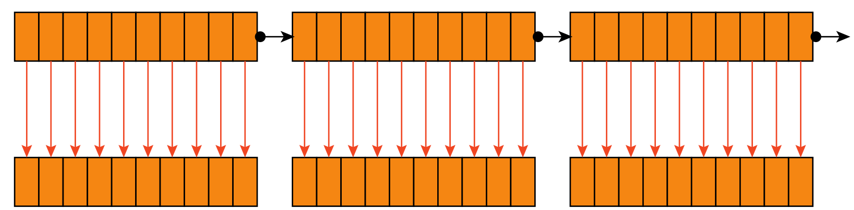
(D) I/O cost of heap pages

Correlation between index and heap ordering: α

<u>$\alpha = 0$: I/O pattern is random</u>



<u>$\alpha = 1$: I/O pattern is sequential</u>



$$n_{\mathrm{seq}} \mathrel{+}= \alpha^2 \times \text{\#match\_pages}$$

$$n_{\mathrm{rand}} \mathrel{+}= (1-\alpha^2) \times \text{\#match\_tuples}$$

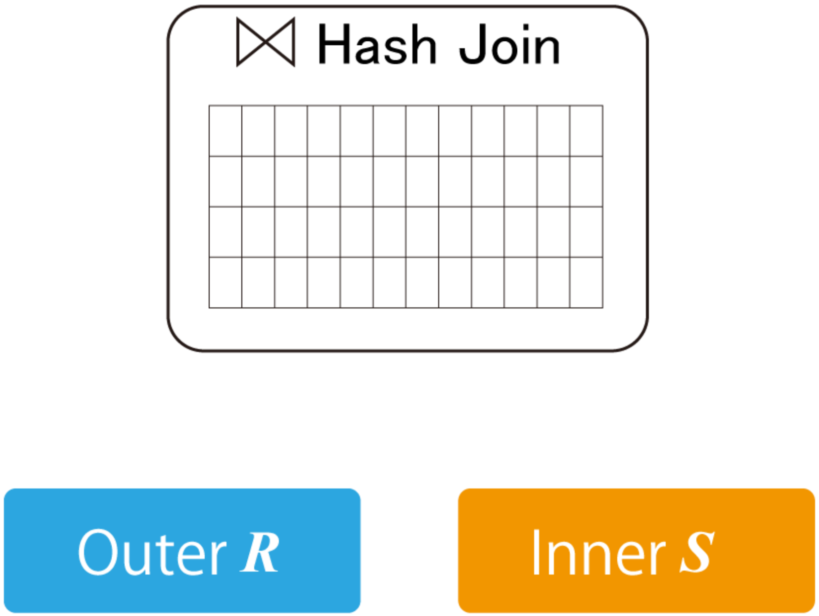
(E) CPU cost of scanning heap tuples

- Estimate the number of scanned tuples from $\boldsymbol{\sigma}$

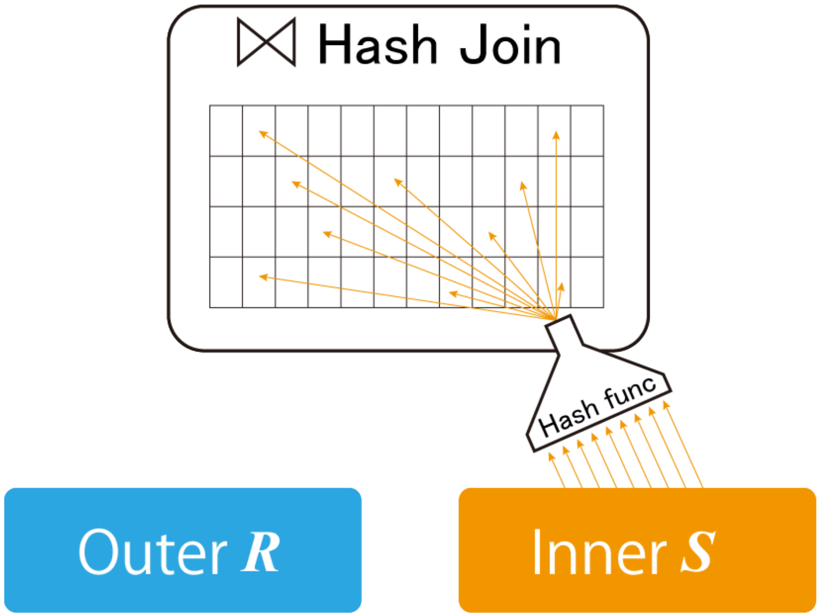
Detailed Look At Join Methods

- Hash join
  - Efficient for joining large number of records
  - Usually combined with sequential scans
- Nested Loop Join
  - Efficient for joining small number of records
  - Usually combined with index scans or small table sequential scans

ℕ of **HashJoin**



$\mathbb{N}$ of **`HashJoin`**



Build phase

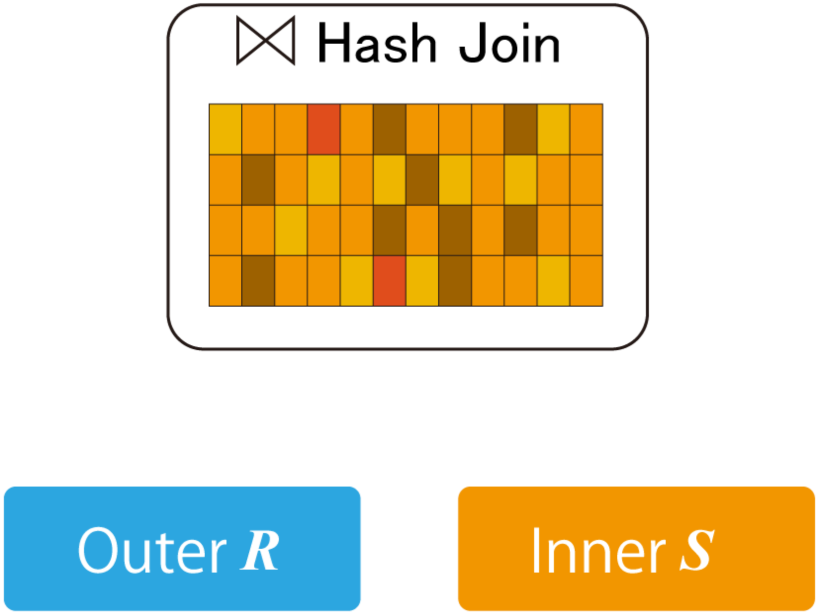
- Cost += Cost(*inner*)

  n_{op} += #qual_op × #inner_tuples

  n_{tup} += #inner_tuples

Hashing cost

$\mathbb{N}$ of **HashJoin**



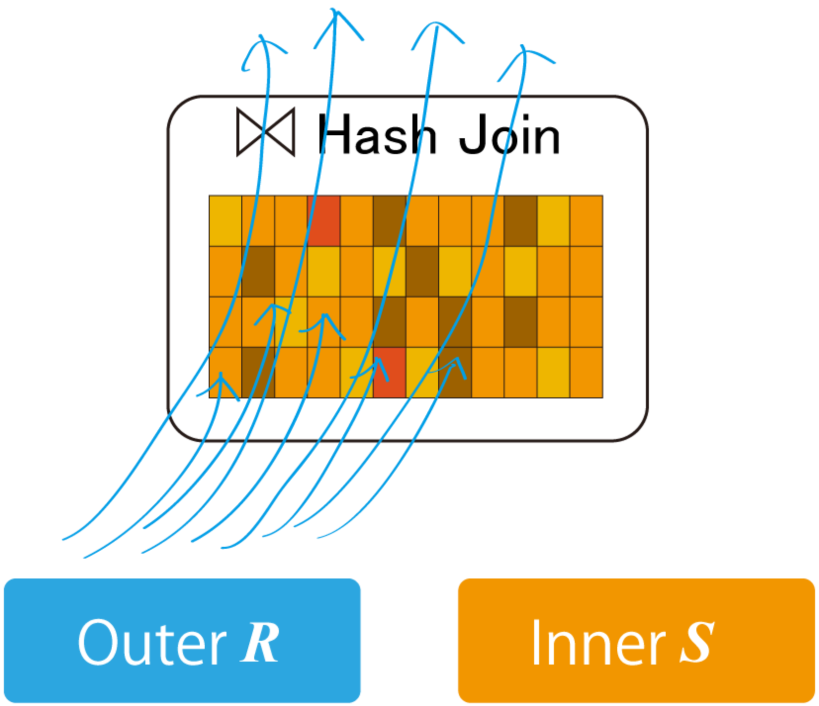
Build phase

- Cost += Cost(*inner*)

n_{op} += #qual_op × #inner_tuples

n_{tup} += #inner_tuples

$\mathbb{N}$ of **HashJoin**



Build phase

- $\text{Cost} \mathrel{+}= \text{Cost}(\textit{inner}) + \mathbb{C} \cdot \mathbb{N}$

  n_{op} += #qual_op × #inner_tuples

  n_{tup} += #inner_tuples

Probe phase

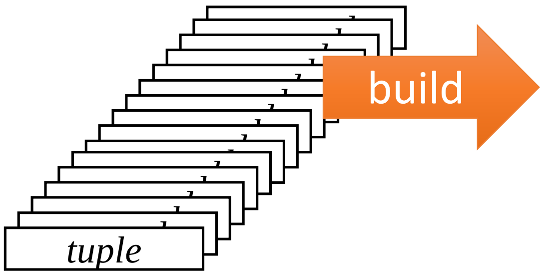
- $\text{Cost} \mathrel{+}= \text{Cost}(\textit{outer}) + \mathbb{C} \cdot \mathbb{N}$

n_{op} += #qual_op × (1 + **#bucket_size × 0.5)**
× #outer_tuples

Hashing & table lookup (bucket search) cost

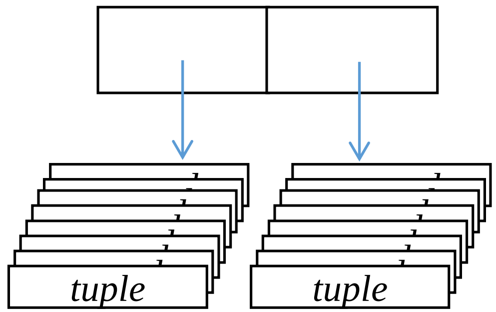
n_{tup} += #match_tuples

ℕ of **HashJoin**



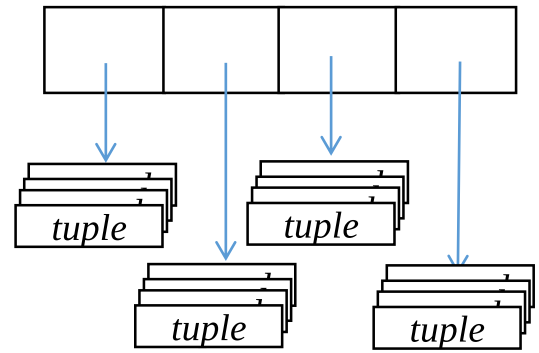
16 records

#buckets: 2



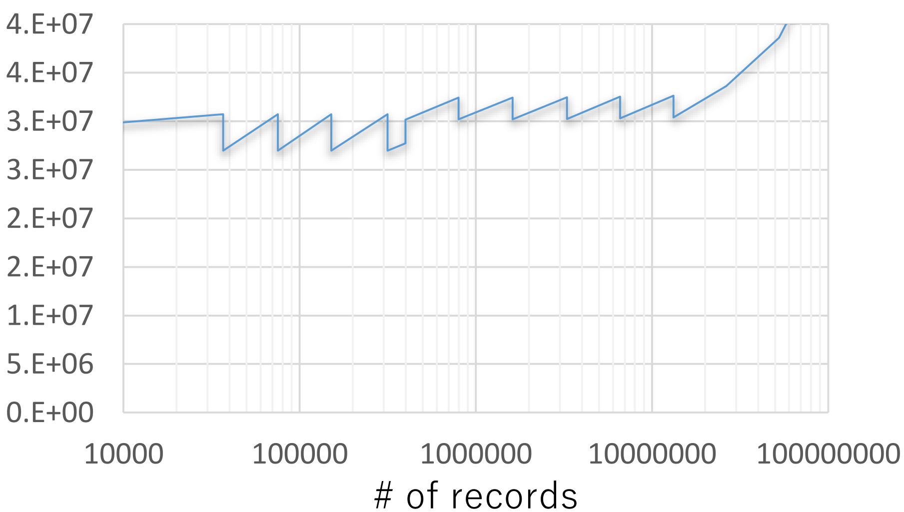
4 tuples are compared for lookup in average

#buckets : 4



2 tuples are compared for lookup in average

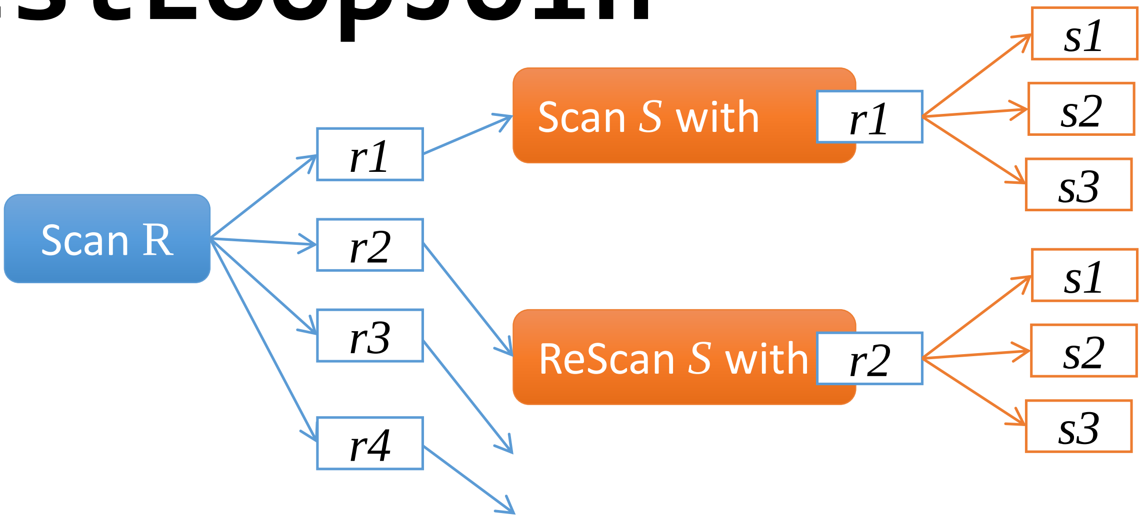
Estimated cost of 2-way HashJoin



ℕ of **NestLoopJoin**

$R \bowtie S$

outer *inner*



$\mathbb{N}$ of **NestLoopJoin**

$R \bowtie S$

outer *inner*

- When #outer_tuples = 1

$$\text{Cost} = \text{Cost}(\textit{outer}) + \text{Cost}(\textit{inner}) + \mathbb{C} \cdot \mathbb{N}$$

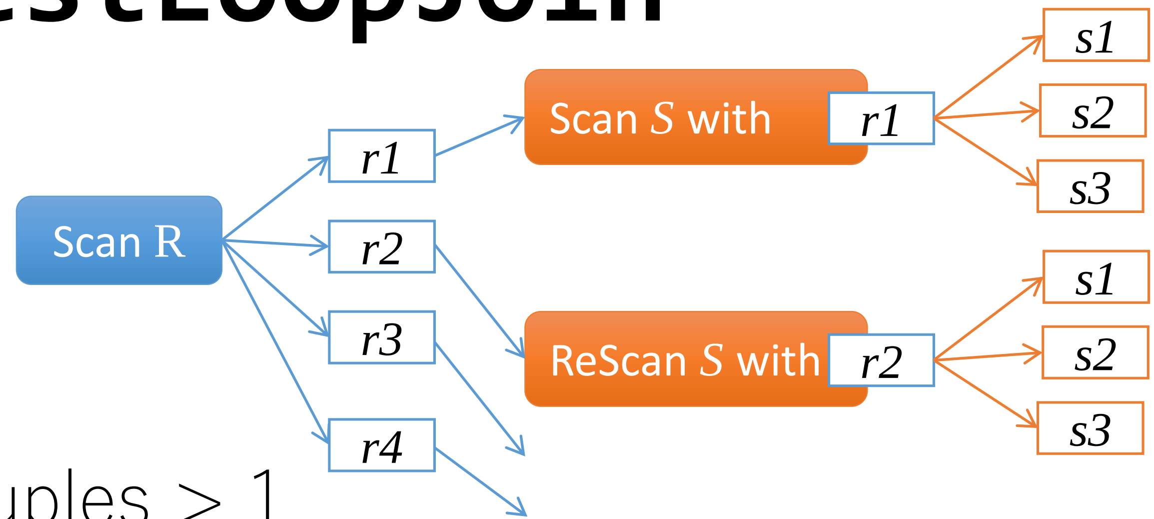
n_{tup} += #inner_tuples

n_{op} += #qual_operator × #inner_tuples

$\mathbb{N}$ of **NestLoopJoin**

$R \bowtie S$

outer *inner*



- When #outer_tuples > 1

$$\mathrm{Cost} = \mathrm{Cost}(\mathit{outer}) + \mathrm{Cost}(\mathit{inner}) + \mathbb{C} \cdot \mathbb{N}$$
$$+ \ (\#\text{outer\_tuples} - 1) \ \times \ \mathrm{Cost}(\textbf{ReScan}\ \text{inner})$$

Higher buffer hit ratio in ReScan
→ Cost of ReScan is lower than cost of IndexScan

n_{tup} += #inner_tuples × #outer_tuples

n_{op} += #qual_operator × #inner_tuples × #outer_tuple

See How It Works

- TPC-H Benchmark
  - Specification and tools for benchmarking data warehouse workload
    - Open source implementation: DBT-3, pg_tpch
  - Schema, data generation rules and queries

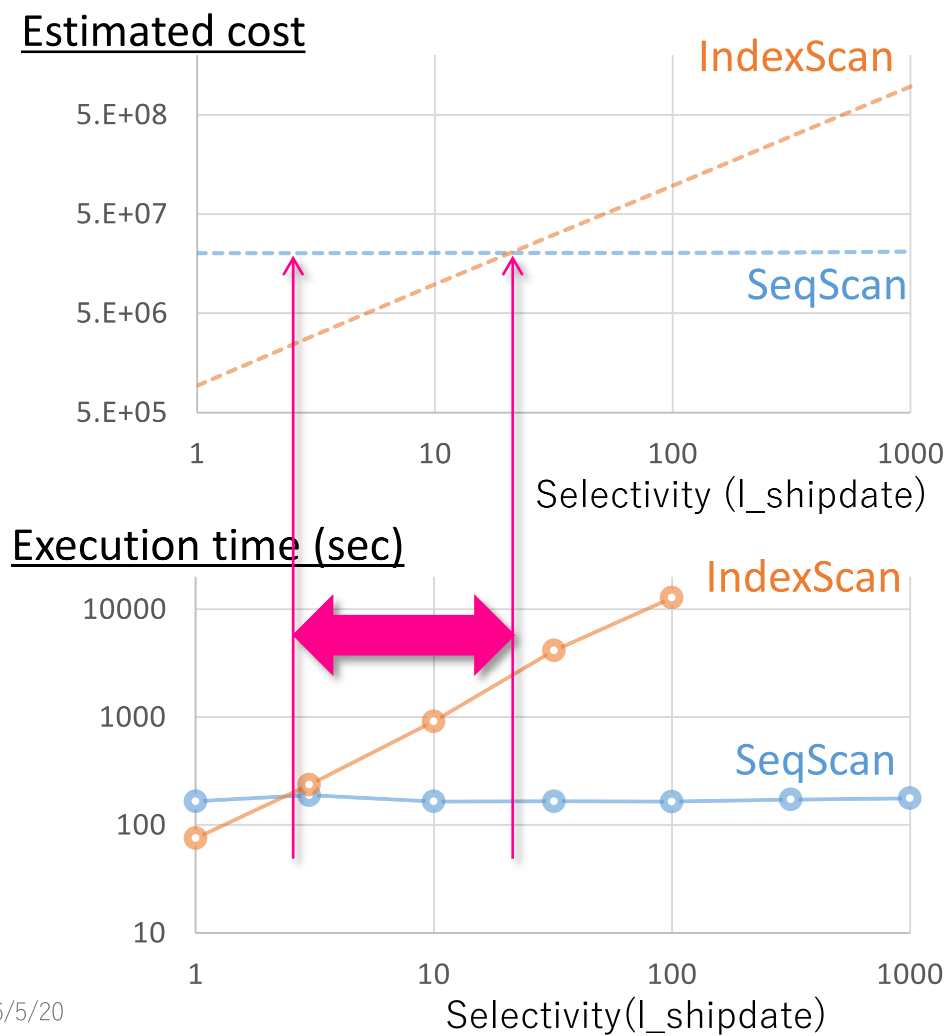
- Experiments with 100GB
  - Scale Factor = 100

Experimental Setup

- Dell R720xd
  - Xeon (2sockets, 16cores)
  - x24 NL-SAS HDD
- With PostgreSQL 9.5
  - Default cost parameter settings
  - `SeqScan` & `HashJoin`
    - `enable_seqscan = on, enable_hashjoin = on` and disables other methods
  - `IndexScan` & `NestLoopJoin`
    - `enable_indexscan = on, enable_nestloop = on` and disables other methods

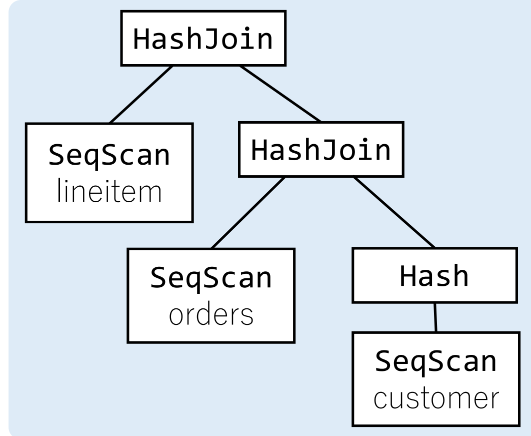
TPC-H Q.1: The Simplest Case

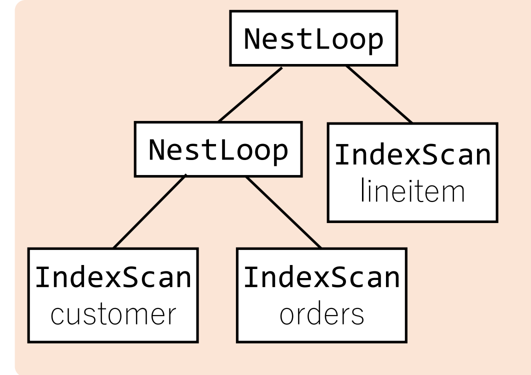
```
SELECT count(*), ... FROM lineitem
  WHERE l_shipdate BETWEEN [X] AND [Y]
```



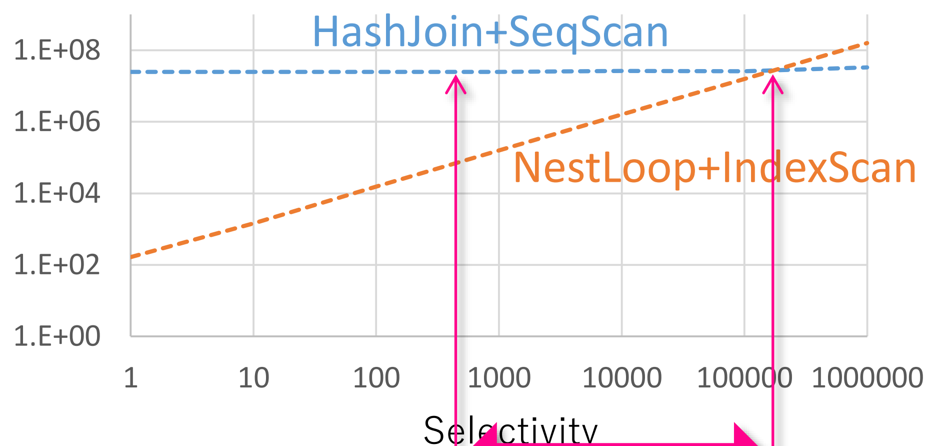
- Good trend estimation for each method
- Estimated break-event point is errorneus
  - IndexScan should be more expensive (need parameter calibration)

TPC-H Q.3



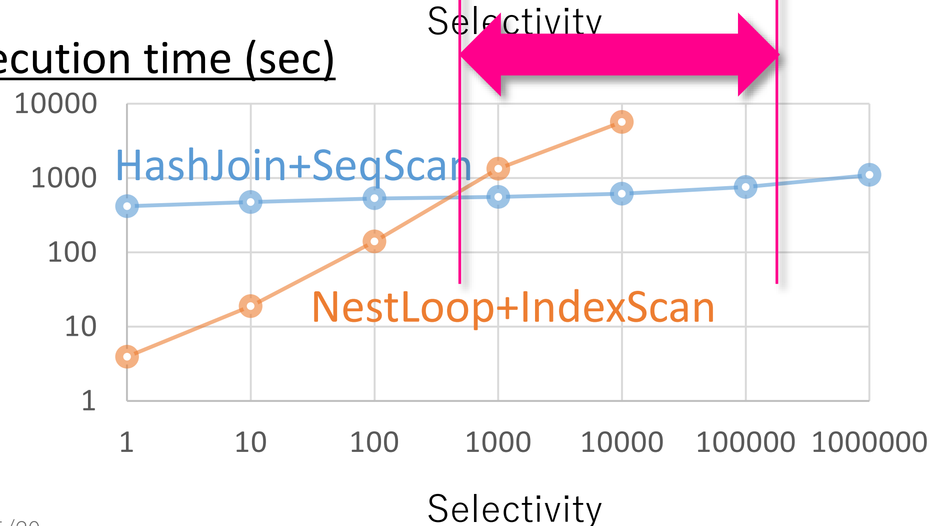


Estimated cost




```
SELECT count(*), ...
  FROM customer, orders, lineitem
  WHERE c_custkey = o_custkey AND
        o_orderkey = l_orderkey AND
        c_custkey < [X] AND
        c_mktsegment = ‘MACHINERY’;
```

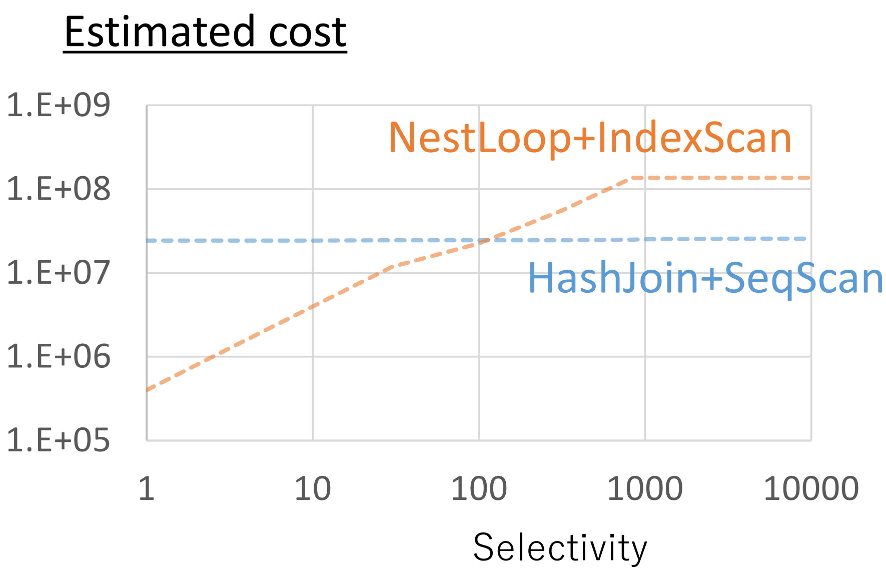
Execution time (sec)

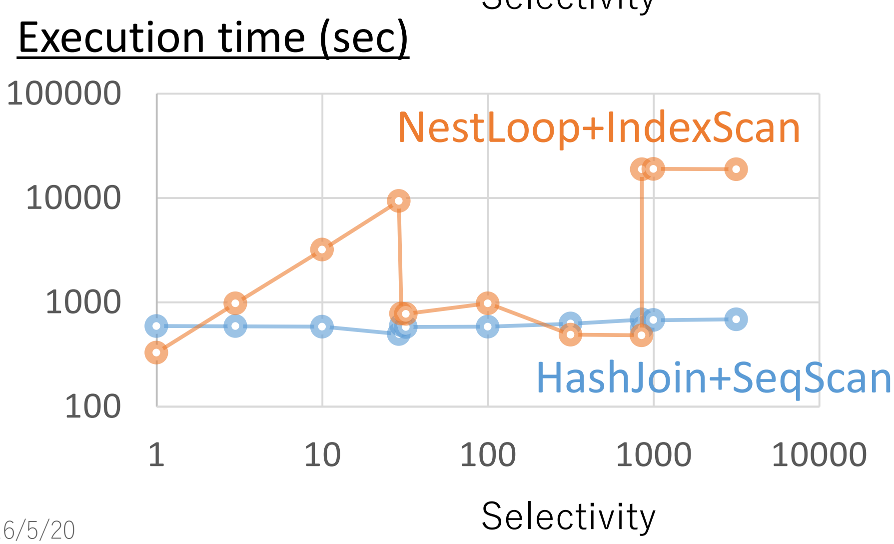


Similar result as in Q.1

- Good trend estimation for each
- Erroneous break-event point without parameter calibration

More Complex Case
TPC-H Q.4: Semi-Join Query



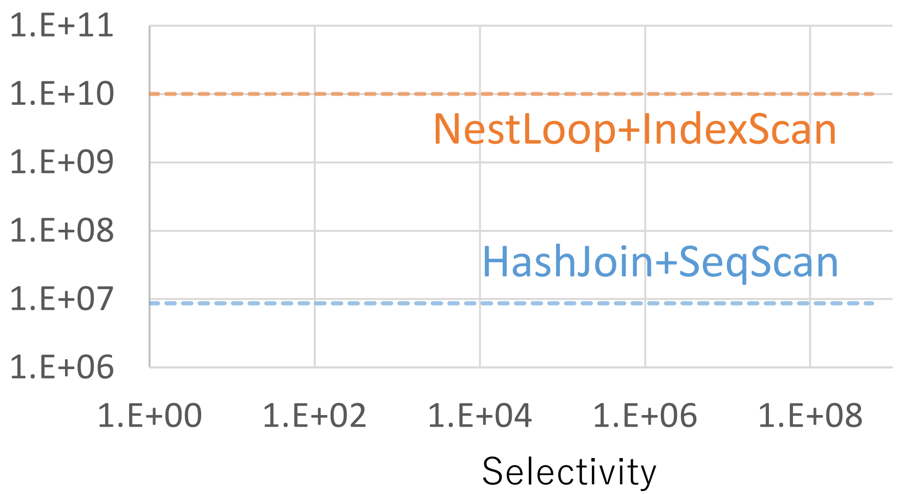



```
SELECT count(*), ...
  FROM orders
  WHERE
    o_orderdate >= '1995-01-01' AND
    o_orderdate < '1995-01-01'
                   + interval '3 month' AND
    EXISTS(
      SELECT * FROM lineitem
       WHERE l_orderkey = o_orderkey
         AND l_commitdate < l_receiptdate)
```

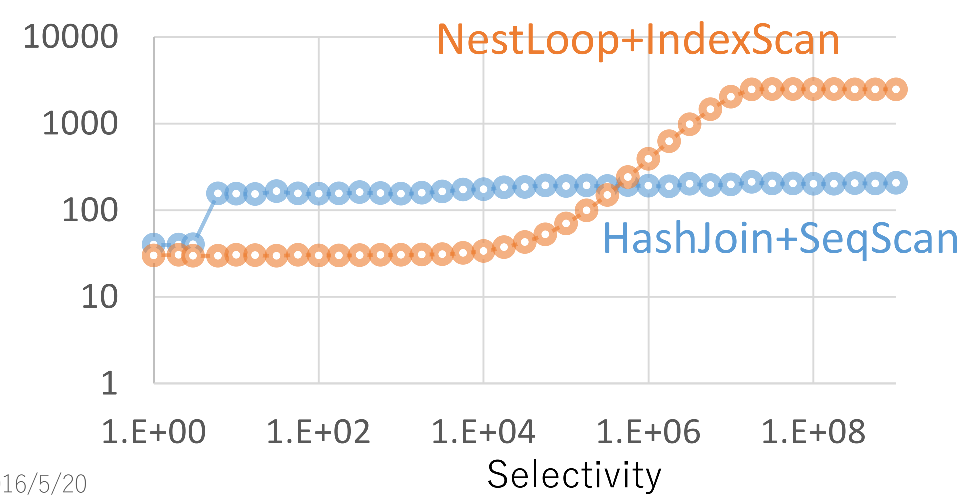
- Plan selection for semi-join tend to be unstable

More Complex Case
TPC-H Q.22: Anti-Join Query

Estimated cost



Execution time (sec)




```
SELECT count(*), ...
  FROM supplier, lineitem l1, orders, nation
  WHERE s_suppkey = l1.l_suppkey AND
        o_orderkey = l1.l_orderkey AND
        o_orderstatus = 'F' AND
        l1.l_receiptdate > l1.l_commitdate AND
    EXISTS (
      SELECT * FROM lineitem l2
       WHERE l2.l_orderkey = l1.l_orderkey
         AND l2.l_suppkey <> l1.l_suppkey)
    AND NOT EXIST (
      SELECT * FROM lineitem l3
       WHERE l3.l_orderkey = l1.l_orderkey
         AND l3.l_suppkey <> l1.l_suppkey
         AND l3.l_receiptdate > l3.l_commitdate)
    AND s_nationkey = n_nationkey
    AND n_name = ‘JAPAN'
```

- Difficulties in overall cost trend estimation

Summary: PostgreSQL Optimizer

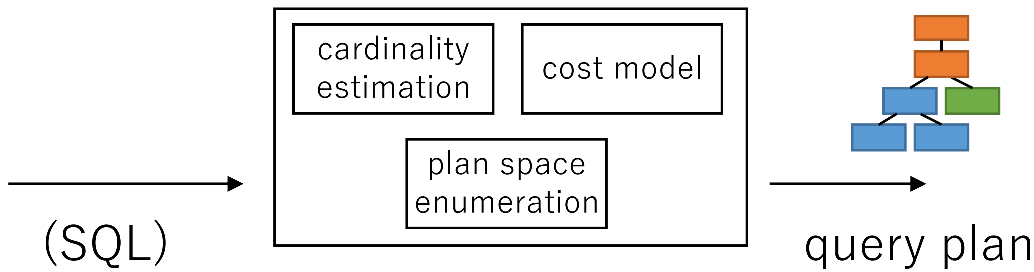
- Detailed look at cost modeling of basic methods
  - SeqScan, IndexScan
  - HashJoin, NestedLoopJoin
- Observation with TPC-H benchmark
  - Good cost trend estimation for simple join queries
    - Erroneous cheapest plan selection without parameter tuning
  - Difficulties with semi-join and anti-join queries

Outline

- Introduction
- Theory: Query Optimization Framework
- Code: PostgreSQL Optimizer
- Theory: Cutting-Edge Technologies Overview
- Summary

Cutting-Edge Technologies

- Traditional optimization was a “closed” problem



- “Rethink the contract” — *Surajit Chaudhuri*
    - Feedback from previous execution
    - Dynamic integration with execution

Mid-query Re-optimization

[N. Kabra et.al., SIGMOD'98]

- Detects sub-optimality of executing query plan
  - Query plans are annotated for later estimation improvement
  - Runtime statistics collection
    - Statistics collector probes are inserted into operators of executing query plan
- Plan modification strategy
  - Discard current execution and re-optimize whole plan
  - Re-optimizer only subtree of the plan that are not started yet
  - Save partial execution result and generate new SQL using the result

Plan Bouquet

[A. Dutt et.al., SIGMOD'14]

- Generate a set of plans for each selectivity range
- Estimation improvement with runtime statistics collection

- Evaluation with PostgreSQL

Bounding Impact of Estimation Error

[T. Neumann et.al., BTW Conf '13]

- "Uncertainty" analysis of cost estimation
  - Optimality sensitivity to estimation error
- Execute partially to reduce uncertainty

Outline

- Introduction
- Theory: Query Optimization Framework
- Code: PostgreSQL Optimizer
- Theory: Cutting-Edge Technologies Overview
- Summary

Summary

- Cost-based optimization framework
  - System-R style bottom-up optimization
  - Volcano style top-down optimization
- Detailed look at PostgreSQL optimizer
  - Cost modeling of basic scan and join method
  - Experiment with TPC-H benchmark
- Brief overview of cutting-edge technologies