

CMSC 724: Database Management Systems

Query Processing and Optimization

Instructor: Amol Deshpande
amol@cs.umd.edu

Basics of Query Processing

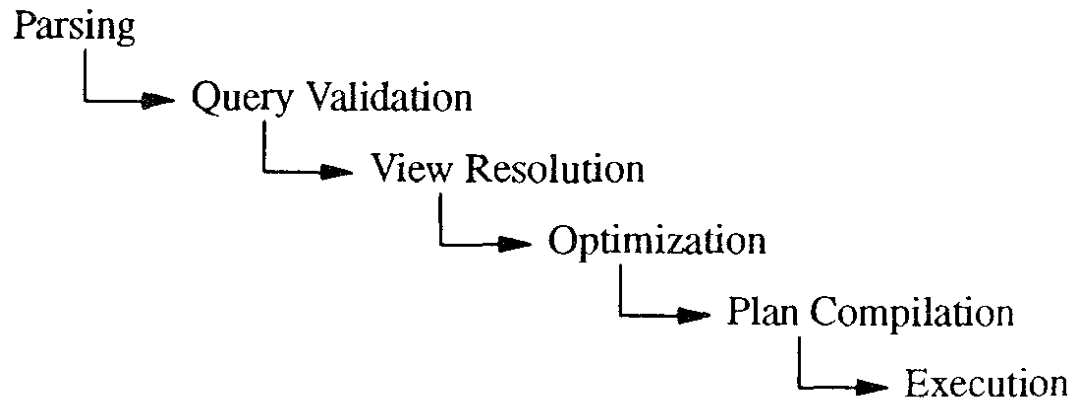


Figure 2. Query processing steps.

Update queries usually handled through “deferred updates” (use standard read-only techniques to identify the modifications, and apply them afterwards).

Architectural Issues

- ▶ Logical algebra vs physical algebra
 - Latter is system-specific, and refers to the specific implementations of operators
 - Mapping from logical to physical operators is often not one-to-one
 - Most operator implementations usually handle subsequent selects and projects
 - A single logical operator may be broken up into multiple physical ones (e.g., “sort” is done separately from “merge” for “sort-merge join”)
 - A “symmetric” logical operator may be implemented by an “asymmetric” physical operator

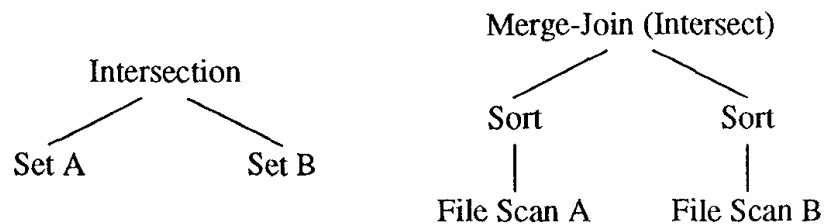


Figure 3. Logical and physical algebra expressions.

How to pass tuples between operators

- ▶ Materialization: Write out the results to a file, and the next operator reads it from the file
- ▶ Pipelining: Have both (or more) operators running at the same time (e.g., in different threads or processes), and use queues to transfer tuples
 - Hard to make this work efficiently (e.g., OS may switch to an operator that has no inputs, leading to wasted context switches)
- ▶ Iterator model: Have operators “schedule” each other
 - When an operator needs more inputs, it “calls” the child operator(s)
 - No IPC needed – these are function calls
 - For Query Processing, can separate the work of an operator into:
 - initialization (`init()`)
 - produce the next tuple (`next()`)
 - clean up (`close()`)
 - Main drawback (as we discuss later): too many function calls for modern architectures

Types of Query Plans

- ▶ In some older papers, left-deep and right-deep are switched
 - Think of “left” as “outer” and “right” as “inner”
 - “Right-deep plans have only recently received more interest and may actually turn out to be very efficient, in particular in systems with ample memories” – refers to the ability to build many hash indexes at once, and today makes sense for “left-deep” plans

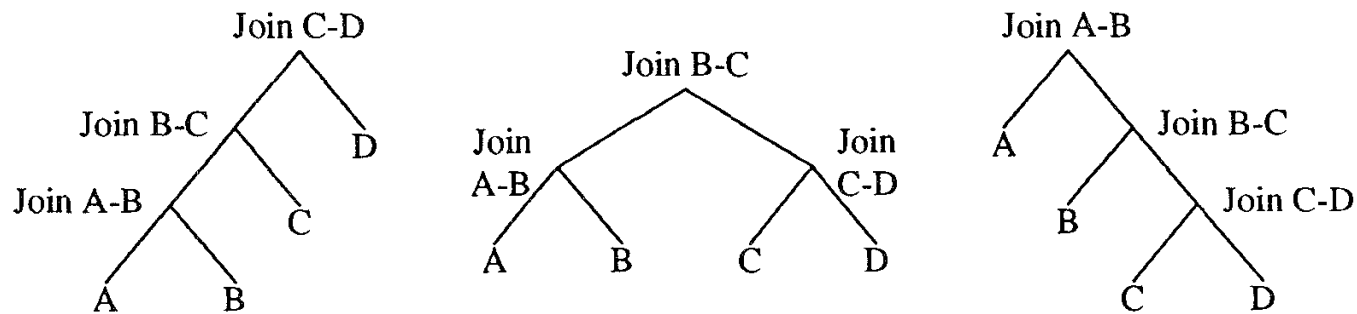


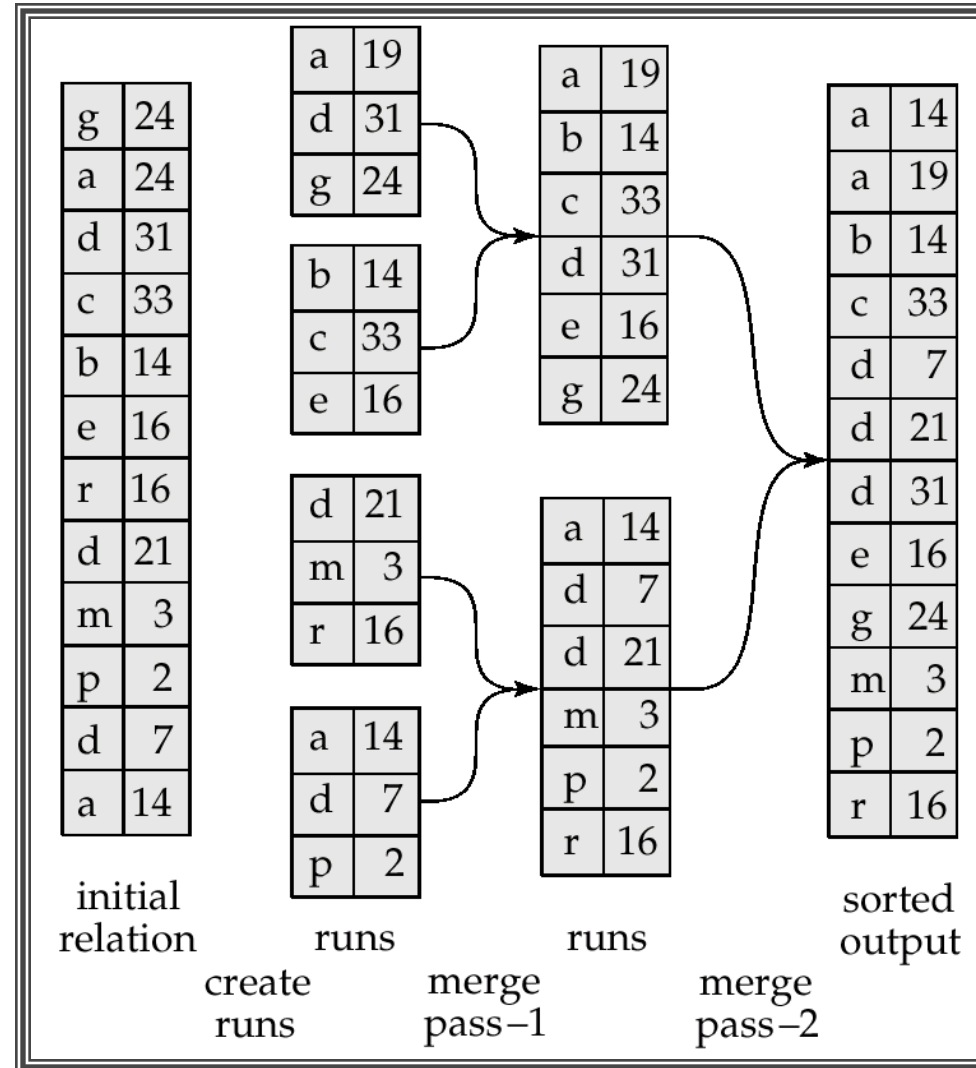
Figure 4. Left-deep, bushy, and right-deep plans.

- ▶ In general, may be a DAG (directed acyclic graph)
 - In case of common subexpressions

Sorting

▶ Volcano implementation:

- open() does most of the work
 - If the input fits in memory, reads the entire input and does a quick-sort
 - If it doesn't fit in memory, uses external merge-sort except for the last merge
- next() simply produces the tuples in the first case, and actually does the last merge in the second case
- Probably better to do all the work in "next()" (with special-case code for the first call)



Sorting: Creating the initial runs

- ▶ Say main memory = M blocks (of b tuples each)
- ▶ Option 1: Read M blocks at a time, quick-sort, and write out the “sorted run” to disk
 - Generates runs of size M
- ▶ Option 2: Replacement selection
 - Read $M \cdot b$ tuples in memory, and keep it (always) in sorted order
 - Write out the first tuple to disk as the first sorted run
 - Say the largest value written out so far is 1000
 - Read the next tuple from the original relation
 - If > 1000 , add it to the same sorted run, and output the next tuple from that
 - If not, start (or add to) a second sorted run in memory
 - Keep doing this until the you the first sorted run in memory finishes is done (i.e., all new tuples get added to the second run)
 - Can use the Heap data structure to do this efficiently

Replacement Selection Example

Input

33, 18, 24, 58, 14, 17, 7, 21, 67, 12, 5, 47, 16

└─ Front of input string

(Heap sort!)

Remaining input	Memory(P=3)	Output run(A)
33, 18, 24, 58, 14, 17, 7, 21, 67, 12	5 47 16	-
33, 18, 24, 58, 14, 17, 7, 21, 67	12 47 16	5
33, 18, 24, 58, 14, 17, 7, 21	67 47 16	12, 5
33, 18, 24, 58, 14, 17, 7	67 47 21	16, 12, 5
33, 18, 24, 58, 14, 17	67 47 (7)	21, 16, 12, 5
33, 18, 24, 58, 14	67 (17) (7)	47, 21, 16, 12, 5

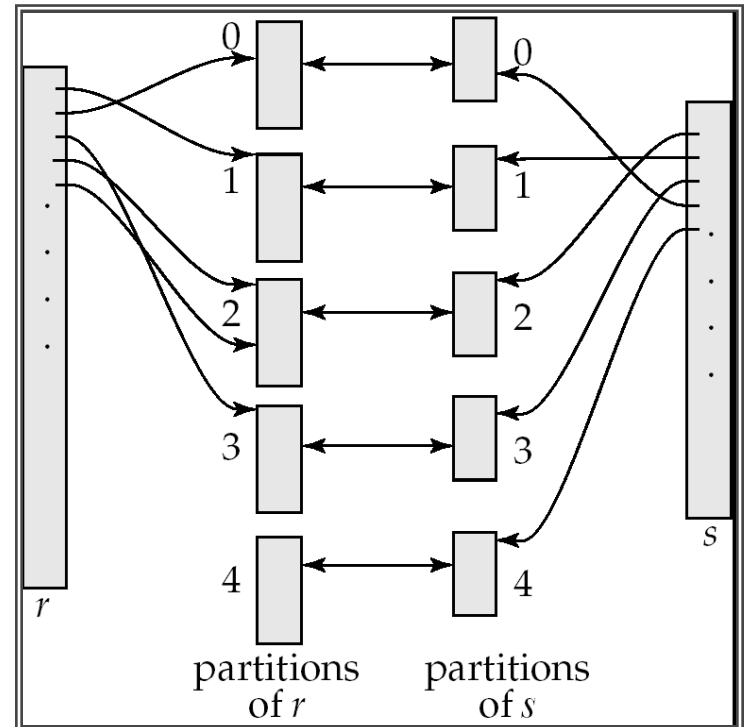


Replacement Selection


- ▶ Need a data structure that efficiently supports removal of the smallest entry
 - The “heap” data structure works well
- ▶ Replacement selection results in larger runs → more efficient merge
 - If the input is already sorted or almost sorted, there is only one run
 - For random inputs, the runs are of size $2M$
- ▶ But RS has more complex I/O patterns and there are other complications
 - Need to balance against the benefits of having fewer runs

Hashing

- ▶ Usually better when “equality matching” is required
- ▶ Basic idea:
 - “Build” a hash table on one of the inputs on the equality attribute(s)
 - “Probe” using the second input in any order
- ▶ What if the smaller input is too large?
 - Partition both the inputs using some criteria on the equality attribute (could be another hash function, or a range function)
 - Do partition-by-partition join





Hashing

- ▶ Usually better when “equality matching” is required
 - ▶ Basic idea:
 - “Build” a hash table on one of the inputs on the equality attribute(s)
 - “Probe” using the second input in any order
 - ▶ What if the smaller input is too large?
 - Partition both the inputs using some criteria on the equality attribute (could be another hash function, or a range function)
 - Do partition-by-partition join
 - ▶ May need to do this “recursively”
 - Very unlikely to happen with today’s large memories
 - ▶ Hybrid hash join
 - Keep one of the partitions in memory when doing the initial partitioning
 - Can be done in a reactive fashion
 - Works very well when the smaller input is just larger than memory
- 

Sorting vs Hashing

- ▶ Most operators can be implemented using sorting or hashing
- ▶ Many papers written on which one is better
 - Depends a lot on the specific computing architecture
- ▶ Lot of recent work on multi-core sorting and hashing, and in shared-nothing settings

 SORT VS. HASH REVISITED: FAST JOIN IMPLEMENTATION ON MODERN MULTI-CORE CPUs
VLDB 2009



→ Hashing is faster than Sort-Merge.
→ Sort-Merge is faster w/ wider SIMD.

 DESIGN AND EVALUATION OF MAIN MEMORY HASH JOIN ALGORITHMS FOR MULTI-CORE CPUs
SIGMOD 2011



→ Trade-offs between partitioning & non-partitioning Hash-Join.

 MASSIVELY PARALLEL SORT-MERGE JOINS IN MAIN MEMORY MULTI-CORE DATABASE SYSTEMS
VLDB 2012





→ Sort-Merge is already faster than Hashing, even without SIMD.

 MASSIVELY PARALLEL NUMA-AWARE HASH JOINS
IMDM 2013





→ Ignore what we said last year.
→ You really want to use Hashing!

 MAIN-MEMORY HASH JOINS ON MULTI-CORE CPUs: TUNING TO THE UNDERLYING HARDWARE
ICDE 2013



→ New optimizations and results for Radix Hash Join.

 AN EXPERIMENTAL COMPARISON OF THIRTEEN RELATIONAL EQUI-JOINS IN MAIN MEMORY
SIGMOD 2016

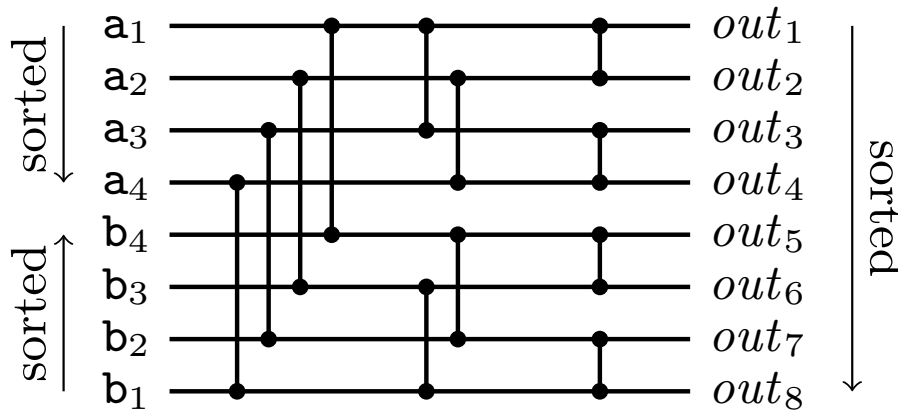
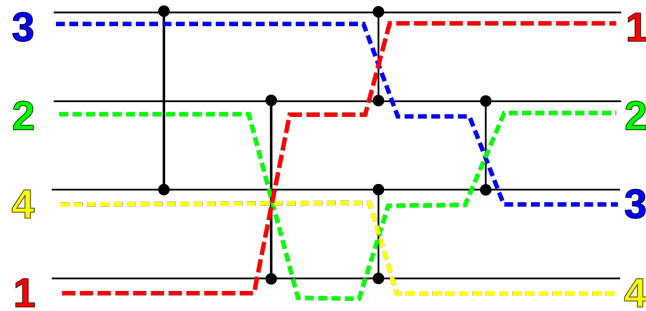
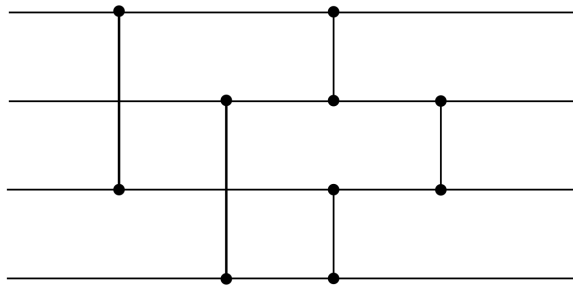


→ Hold up everyone! Let's look at everything more carefully!

Multi-core: Sorting and Hashing

▶ Sorting and Bitonic Merge Networks

- Fewer branches and more amenable to SIMD (vectorization)



Query Optimization

- ▶ Goal: Given a SQL query, find the best “physical operator” tree to execute the query
 - Large number of logically equivalent algebraic representations for a query
 - Many operator trees for each algebraic expression
- ▶ For “cost-based” optimization, we need:
 - A space of plans to search through (search space)
 - Cost estimation techniques
 - Enumeration/search algorithm
- ▶ Heuristic optimizers typically use “rules”
 - e.g., push down selections as much as possible – typically a good idea but not always

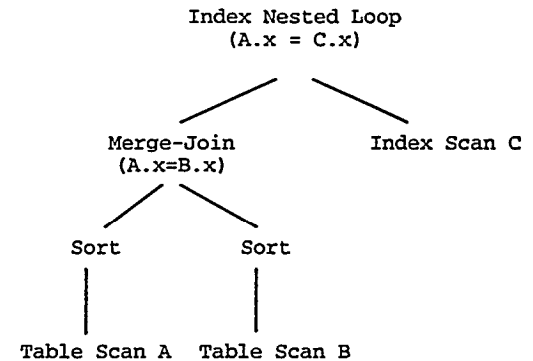



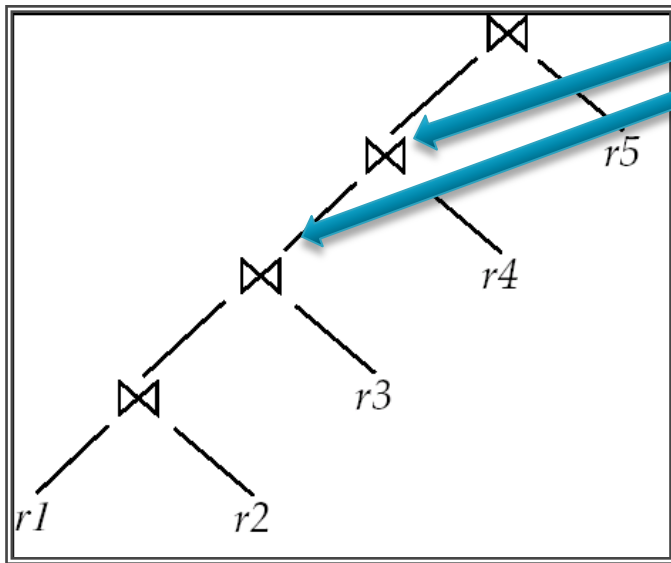
Figure 1. Operator Tree

System R Query Optimizer (1979)

- ▶ Focused on SPJ queries (select-project-join)
 - ▶ Search space:
 - Linear (left-deep) plans
 - Each join can be nested loop or sort-merge (no hash joins)
 - Each scan node either an index scan or a sequential scan
 - ▶ Cost estimation done using:
 - A set of statistics: #data pages for a relation, #distinct values in a column
 - Formulas for estimating intermediate result sizes
 - Relied on “magic” constants for anything not covered by the statistics
 - Formulas for CPU and I/O cost for each operator
- 

System R Query Optimizer (1979)

- ▶ Search algorithm: Bottom-up Dynamic Programming
 - Insight: the best overall plan uses the best plan for any subexpression inside of it

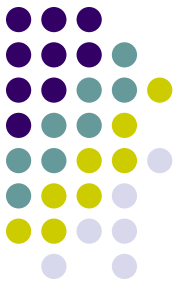


The best overall plan should use the “best” plan for (r1 join r2 join r3 join r4) and the “best” plan for (r1 join r2 join r3)..

e.g., if the best plan for r1 – r2 – r3 was to join r1 and r3 first and then join with r2, we can just substitute that plan, and get an overall better plan

Major caveat: the alternate plan should not miss any “physical properties” that are important
e.g., if the original plan produce r1-r2-r3 in sorted order by D, and the alternate doesn’t, the substitution may change the cost of the next join (with r4)

Dynamic Programming Algo.



- Join R1, R2, R3, R4, R5

R1 ⋈ R2 ⋈ R3

Options:

- Join R1R2 with R3 using HJ
cost = 100 + cost of this join
- Join R1R2 with R3 using SMJ
cost = 100 + cost of this join
- Join R1R3 with R2 using HJ
cost = 300 + cost of this join
- ...

R1 ⋈ R2
cost: 100
plan: HJ

R1 ⋈ R3
cost: 300
plan: SMJ

R1 ⋈ R4
....

R4 ⋈ R5
cost: 300
plan: HJ

R1

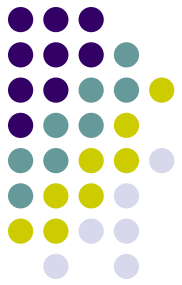
R2

R3

R4

R5

R1 ⋈ R2 ⋈ R3 ⋈ R4 ⋈ R5
cost: 1200
plan: *HJ(R1R2R3, R4R5)*

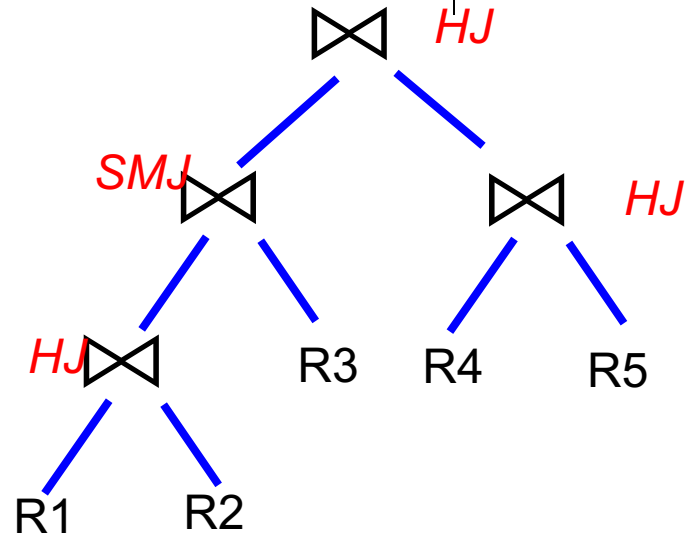


R1 ⋈ R2 ⋈ R3 ⋈ R4
cost: 700
plan: *HJ(R1R2R3, R4)*

R1 ⋈ R2 ⋈ R3
cost: 400
plan: *SMJ(R1R2, R3)*

....

....



R1 ⋈ R2
cost: 100
plan: HJ

R1 ⋈ R3
cost: 300
plan: SMJ

R1 ⋈ R4
....

....

R4 ⋈ R5
cost: 300
plan: HJ

R1

R2

R3

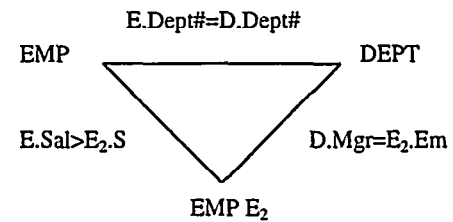
R4

R5

System R Query Optimizer (1979)

- ▶ Interesting orders
 - Sort orders is an important physical property for the query executor (given the reliance on sort-merge joins)
 - So keep track of the sort order in which results are generated
 - Two plans for a subexpression are NOT comparable if the sort orders are different
 - ➔ For each subexpression, more than one plan may be maintained with different sort orders
- ▶ Can be generalized to handle “incomparable-ness” in general
 - e.g., one subplan may have better CPU but worse Memory, and the other subplans may have better Memory but worse CPU

Search Space: Reordering

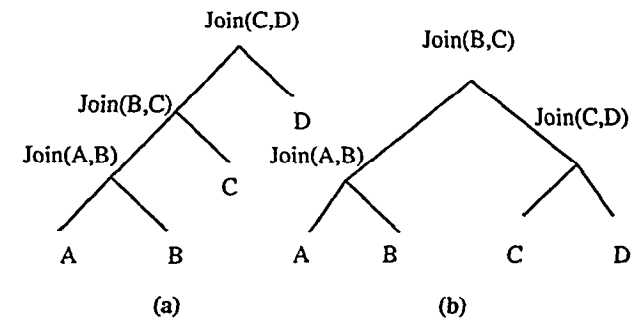


▶ Intermediate representations

- Query graphs commonly used in research papers, but only capture a simple subset
- QGM Structure used in Starburst (will cover later)
- Many others just use an “operator tree” or an “expression tree”

▶ Join ordering

- Bushy plans commonly considered today
- Significantly add to the search complexity
- Cartesian products may be allowed in some cases



▶ Outerjoins

- Only commute with joins in some cases (will cover later)
- e.g., $\text{Join}(R, S \text{ LOJ } T) = \text{Join}(R, S) \text{ LOJ } T$

Search Space: Reordering

▶ Group-By and Joins

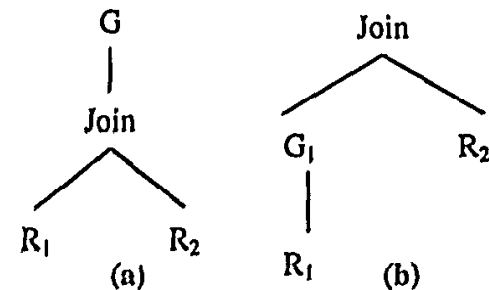
- Pushing group by below a join results in significant reductions in tuples being joined

```
select R1.A, sum(R1.B)
from R1, R2
where R1.A = R2.A
group by R1.A
```

equivalent to

```
select x.A, x.sumB
from R2, (select A, sum(R1.B) as sumB
         from R1
         group by A) x
where R2.A = x.A
```

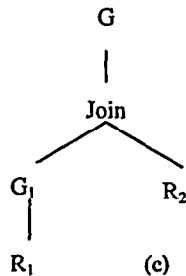
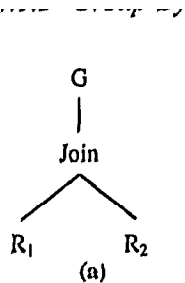
only if: A is a primary key of R2



Search Space: Reordering

▶ Group-By and Joins

- Pushing group by below a join results in significant reductions in tuples being joined



```
select R1.A, sum(R1.B)
from R1, R2
where R1.A = R2.A
group by R1.A
```

equivalent to

```
select R2.C, sum(x.sumB)
from R2, (select A, sum(R1.B) as sumB
          from R1
          group by A) x
where R2.A = x.A
group by R2.C
```

only if: in R2, $A \rightarrow C$

Search Space: Subqueries

- ▶ Collapsing nested subqueries results in more optimization opportunities
 - Need to be very careful: NULLs, Distincts, Aggregates, etc., cause problems

```
SELECT Emp.Name
FROM Emp
WHERE Emp.Dept# IN
      SELECT Dept.Dept# FROM Dept
      WHERE Dept.Loc='Denver'
      AND Emp.Emp# = Dept.Mgr
```



```
SELECT E.Name
FROM Emp E, Dept D
WHERE E.Dept# = D.Dept#
AND D.Loc = 'Denver' AND E.Emp# = D.Mgr
```

Search Space: Subqueries

- ▶ Collapsing nested subqueries results in more optimization opportunities
 - Need to be very careful: NULLs, Distincts, Aggregates, etc., cause problems

```
SELECT Dept.name
FROM Dept
WHERE Dept.num-of-machines ≥
(SELECT COUNT(Emp.*) FROM Emp
WHERE Dept.name= Emp.Dept_name)
```



```
SELECT Dept.name FROM Dept LEFT OUTER JOIN Emp
ON (Dept.name= Emp.dept_name )
GROUP BY Dept.name
HAVING Dept. num-of-machines < COUNT (Emp.*)
```

LOJ is essential here
Otherwise will miss depts with no employees

Search Space: Semijoins for Optimizing

```
CREATE VIEW DepAvgSal As (  
    SELECT E.did, Avg(E.Sal) AS avgsal  
    FROM Emp E  
    GROUP BY E.did)  
SELECT E.eid, E.sal  
FROM Emp E, Dept D, DepAvgSal V  
WHERE E.did = D.did AND E.did = V.did  
AND E.age < 30 AND D.budget > 100k  
AND E.sal > V.avgsal
```



```
CREATE VIEW partialresult AS  
(SELECT E.id, E.sal, E.did  
    FROM Emp E, Dept D  
    WHERE E.did=D.did AND E.age < 30  
    AND D.budget > 100k)  
CREATE VIEW Filter AS  
(SELECT DISTINCT P.did FROM PartialResult P)  
CREATE VIEW LimitedAvgSal AS  
(SELECT E.did, Avg(E.Sal) AS avgsal  
    FROM Emp E, Filter F  
    WHERE E.did = F.did GROUP BY E.did)  
  
SELECT P.eid, P.sal  
FROM PartialResult P, LimitedDepAvgSal V  
WHERE P.did = V.did AND P.sal > V.avgsal
```

- ▶ Say only a few departments (say 10) satisfy the join condition out of, say 10000
 - Only need to compute the “view” tuples for those 10 departments
- ▶ So we are passing information “sideways” from the main block into the nested block


Statistics and Cost Estimation

- ▶ In general: more information about the data → better estimates
- ▶ Single-column statistics
 - min, max, #distinct, #bytes, etc.
 - Histograms for value distributions (e.g., to estimate #tuples satisfying “age < 20”)
 - Many different types of histograms proposed over the years
- ▶ Multi-column statistics
 - Correlations among attributes a major issue for estimates
 - Queries of type: “SSN = 0123 and Name = ‘John Smith’” pretty common
 - Independence assumption → huge underestimation of the result size
 - Many proposals for capturing correlations, but hard to make work in practice
- ▶ Propagation of errors
 - Even if estimates lower in the query plan are pretty good, estimates for more complex subexpressions become erroneous very quickly


Enumeration Architectures

- ▶ Need the optimization algorithm to be “extensible”
 - So it can handle new physical operators, new transformations, new cost estimation approaches, easily
- ▶ Starburst:
 - Uses a rule engine and an intermediate representation called QGM to do query rewrites/transformations
 - Uses a somewhat generalized bottom-up query optimizer
- ▶ Volcano/Cascades:
 - Transformation rules to map algebraic expressions
 - Implementation rules to map algebraic expression into an operator tree
 - Uses a “top-down” query optimizer
 - Starts with the overall expression and tries to find all possible ways to get to it
 - Uses “memoization” to keep avoid redoing work
 - Formed the basis of the Microsoft database systems

More...

- ▶ **Distributed and Parallel Databases**
 - Much bigger search space (can place operators anywhere, and can partition them)
 - What to optimize for? Communication cost? Total resources? Response time?
 - Standard approach is to generate a single-machine query plan and then parallelize it (2-phase optimization)
 - ▶ **User-defined Functions**
 - Need to consider the cost of executing those (can be hard to estimate)
 - ▶ **Materialized views**
 - Given a set of materialized views, hard to decide if those can be used in place of the original relations (undecidable in general)
 - ▶ ...
- 

Outline

- ▶ Query evaluation techniques for large databases
 - ▶ Skew avoidance strategies
 - ▶ Query compilation
 - ▶ Vectorization
 - ▶ Query Optimization: Overview
 - ▶ How good are the query optimizers, really?
- 

JOB Benchmark

- ▶ Build using the IMDB dataset
 - 21 tables, total of 3.6 GB in CSV format
- ▶ 113 SPJ queries – no aggregates or subqueries
- ▶ More realistic than the commonly used TPC-H/DS benchmarks (or synthetic benchmarks)

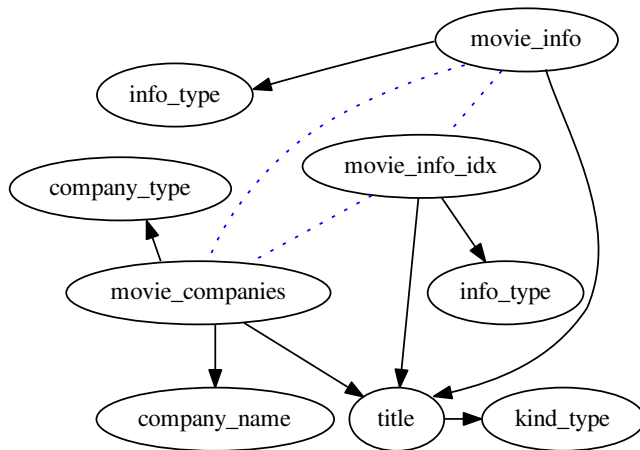
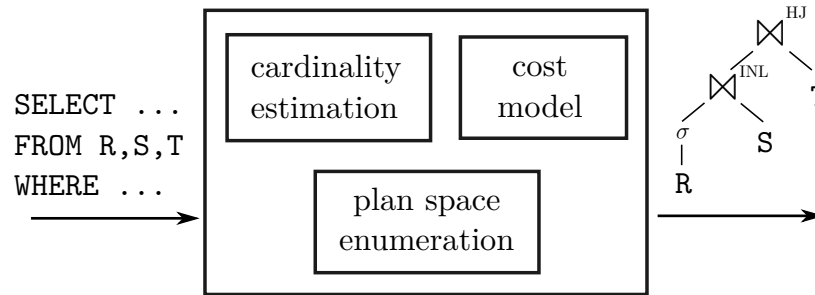


Figure 2: Typical query graph of our workload

```
SELECT cn.name, mi.info, miidx.info
FROM company_name cn, company_type ct,
      info_type it, info_type it2, title t,
      kind_type kt, movie_companies mc,
      movie_info mi, movie_info_idx miidx
WHERE cn.country_code = ' [us]'
AND ct.kind = 'production companies'
AND it.info = 'rating'
AND it2.info = 'release dates'
AND kt.kind = 'movie'
AND ... -- (11 join predicates)
```

PostgreSQL Query Optimizer

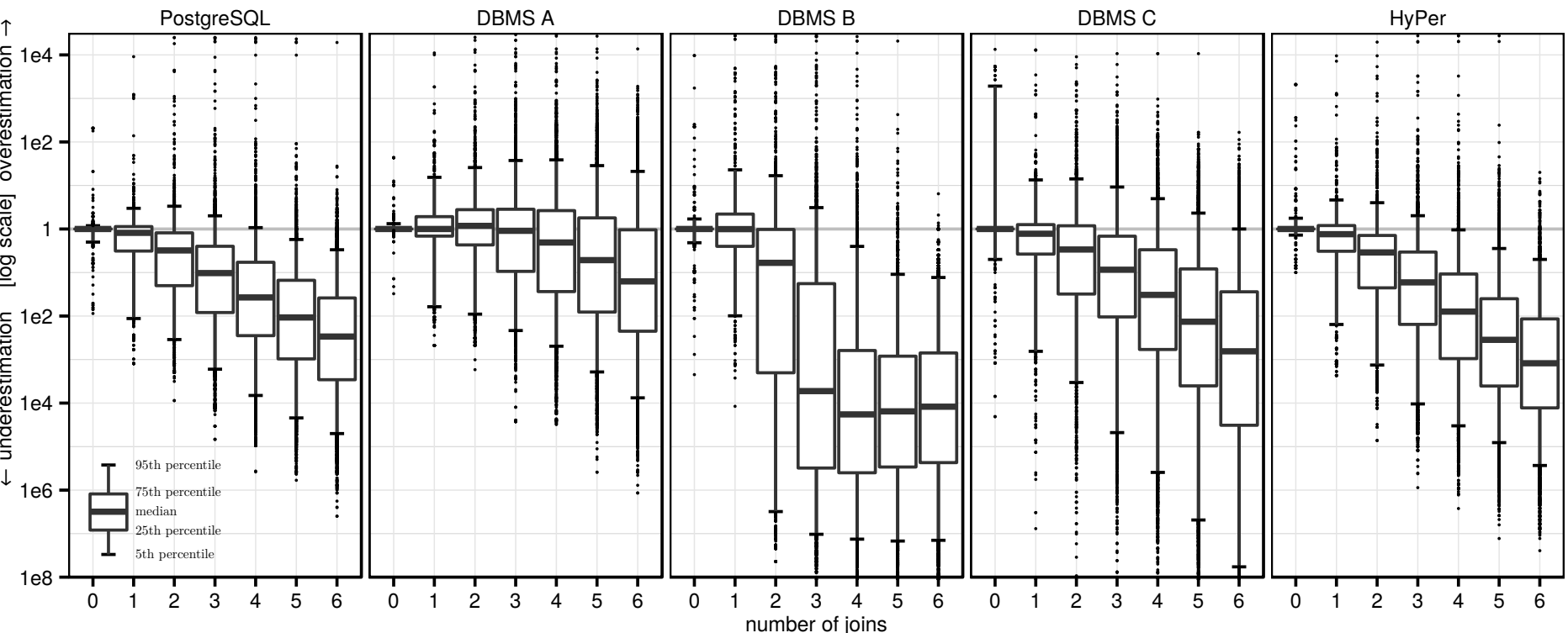
- ▶ Standard dynamic programming-based optimizer
 - Includes bushy plans, but no Cartesian products
- ▶ Statistics: Single-column histograms, min, max, most frequent values, etc.
 - Assume independence and uniformity outside of those
 - Especially for conjunctive predicates (like $A = 10$ and $B = 20$)



- ▶ Modified for the purposes of this paper to accept “cardinality injection”
 - i.e., use different cardinality estimates than the ones it computed
 - e.g., true cardinalities, or cardinalities per another system

Results: Cardinality Estimation

- ▶ q-error: ratio of correct result and estimate
- ▶ Base tables: sampling (Hyper and A) works better than histograms
- ▶ Huge underestimation seen as #joins increases
 - Underestimation generally worse – results in more aggressive plans (e.g., NL joins)
- ▶ Note: The experimental setup *may* naturally “select” for underestimates
 - (Missing enough details to be sure)



Results: What if we used “correct” estimates

- ▶ Used cardinality injection to use other systems’ estimates or the true cardinalities
- ▶ Most bad plans boil down to NL joins
 - Disabling improves performance but doesn’t fully solve the problem

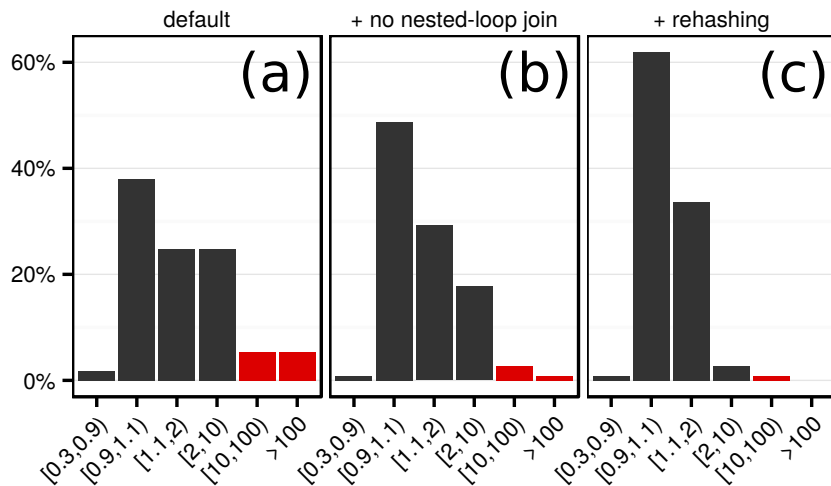


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

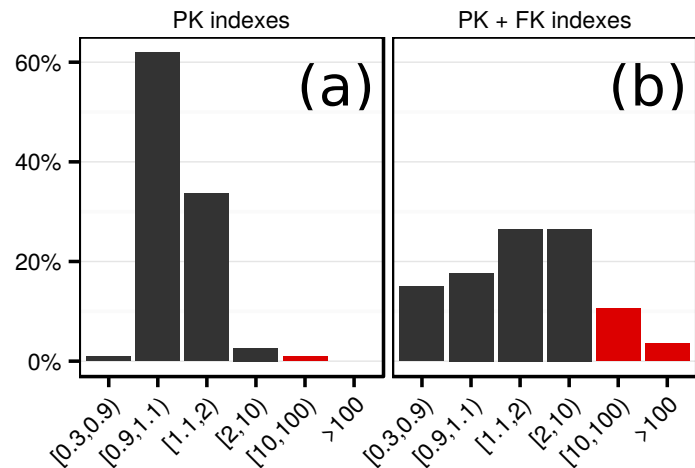


Figure 7: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (different index configurations)

Results: Cost Models

- ▶ PostgreSQL uses a disk-oriented cost model – a weighted sum of I/O and CPU costs
 - No easy way to set the parameters
- ▶ Plot predicted costs vs actual costs – a linear line is the best outcome here
- ▶ Findings:
 - Default estimates result in fairly poor fit – predicted and actual costs quite different
 - Most of the error goes away if the optimizer has access to true cardinalities
 - Tuning the cost model doesn't really help that much
 - Using a much simpler cost model gives similar results
 - Just count the number of tuples being processed by each operator

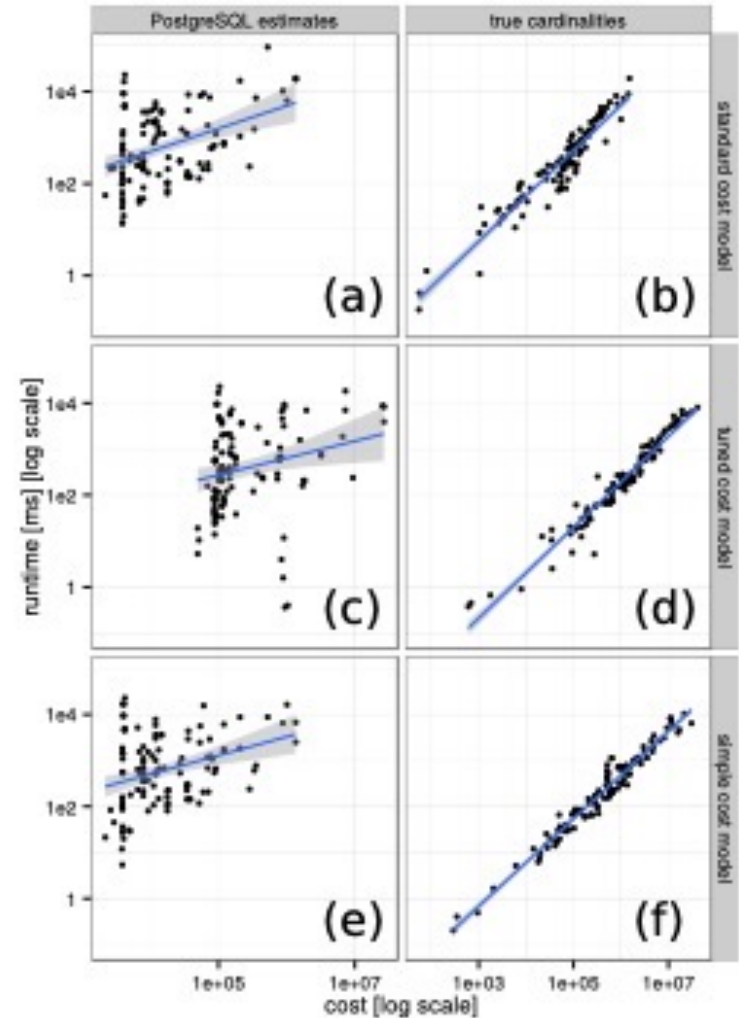


Figure 8: Predicted cost vs. runtime for different cost models

Results: Join Orders

Computed estimated costs with true cardinalities for 1000 random plans

Slowest or even median query plans much worse than optimal (several orders of magnitude in many cases)

Prior work from approx. 20 years ago that does this in more depth

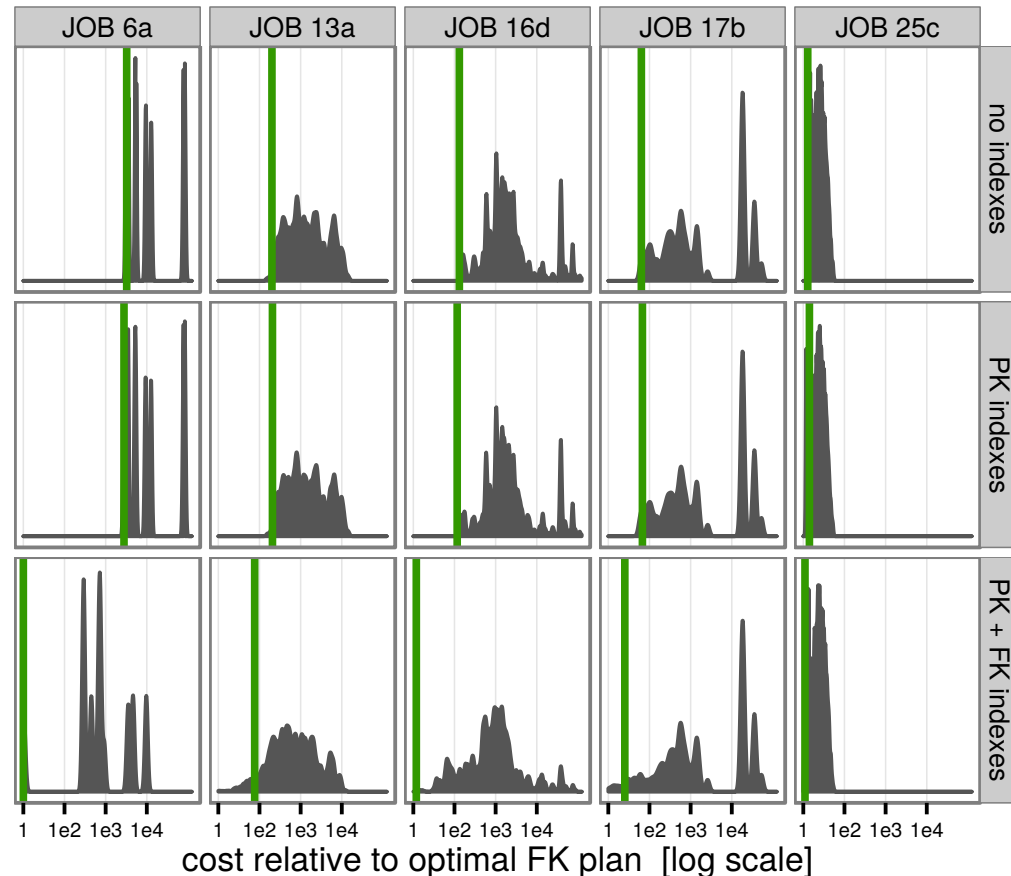


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

Results: Join Orders

- ▶ Bushy trees important to consider

	PK indexes			PK + FK indexes		
	median	95%	max	median	95%	max
zig-zag	1.00	1.06	1.33	1.00	1.60	2.54
left-deep	1.00	1.14	1.63	1.06	2.49	4.50
right-deep	1.87	4.97	6.80	47.2	30931	738349

Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)

- ▶ Exhaustive algorithms (DP or top-down) needed

	PK indexes						PK + FK indexes					
	PostgreSQL estimates			true cardinalities			PostgreSQL estimates			true cardinalities		
	median	95%	max	median	95%	max	median	95%	max	median	95%	max
Dynamic Programming	1.03	1.85	4.79	1.00	1.00	1.00	1.66	169	186367	1.00	1.00	1.00
Quickpick-1000	1.05	2.19	7.29	1.00	1.07	1.14	2.52	365	186367	1.02	4.72	32.3
Greedy Operator Ordering	1.19	2.29	2.36	1.19	1.64	1.97	2.35	169	186367	1.20	5.77	21.0

Table 3: Comparison of exhaustive dynamic programming with the Quickpick-1000 (best of 1000 random plans) and the Greedy Operator Ordering heuristics. All costs are normalized by the optimal plan of that index configuration

Correlations

▶ Single-table (e.g., R.A and R.B are correlated, throwing off estimation of R.A = 10 and R.B = 20)


- Handled by the “sampling” techniques
- Build multi-dimensional histograms (don't really work well)
- Identify “soft” functional dependencies (i.e., very highly correlated columns)
 - e.g., “car make” and “car model” are highly correlated
 - Queries like: Make = Honda and Model = Accord are underestimated
 - But not a functional dependency: Model → Make is false

▶ Join-crossing Correlations

```
select *  
from actors JOIN movies  
where actors.location = 'Paris' and movies.language = 'French'
```

- Unclear how one can benefit from capturing this correlation (even if one could)
- Need a new operator or access method

Outline

- ▶ Query evaluation techniques for large databases
 - ▶ Skew avoidance strategies
 - ▶ Query compilation
 - ▶ Vectorization
 - ▶ Query Optimization: Overview
 - ▶ How good are the query optimizers, really?
 - ▶ Reordering for Outerjoins
 - ▶ **Query Rewriting**
 - Starburst
 - Unnesting arbitrary queries
 - APPLY (SQL Server)
- 

Why Query Rewrite?

- ▶ Many queries are written in a way that forces a procedural execution
 - Use of WITH clause or Views to simplify
 - Procedural code easier for users to write
 - Modern frameworks/query languages often not that declarative
 - Automated translation of other DSLs into SQL
 - Program synthesis?

```
WITH TBL1 AS (SELECT p.id AS pid, q.id AS qid, p.temp AS temp,
                p.weight AS weight, log(sum(q.weight)) AS logsum
             FROM particles p, particles q
             WHERE pid != qid AND p.temp < q.temp
                AND p.time = 20 AND q.time = 20
             GROUP BY p.id, p.temp, p.weight, q.id
            ) WITH TBL2 AS (SELECT pid, temp, weight, exp(sum(logsum)) AS prob
             FROM TBL1 GROUP BY pid, temp
             HAVING Count(*) = (SELECT COUNT distinct id FROM particles)+1
            )
(a) VMinQ: SELECT sum(prob*weight*temp) FROM TBL2;
(b) EMinQ: SELECT pid, sum(prob*weight) FROM TBL2 GROUP BY pid;
```

- ▶ Harder for optimizers to deal with
 - Join order optimization usually goes block-by-block → significant benefits in reducing the number of blocks
 - Redundant DISTINCTs etc., lead to unnecessary work

Two Main Issues

▶ Merging of select blocks

- Different “blocks” get created because of:
 - WITH, Views
 - Table expressions in FROM (e.g., `select * from R, (select S.A, max(S.B) from S group by S.A) X...`)
 - Table expressions in WHERE/SELECT/HAVING etc. (e.g., `where R.A in (select S.A from S)`)
 - Scalar expressions in WHERE/SELECT/HAVING etc. (e.g., `where R.A = (select max(S.A) from S)`)

▶ Correlations Across Blocks

- When an “lower” block refers to an “upper” block
- Forces a “dependent” “nested-loops” execution
 - For every tuple in the outer block, the inner block is executed

Example

```
select *
from users
where users.userid in
(select userid
 from status
 group by userid
 having count(*) > 5);
```

```
with temp as
    (select userid
     from status
     group by userid
     having count(*) > 5)
select *
from users
where users.userid in (select userid from temp);
```

```
select *
from users
where exists
    (select userid
     from status
     where status.userid = users.userid
     group by userid
     having count(*) > 5);
```

Correlated



Join Operators and Implementations

R(A, B), and S(B, C)

R Natural Join S

```
ht = dict()
for r in R:
    if r.B in ht:
        ht[r.B].append(r)
    else:
        ht[r.B] = [r]
for s in S:
    for r in ht.get(s.B, []):
        yield (s, r)
```

Most other join operators built as minor modifications (special cases) of this basic code

S Semi Join R (build on R)

```
ht = set()
for r in R:
    ht.add(r.B)
for s in S:
    if s.B in ht:
        yield s
```

R Semi Join S (build on R)

```
ht = dict()
for r in R:
    if r.B not in ht:
        ht[r.B].append(r)
    else:
        ht[r.B] = [r]
for s in S:
    for r in ht[s.B]:
        yield r
    ht[s.B] = [] -- avoid
                  duplicates
```

Join Operators and Implementations

R(A, B), and S(B, C)

R Natural Join S

```
ht = dict()
for r in R:
    if r.B in ht:
        ht[r.B].append(r)
    else:
        ht[r.B] = [r]
for s in S:
    for r in ht.get(s.B, []):
        yield (s, r)
```

S Anti Join R

```
ht = set()
for r in R:
    ht.add(r.B)
for s in S:
    if s.B not in ht:
        yield s
```

R Anti Join S

```
ht = dict()
for r in R:
    if r.B not in ht:
        ht[r.B].append(r)
    else:
        ht[r.B] = [r]
for s in S:
    ht[s.B] = [] --- remove r
for r in ht.values():
    yield r
```

Join Operators and Implementations

R(A, B), and S(B, C)

R Natural Join S

```
ht = dict()
for r in R:
    if r.B in ht:
        ht[r.B].append(r)
    else:
        ht[r.B] = [r]
for s in S:
    for r in ht.get(s.B, []):
        yield (s, r)
```

R Full Outer Join S

```
ht = dict()
found_set = set()
for r in R:
    if r.B in ht:
        ht[r.B].append(r)
    else:
        ht[r.B] = [r]
for s in S:
    if s.B in ht:
        found_set.add(s.B)
        for r in ht[s.B]:
            yield (s, r)
    else:
        yield (NULLS, s)

for x in ht:
    if x not in found_set:
        for r in ht[x]:
            yield(r, NULLS)
```