CS 4440 A

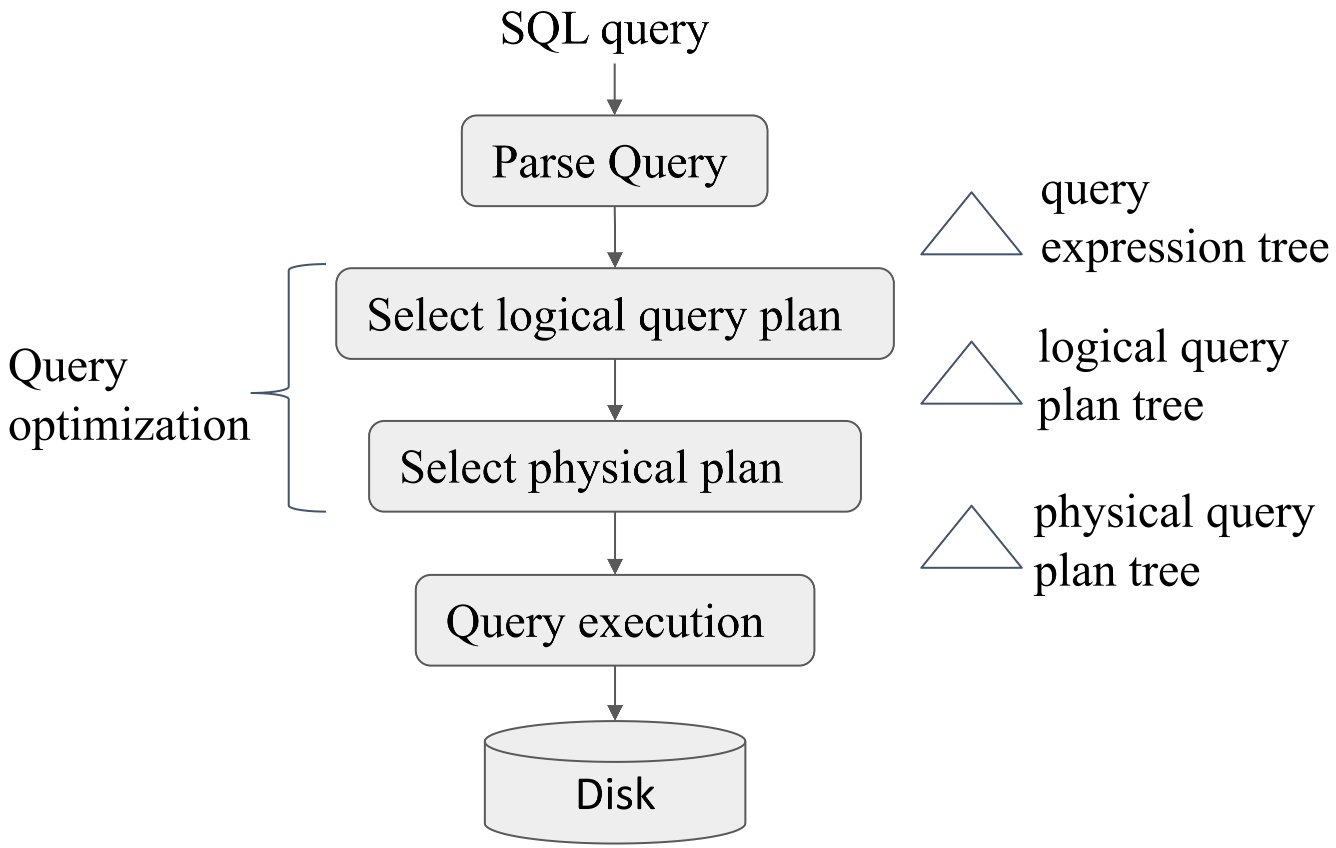
Emerging Database Technologies

Lecture 9

02/07/24

Recap

- Query processor overview
- Estimate the size of results
  - Projection
  - Selection
  - Joins
- Estimate the # of disk I/O's
  - Nested-loop join
  - Hash join
  - Index join



Query Optimization Overview

Output: A good physical query plan

Basic *cost-based query optimization* algorithm

- Enumerate candidate query plans (logical and physical)
- Compute estimated cost of each plan (e.g., number of I/Os)
  - Without executing the plan!
- Choose plan with lowest cost

The Three Parts of an Optimizer

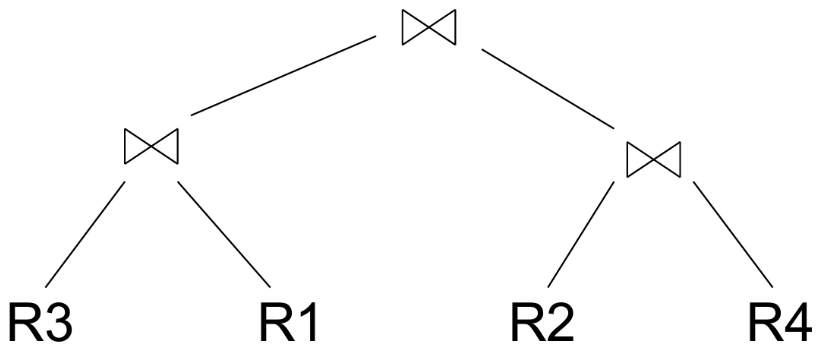
- Cost estimation
  - Estimate size of results
  - Also consider whether output is sorted/intermediate results written to disk etc.

- Search space
  - Algebraic laws, restricted types of join trees

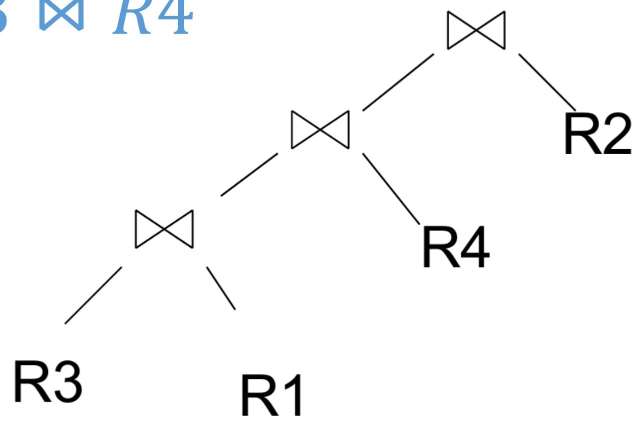
- Search algorithm
  - Example: Selinger algorithm

Search Space

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Bushy plan



Left-deep plan

Logical plan space:

- Several possible structures of the trees
- Each tree can have n! permutations of relations on leaves

Physical plan space:

- Different implementation (e.g., join algorithm) and scanning of intermediate operators for each logical plan

Heuristic for pruning plan space

Apply predicates as early as possible

Avoid plans with cartesian products

- $(R(A,B) \bowtie T(C,D)) \bowtie S(B,C)$

Consider only left-deep join trees

- Studied extensively in traditional query optimization literature
- Works well with existing join algorithms such as nested-loop and hash join
  - e.g., might not need to write tuples to disk if enough memory

Search Algorithm

Selinger Algorithm: dynamic programming based

- Based on System R (aka Selinger) style optimizer [1979]
- Consider different logical and physical plans at the same time
- Limited to joins: join reordering algorithm
- Cost of a plan is I/O + CPU

Exploits ”principle of optimality”

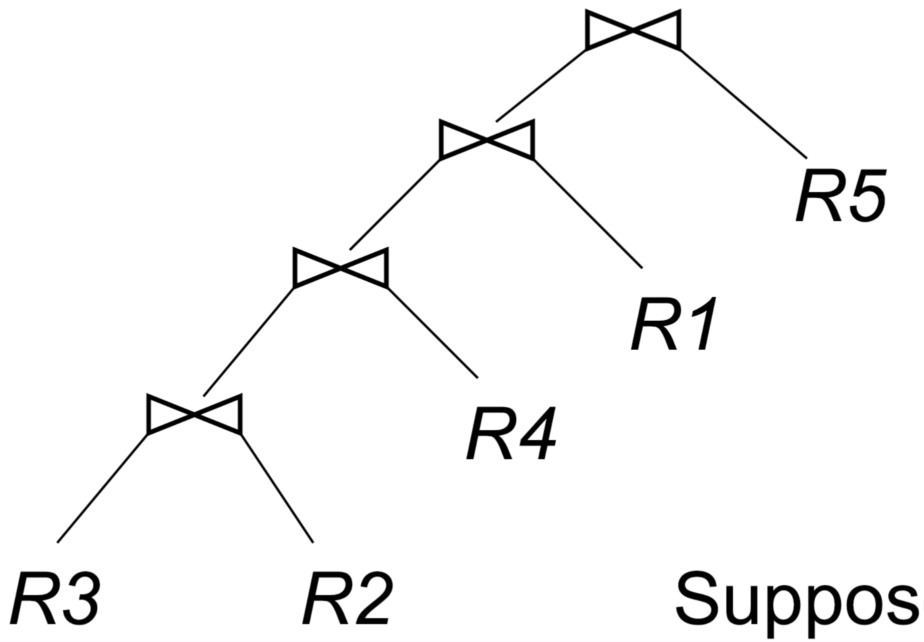
- Optimal for “whole” made up from optimal for ”parts”

Consider the search space of left-deep join trees

- Reduces search space but still n! permutations

Principle of Optimality

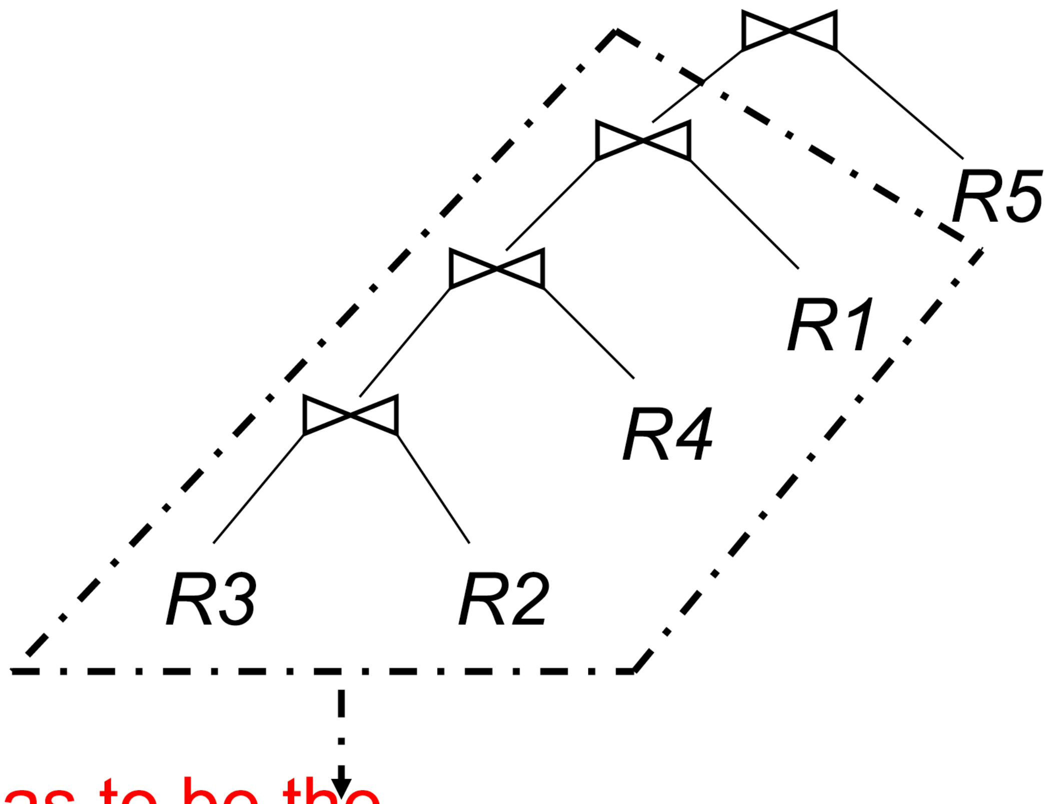
Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5$



Suppose,
this is an Optimal Plan
for joining R1…R5:

Principle of Optimality

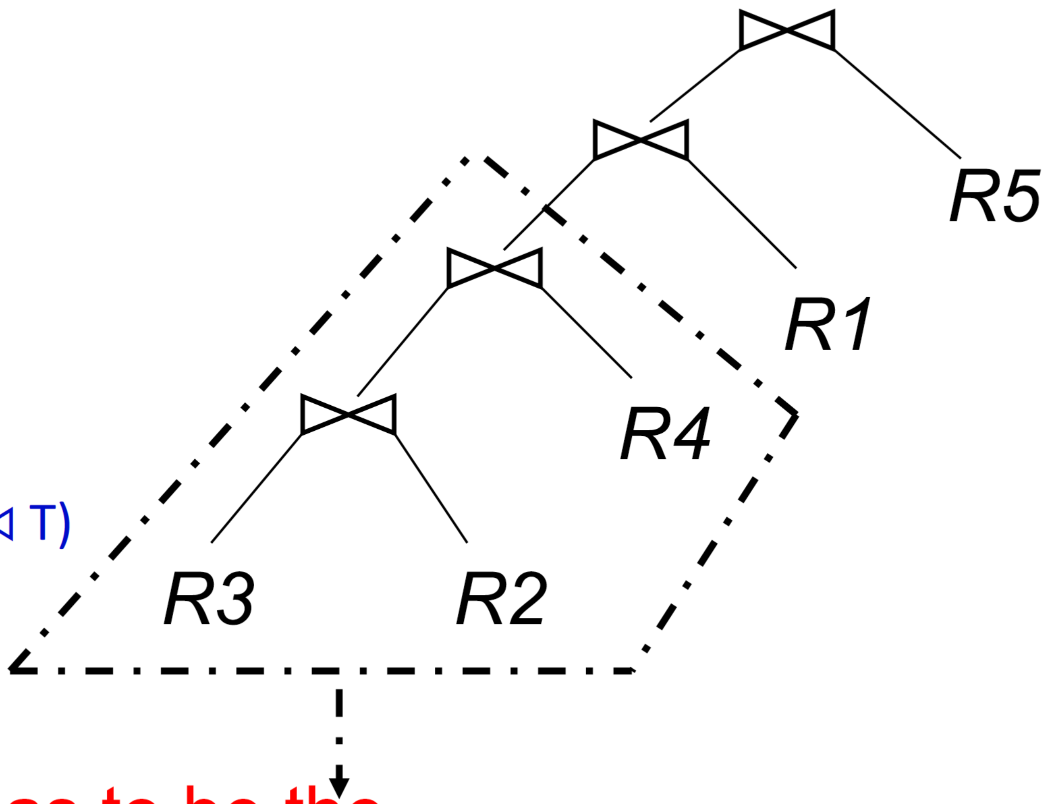
Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5$



This has to be the
optimal plan for joining *R3, R2, R4, R1*

Principle of Optimality

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5$



We are using the associativity and commutativity of joins

$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

$R \bowtie S = S \bowtie R$

This has to be the optimal plan for joining *R3, R2, R4*

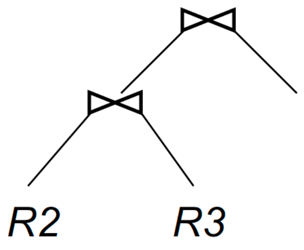
Slides adapted from Duke CompSci 516 by Sudeepa Roy

Principle of Optimality

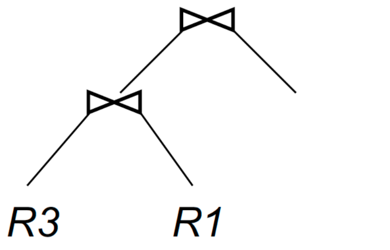
Query: $R1 \bowtie R2 \bowtie \quad \ldots \quad \bowtie Rn$

Both are giving the same result

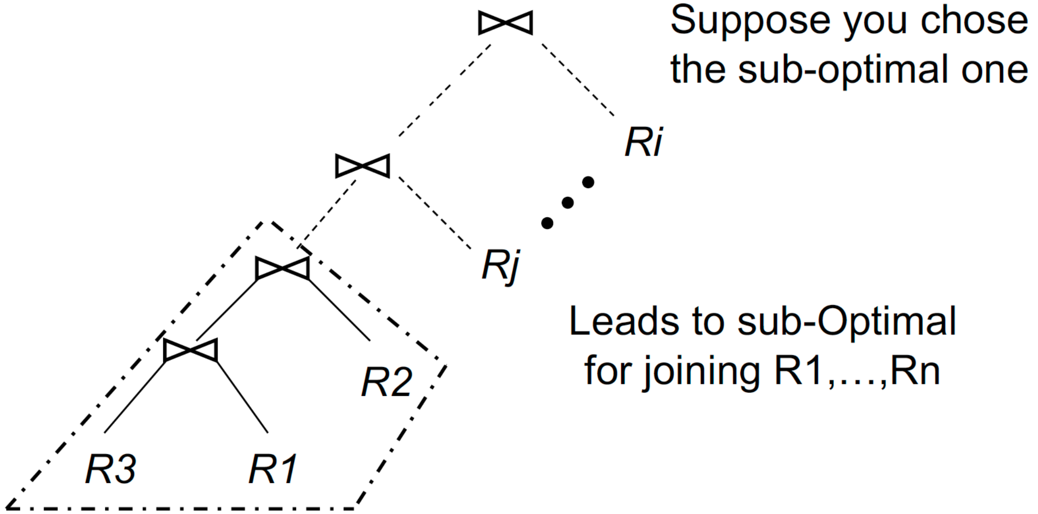
R2 ⋈ R3 ⋈ R1 = R3 ⋈ R1 ⋈ R2



Optimal
for joining *R1, R2, R3*



Sub-Optimal
for joining *R1, R2, R3*



Notation and Setup

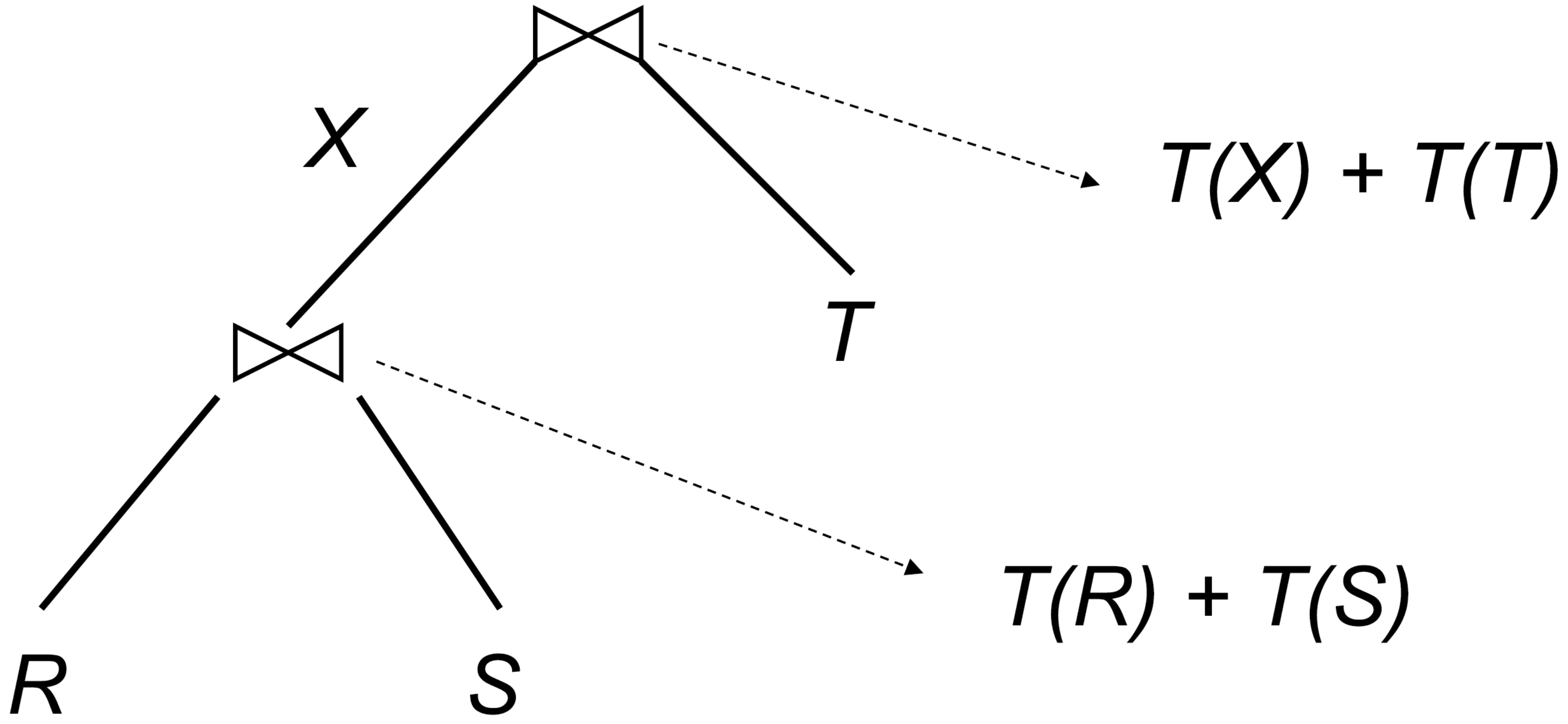
OPT({R1, R2, R3}):
Cost of optimal plan to join R1, R2, R3

T({R1, R2, R3}):
Number of tuples in $R1 \bowtie R2 \bowtie R3$

Simple Cost Model: $\text{Cost}(R \bowtie S) = T(R) + T(S)$
All other operations have 0 cost

* The simple cost model used for illustration only, it is not used in practice

Cost Model Example



Total Cost: $T(R) + T(S) + T(T) + T(X)$

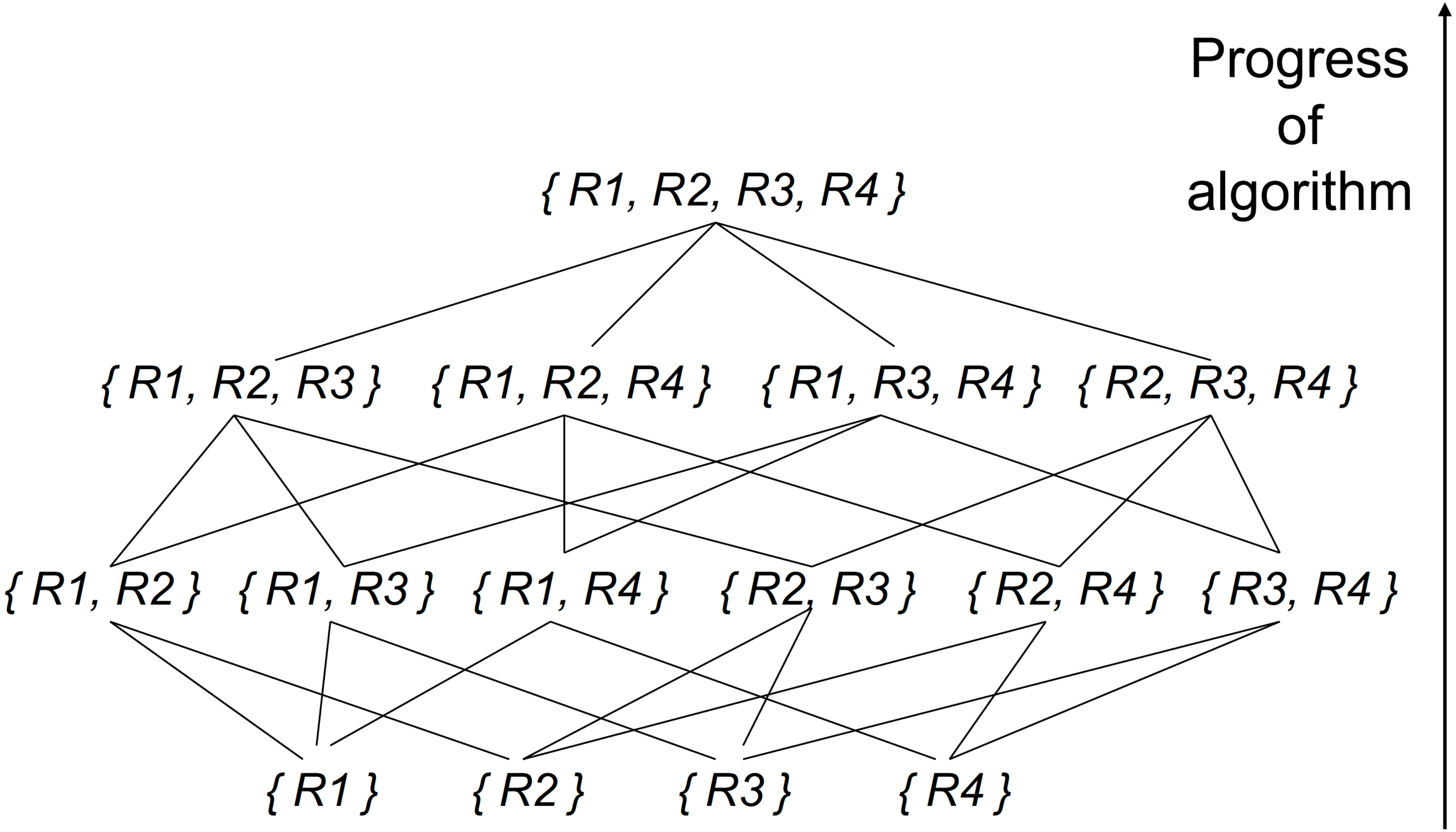
Selinger Algorithm

$$OPT(\{R1, R2, R3\}) = min \begin{cases} OPT(\{R1, R2\}) + T(\{R1, R2\}) + T(R3) \\ OPT(\{R2, R3\}) + T(\{R3, R3\}) + T(R1) \\ OPT(\{R1, R3\}) + T(\{R1, R3\}) + T(R2) \end{cases}$$

* Valid only for the simple cost model

Selinger Algorithm

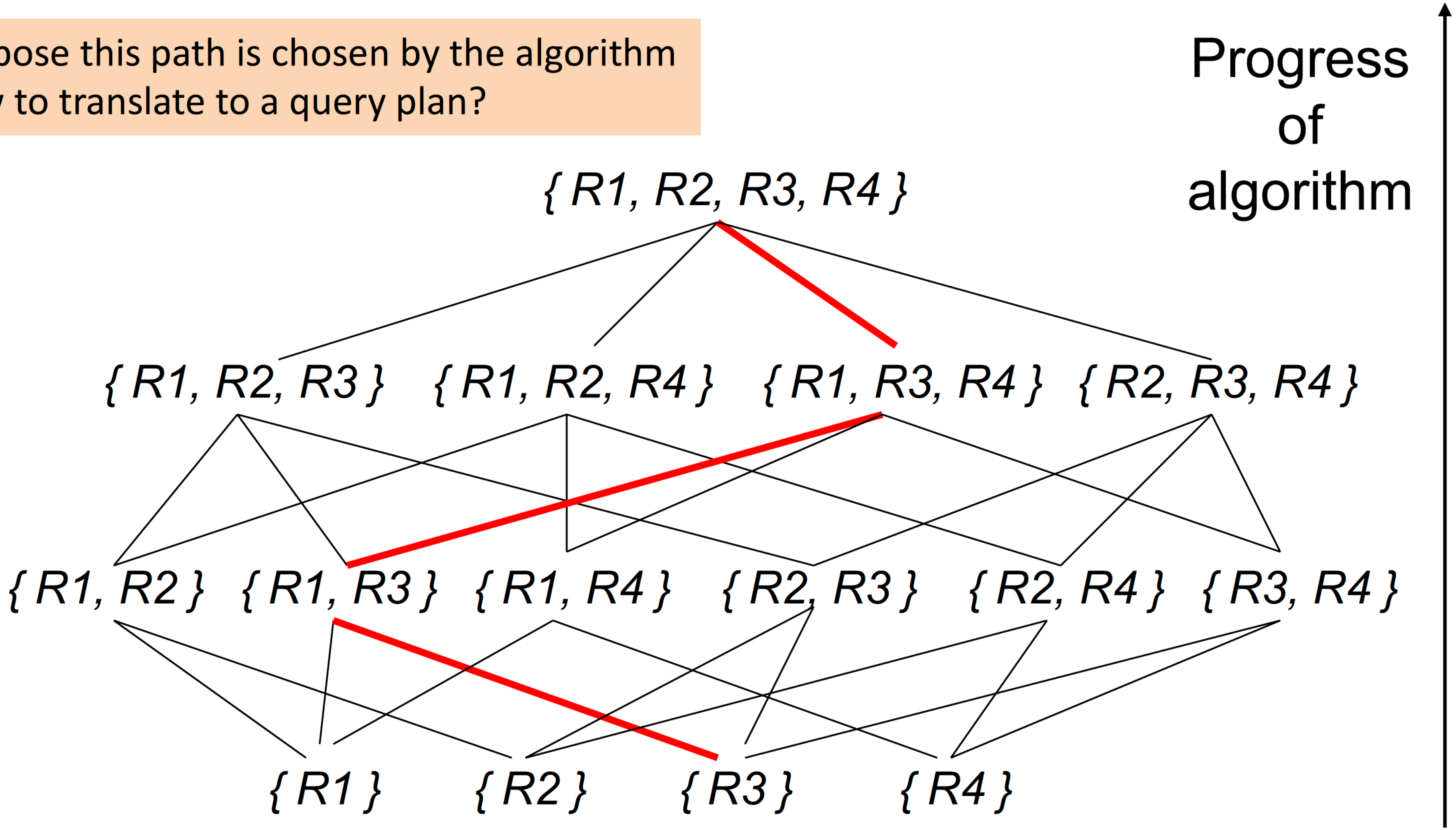
Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Selinger Algorithm

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$

Suppose this path is chosen by the algorithm
How to translate to a query plan?



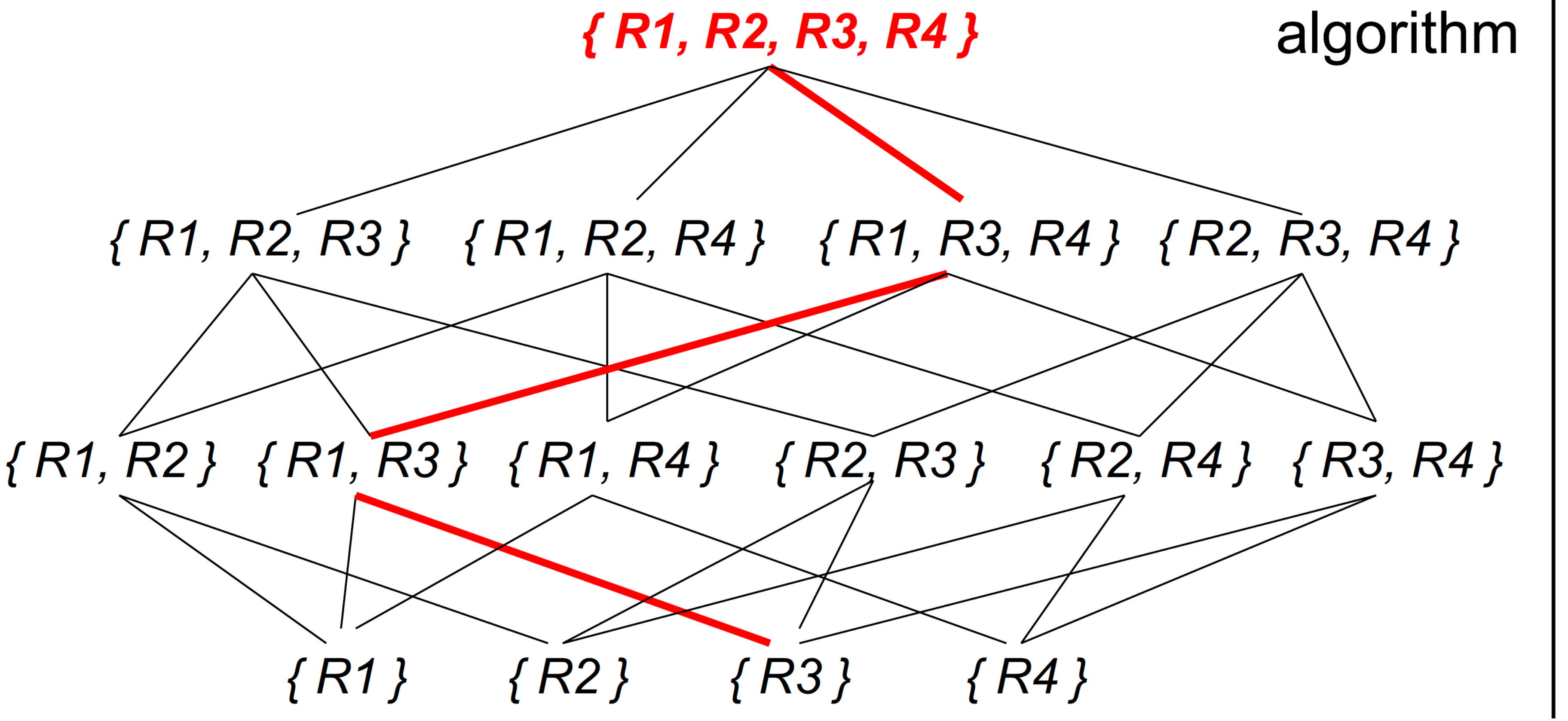
Slides adapted from Duke CompSci 516 by Sudeepa Roy

Selinger Algorithm

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$

Q. How to optimally compute join of {R1, R2, R3, R4}?

Ans: First optimally join {R1, R3, R4} then join with R2 as inner.

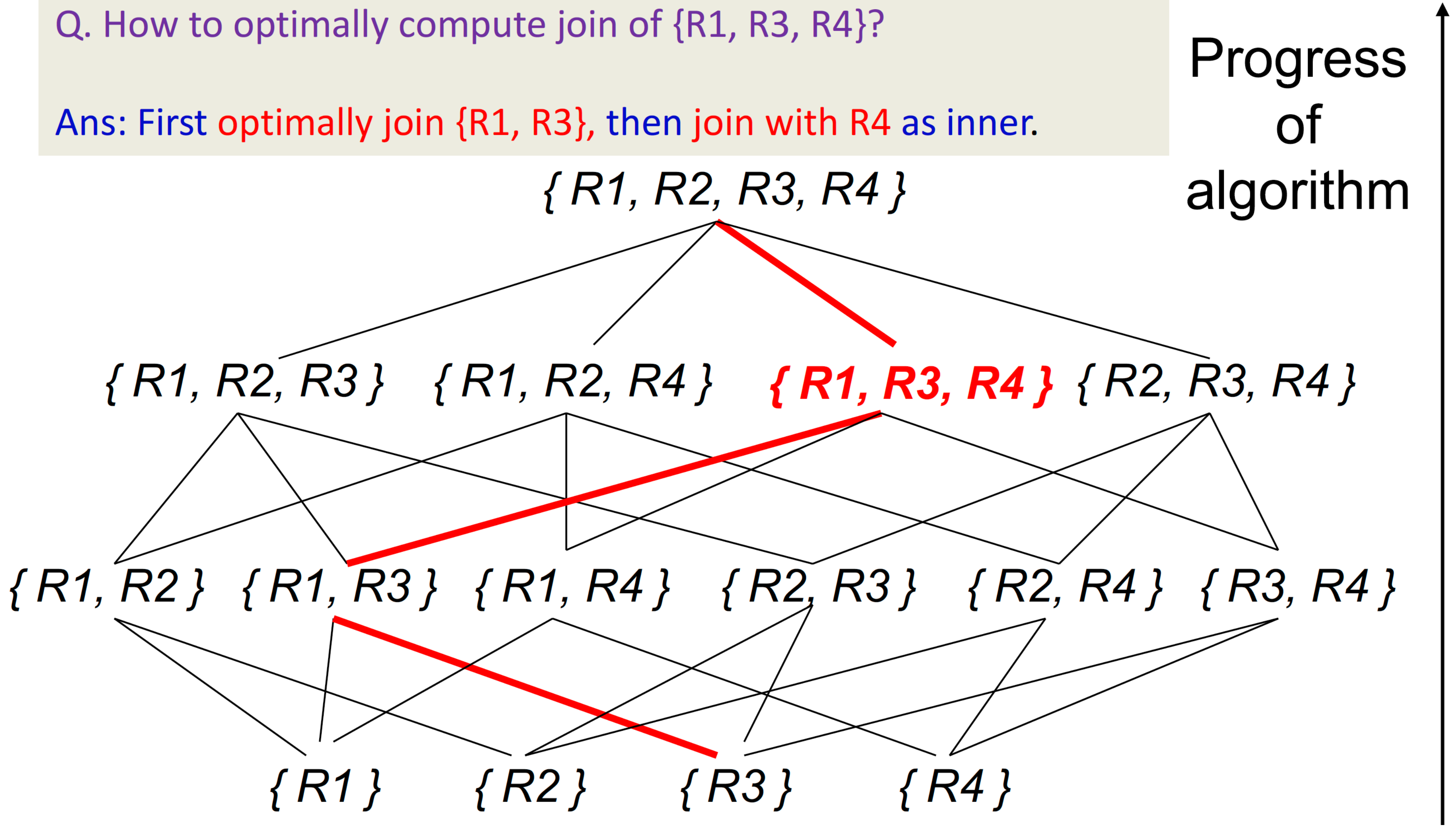


Selinger Algorithm

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$

Q. How to optimally compute join of {R1, R3, R4}?

Ans: First optimally join {R1, R3}, then join with R4 as inner.



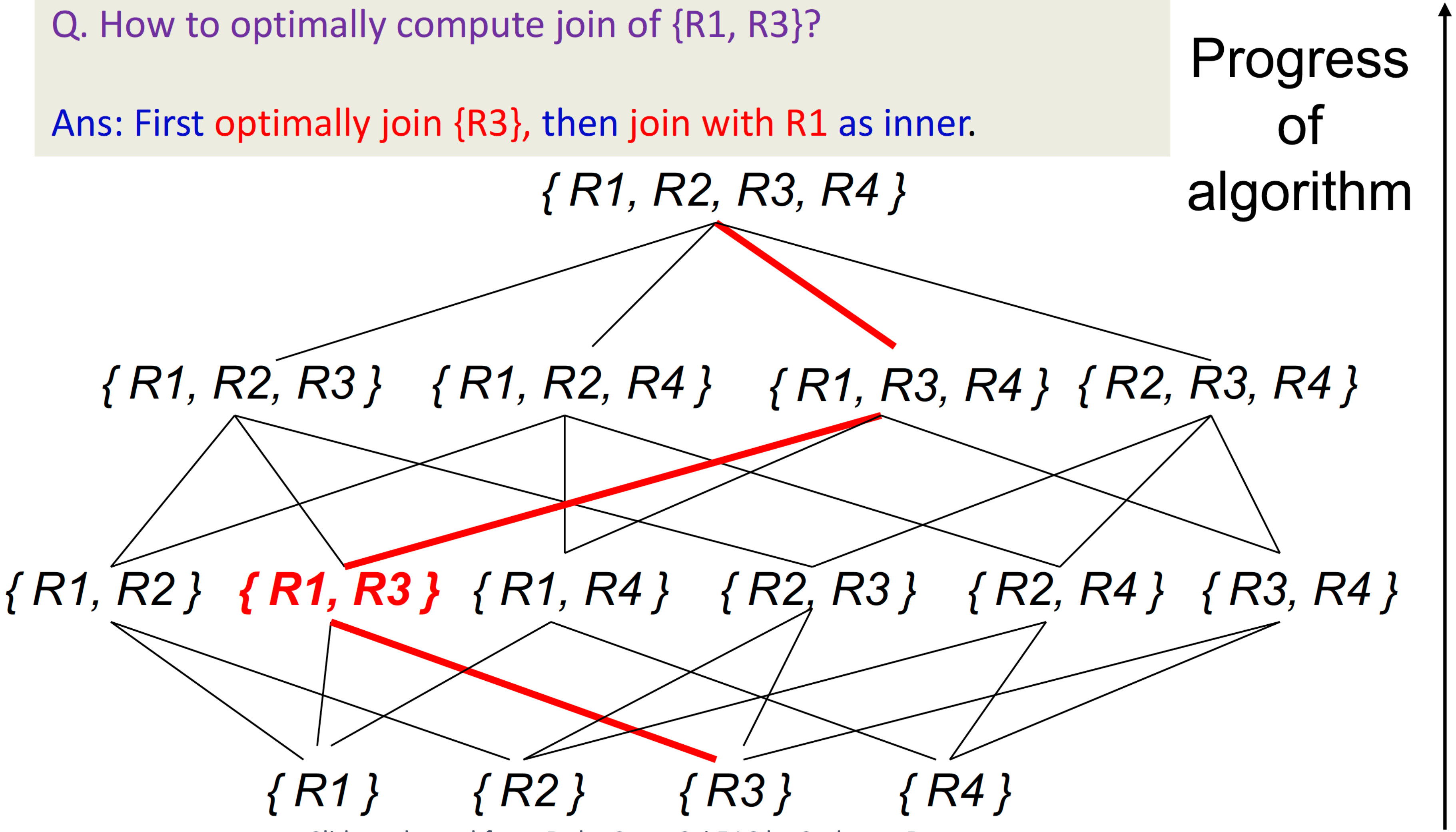
Slides adapted from Duke CompSci 516 by Sudeepa Roy

Selinger Algorithm

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$

Q. How to optimally compute join of {R1, R3}?

Ans: First optimally join {R3}, then join with R1 as inner.



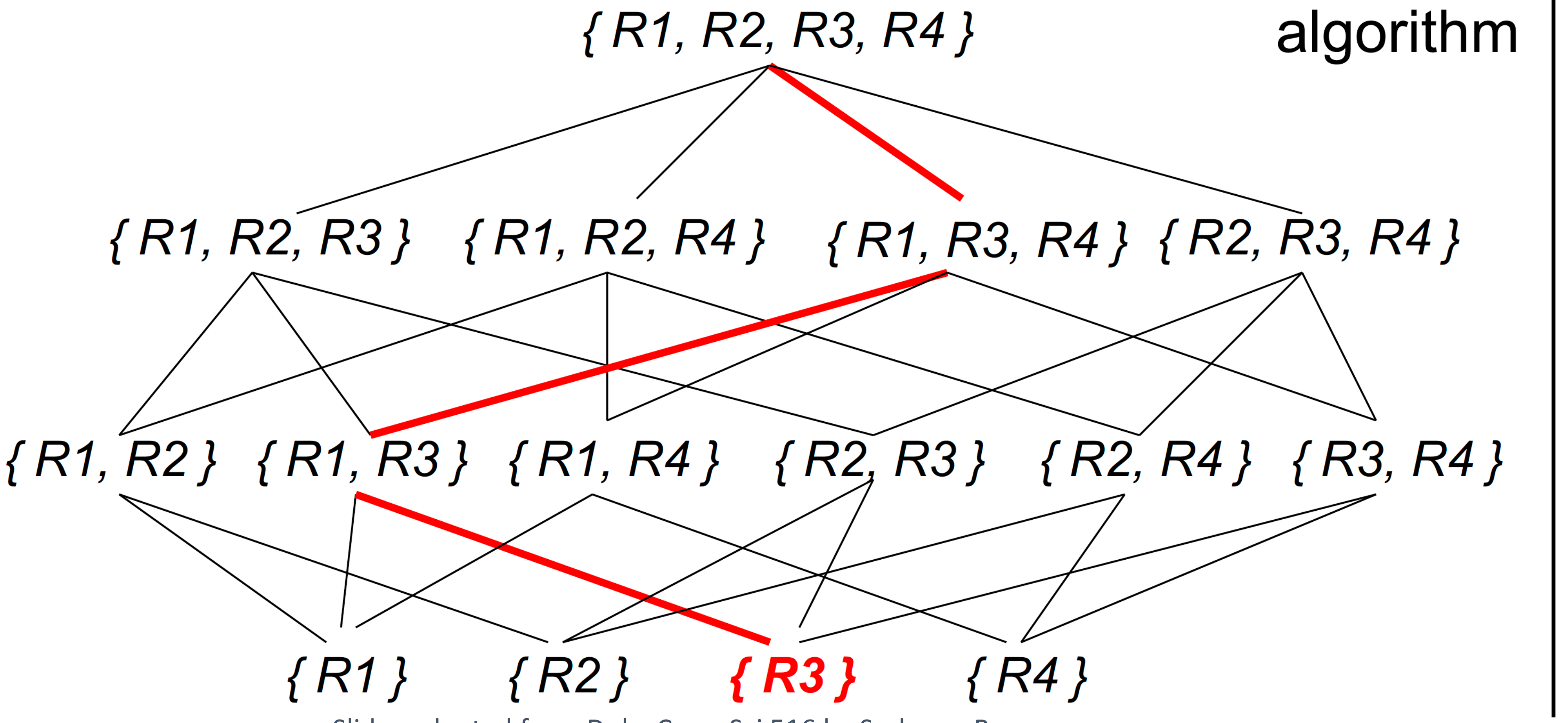
Slides adapted from Duke CompSci 516 by Sudeepa Roy

Selinger Algorithm

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$

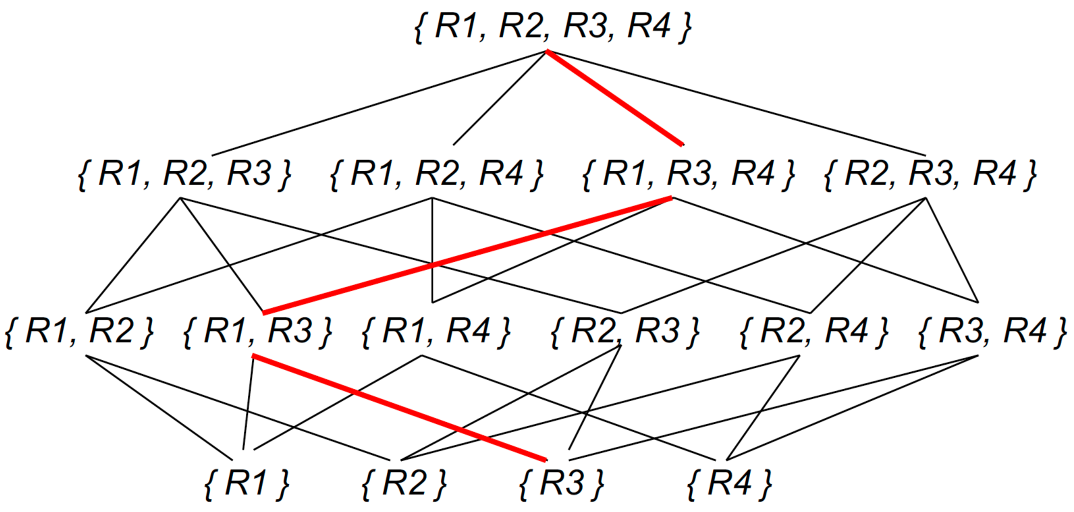
Q. How to optimally compute join of {R3}?

Ans: Single relation – so optimally scan R3.

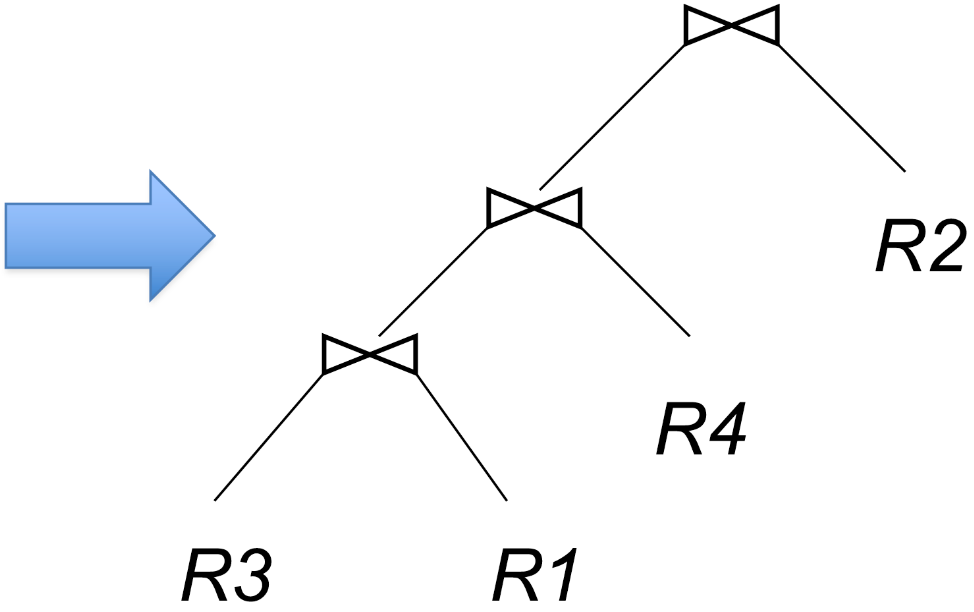


Selinger Algorithm

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Final optimal plan:



NOTE : There is a one-one correspondence between the permutation (R3, R1, R4, R2) and the above left deep plan

Slides adapted from Duke CompSci 516 by Sudeepa Roy

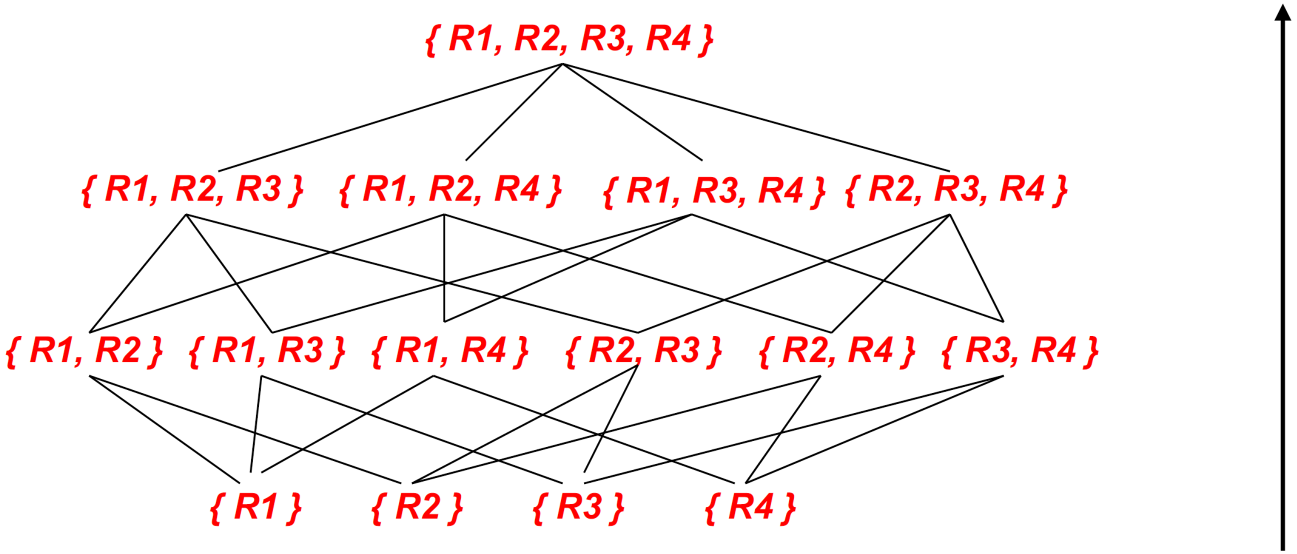
Selinger Algorithm

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$

NOTE:
This is *NOT* done by top-down recursive calls.

- This is done BOTTOM-UP computing the optimal cost of *all* nodes in this lattice only once (dynamic programming).

Progress of algorithm



Is it efficient? ☺

Reduces n! to 2^n

Other optimizations employed too..

Try it yourself

`EXPLAIN` command: Display the execution plan that the PostgreSQL planner generates for the supplied statement.

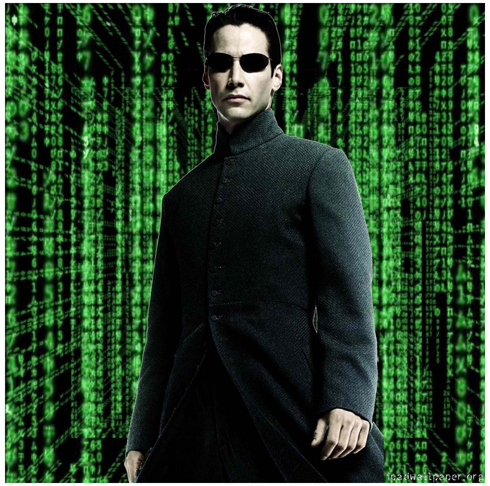
```
EXPLAIN SELECT * FROM foo;

                         QUERY PLAN
---------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

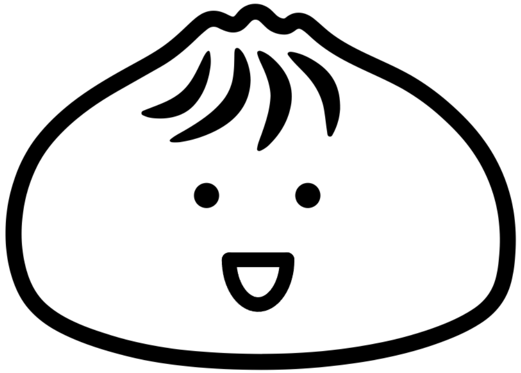
```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                         QUERY PLAN
--------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

Source: https://www.postgresql.org/docs/current/sql-explain.html

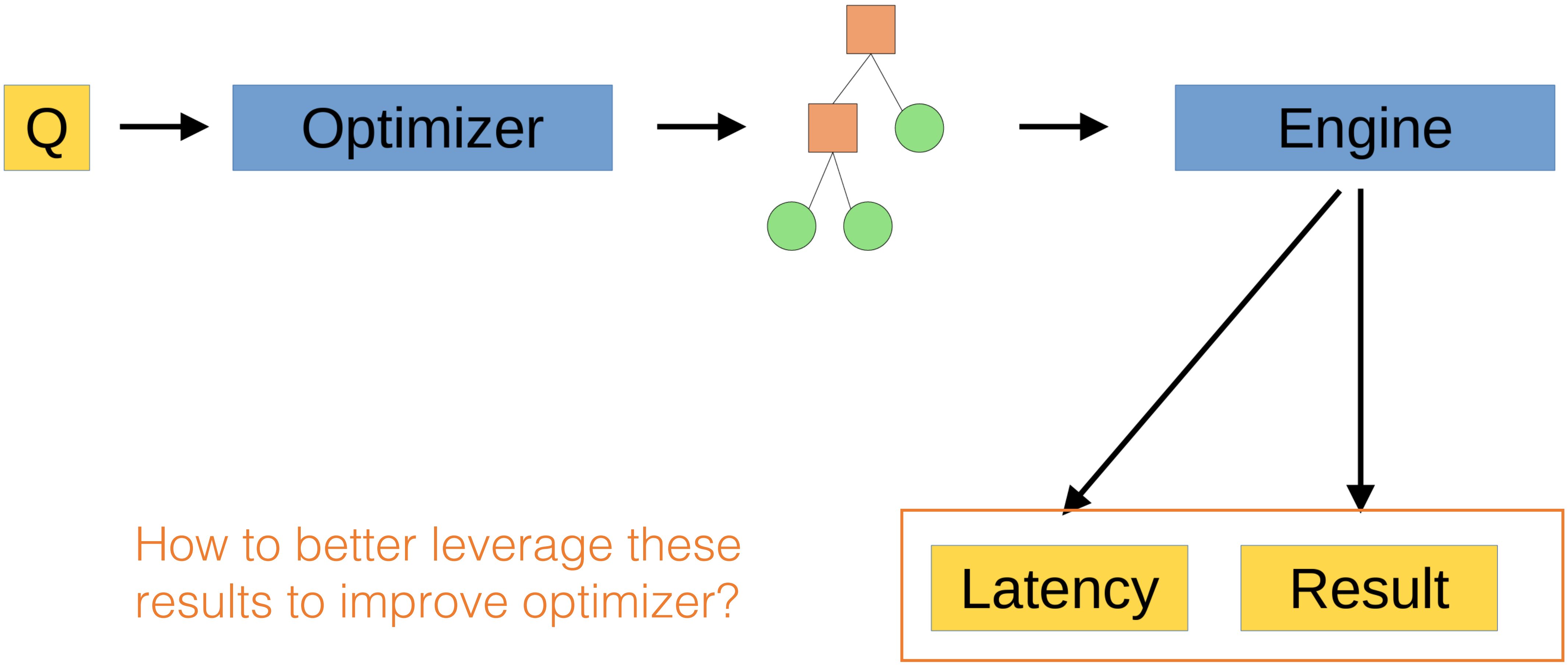
A brief intro to learned query optimizers

Slides adapted from *Machine Learning for Query Optimization … and beyond!* by Ryan Marcus





Query Optimization



How to better leverage these results to improve optimizer?

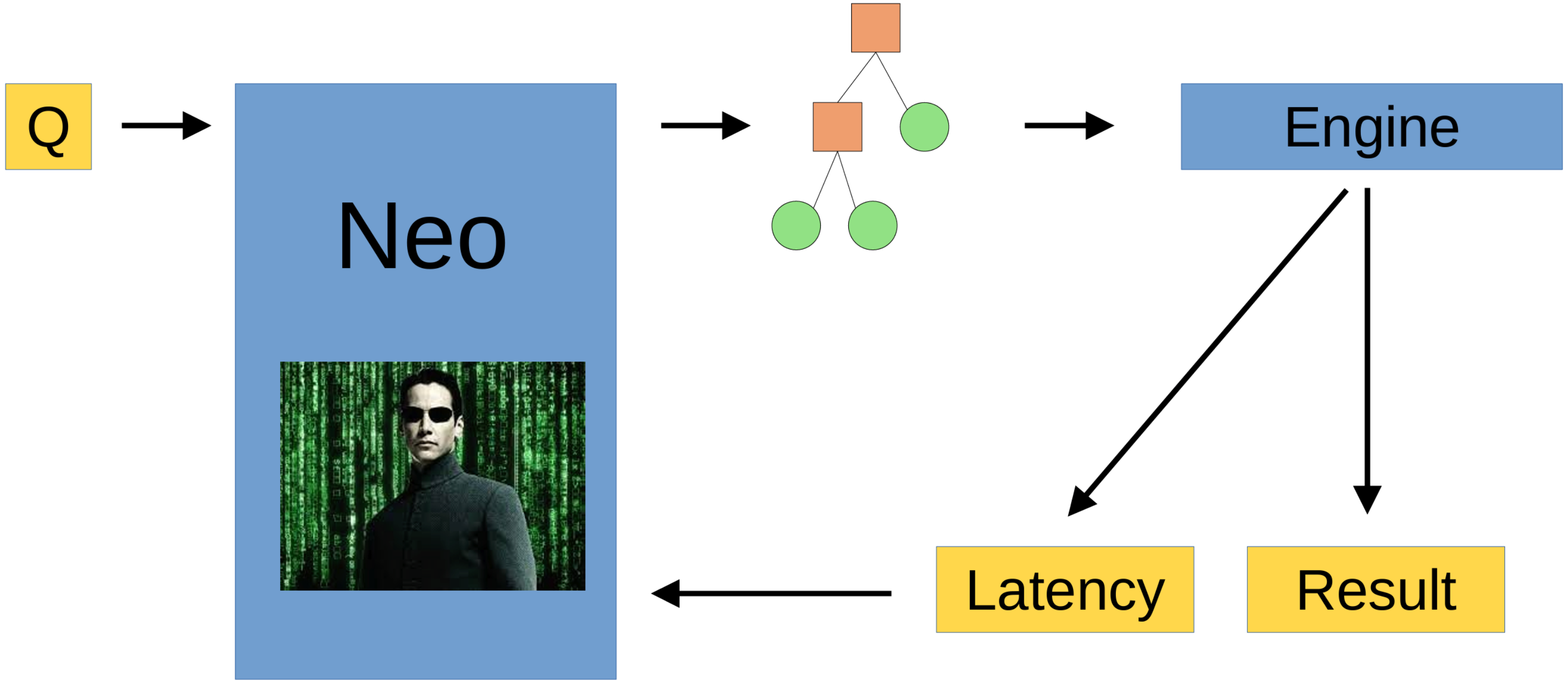
Source: Ryan Marcus

Neo: A Learned Query Optimizer [VLDB'19]

Complete replacement of default query optimizer

First to show we can have *all learned everything*

Deep reinforcement learning guided search



Source: Ryan Marcus

Neo: A Learned Query Optimizer. VLDB '19

Neo: A Learned Query Optimizer [VLDB'19]

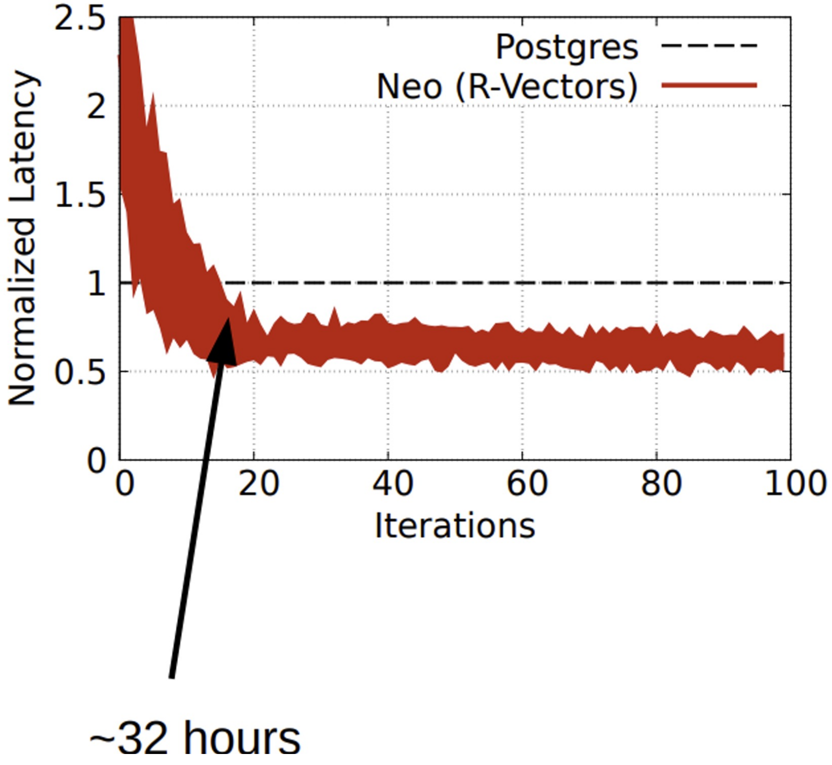
Neo worked great on average but

Sample Inefficiency

- Typically takes > 1 day for pre-train

Brittleness to workload and schema change

- The encoding of cardinality estimate needs retrain



Tail catastrophe

- Deep RL making wrong estimates due to sample inefficiency

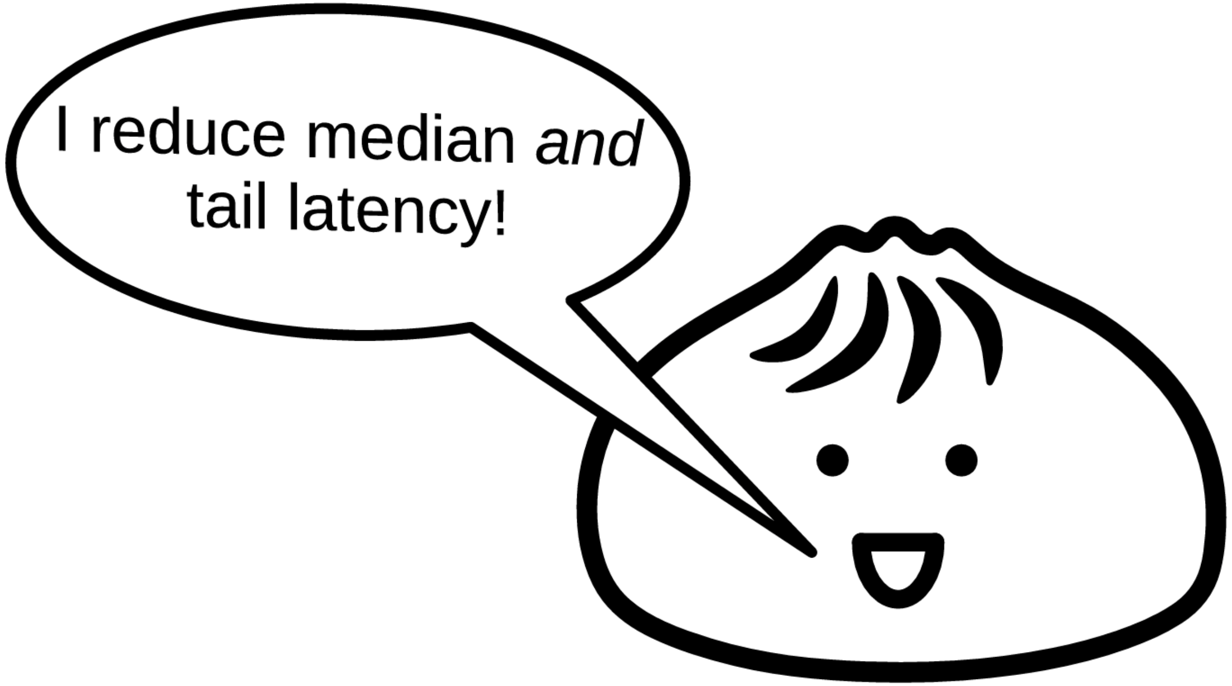
Source: Ryan Marcus

Bao: Making Learned Query Optimization Practical [SIGMOD'21]

Bao: Bandit optimizer

By steering a traditional query optimizer, Bao:

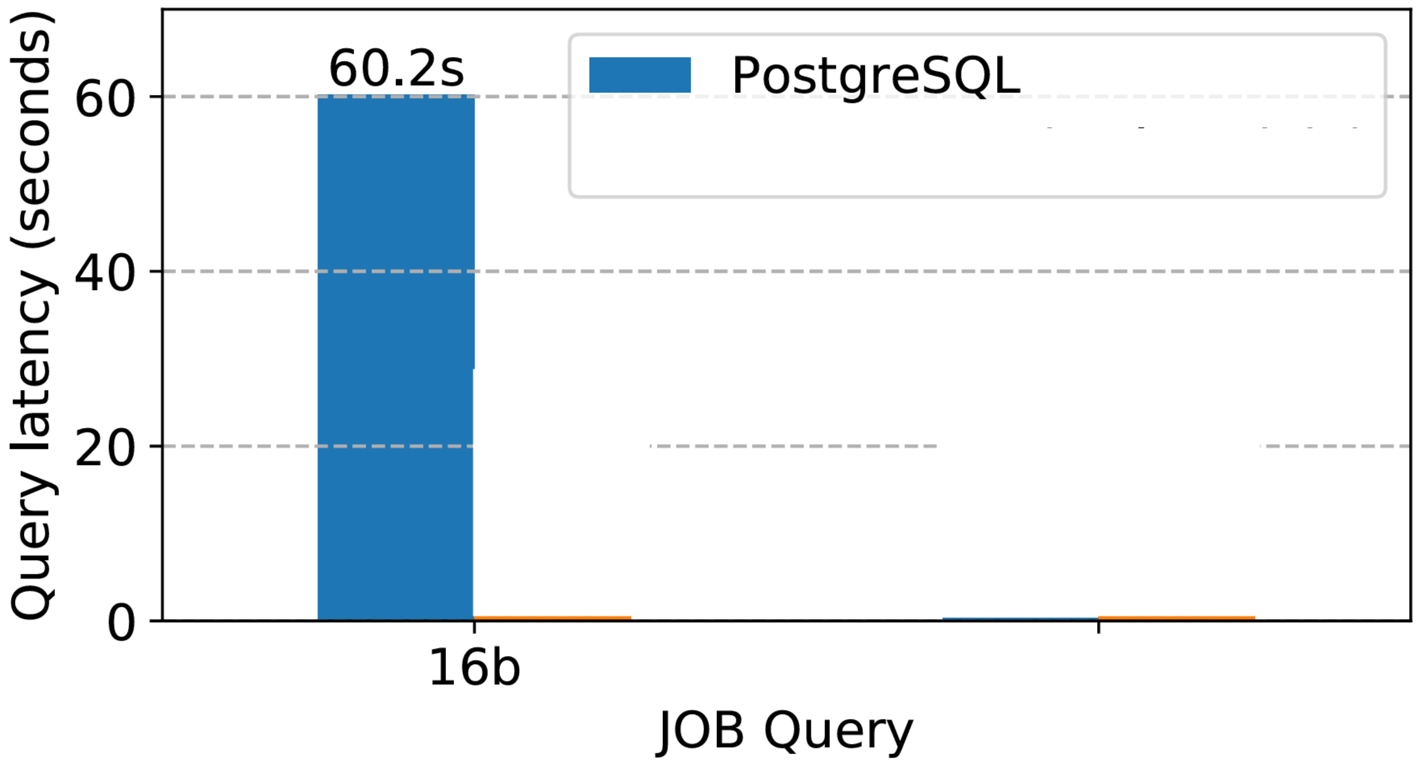
- Outperforms PG after 1 hour of training
- Reduces 99% latency
- Adaptsto changes in workload, schema, and data.



Source: Ryan Marcus

Query Hints

Slow query. Run EXPLAIN.
> Loop join plan,
> Low selectivity

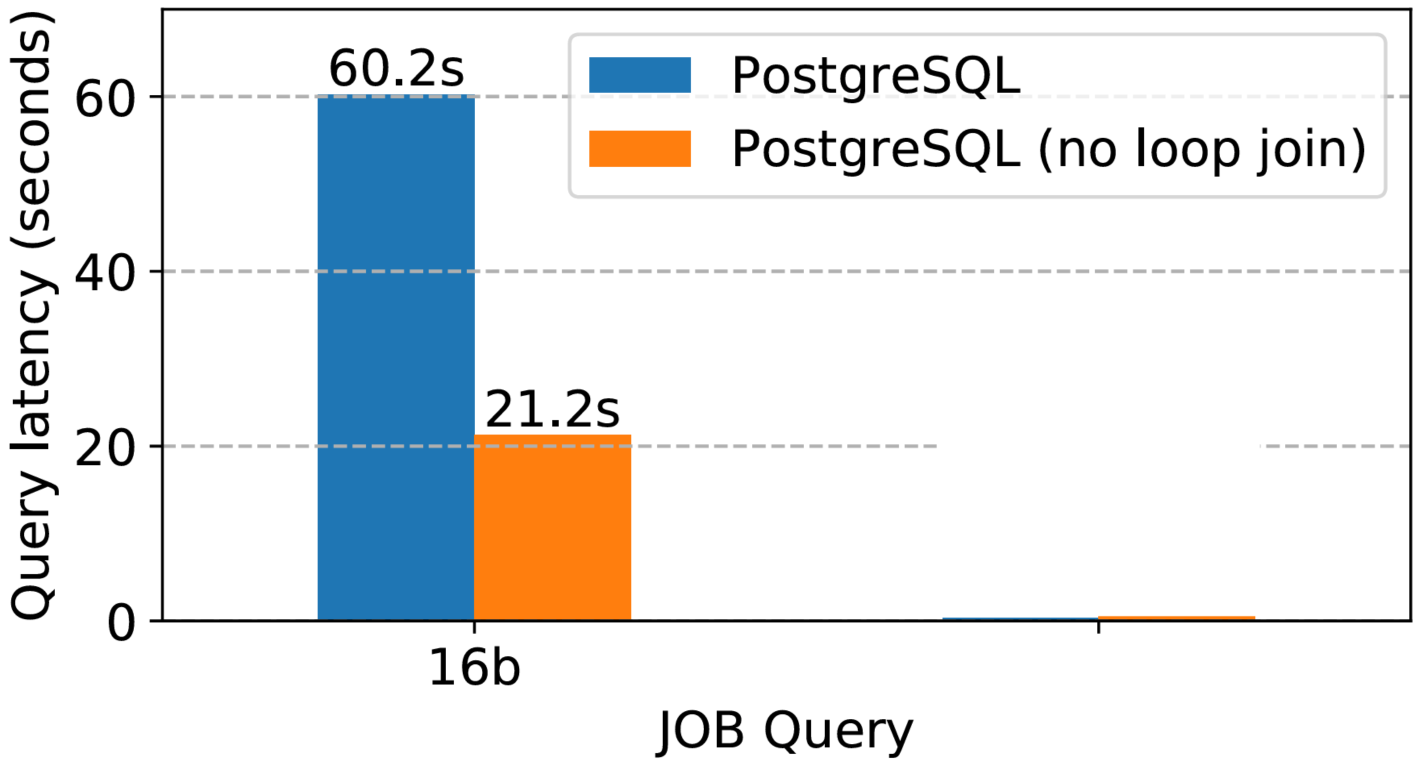


Source: Ryan Marcus

Query Hints

Slow query. Run EXPLAIN.
> Loop join plan,
> Low selectivity

Try disabling loop join
> Huge improvement



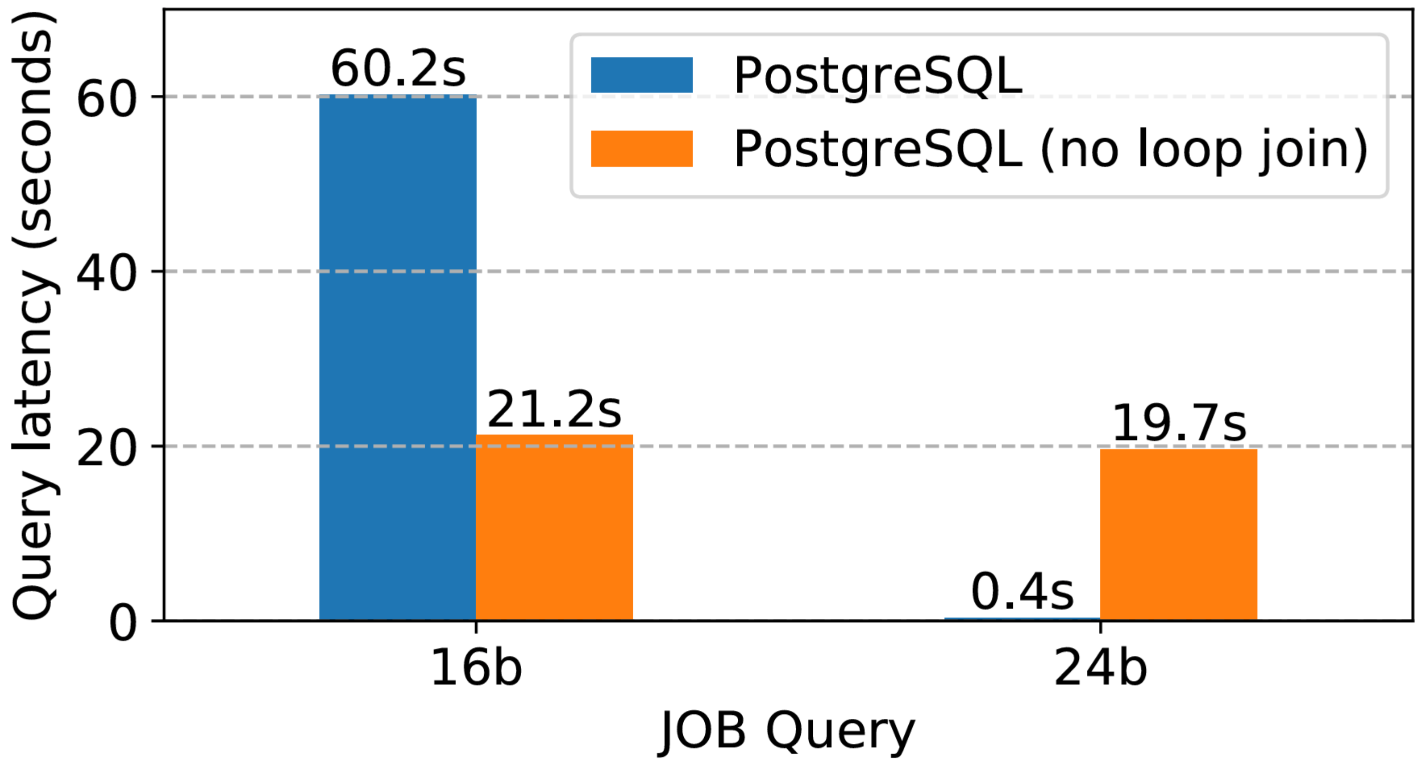
Source: Ryan Marcus

Query Hints

Slow query. Run EXPLAIN.
> Loop join plan,
> Low selectivity

Try disabling loop join
> Huge improvement

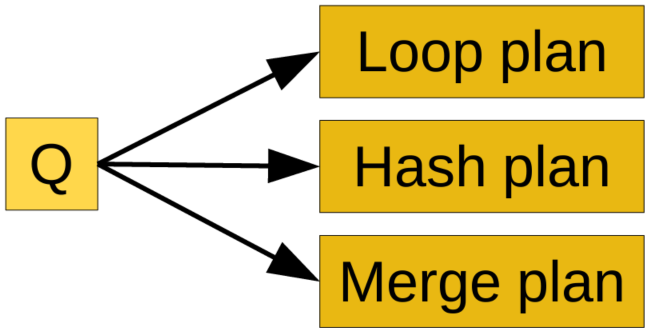
Apply this hint globally
> … other regressions



Source: Ryan Marcus

Bao

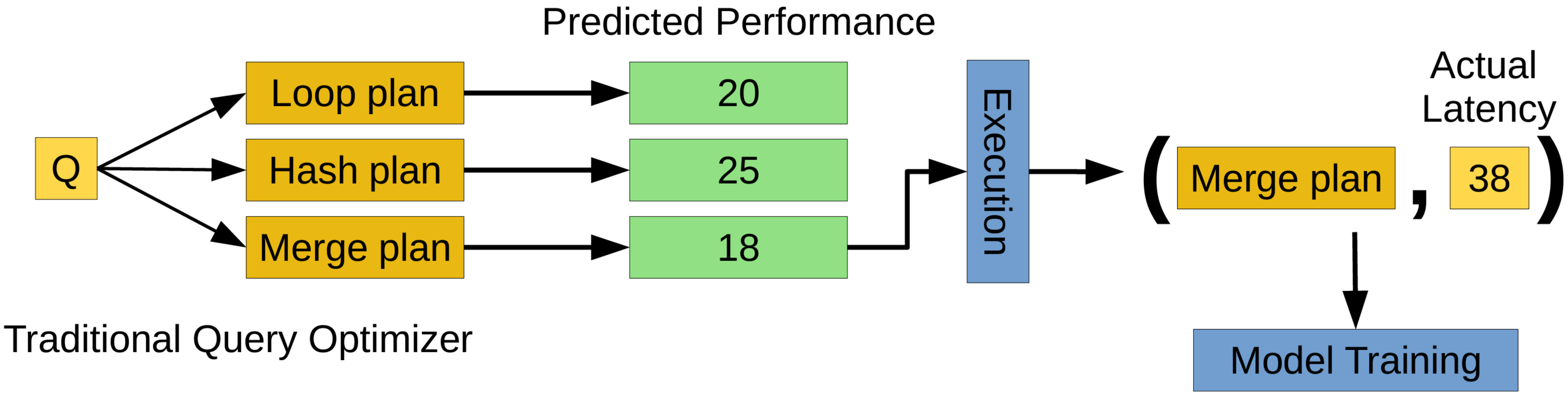
- Bao automatically determines the right hint to use.
- Consider different hints as *arms* in a *contextual multi-armed bandit*



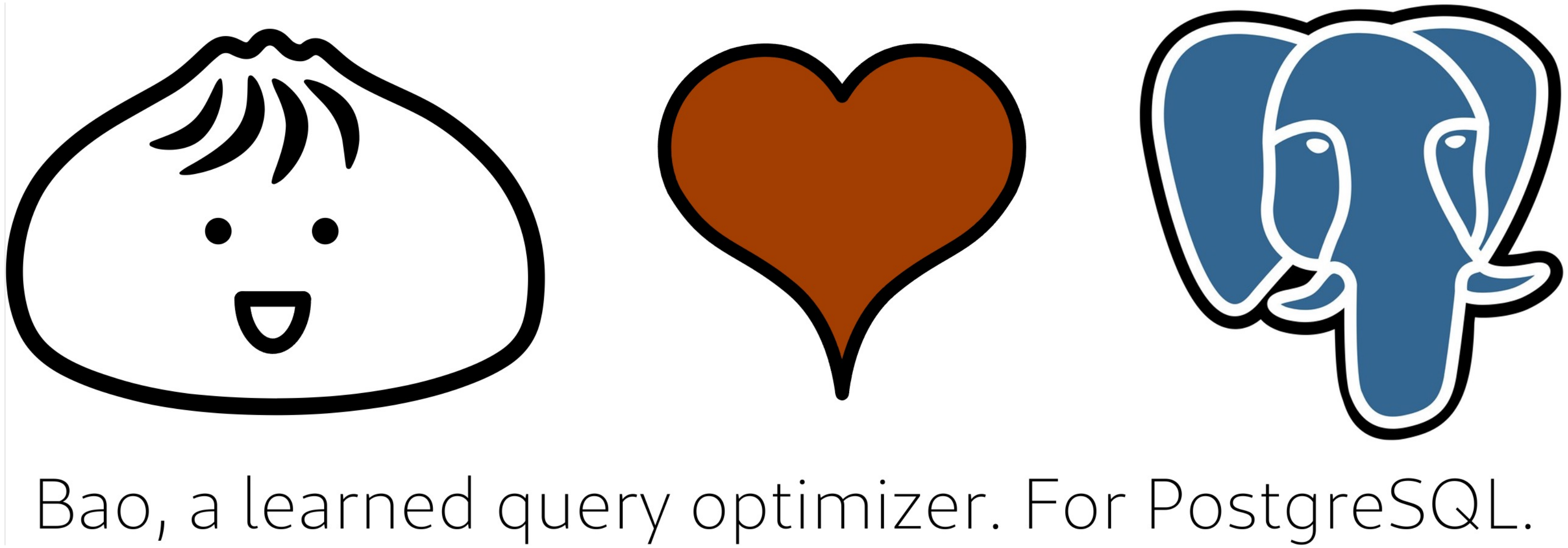
Traditional Query Optimizer

Bao

- Bao automatically determines the right hint to use.
- Consider different hints as *arms* in a *contextual multi-armed bandit*



PostgreSQL Integration



Github: https://github.com/learnedsystems/BaoForPostgreSQL

Moving onto Transactions…

- What are transactions and why are they useful?
- Overview of ACID properties
- Using transactions in SQL

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Reading Materials

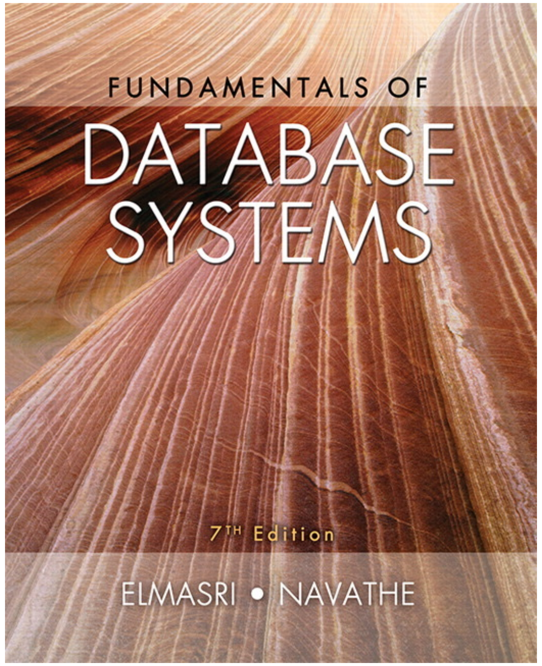
Fundamental of Database Systems (7th Edition)

- Chapter 20 - Introduction to Transaction Processing Concepts and Theory

Supplementary materials

Database Management Systems (Third Edition)

- Chapter 16 – Overview of Transaction Management



Motivation: Concurrent Execution

Single-User System:

- At most one user at a time can use the system.

Multiuser System:

- Many users can access the system concurrently.

Concurrent execution of user programs is essential for good DBMS performance.

- Disk accesses are frequent, and relatively slow
- it is important to keep the CPU busy by working on several user programs concurrently
- We focus on the interleaved processing case (concurrent execution of processes is interleaved in a single CPU) instead of the parallel processing case (processes are concurrently executed in multiple CPUs)

Transactions

A user's program may carry out many operations on the data retrieved from the database

- But the DBMS is only concerned about what data is read/written from/to the database

A transaction is the DBMS's abstract view of a user program

- A sequence of reads and write
- The same program executed multiple times would be considered as different transactions
- Beyond enforcing some integrity constraints, the DBMS does not really understand the semantics of the data (e.g., it does not understand how the interest on a bank account is computed) – it only cares about "read" and "write" sequences

Example

Consider two transactions

```
T1:  BEGIN  A=A+100,  B=B-100  END
T2:  BEGIN  A=1.06*A,  B=1.06*B  END
```

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- However, the net effect must be equivalent to these two transactions running serially in some order

Example

T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END

Consider a possible interleaving (schedule):

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

This is Ok. But what about

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

The DBMS's view of the second schedule

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

Commit and Abort

T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END

- A transaction might commit after completing all its actions
- or it could abort (or be aborted by the DBMS) after executing some actions

Desirable Properties of Transactions

ACID properties:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed.
- **Durability**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Atomicity

A user can think of a transaction as always executing all its actions in one step, or not executing any actions at all

- Users do not have to worry about the effect of incomplete transactions

```
T1:  BEGIN  A=A+100,  B=B-100  END
T2:  BEGIN  A=1.06*A,  B=1.06*B  END
```

Consistency

Each transaction, when run by itself with no concurrent execution of other actions, must preserve the consistency of the database

- e.g., if you transfer money from the savings account to the checking account, the total amount still remains the same

```
T1:  BEGIN  A=A+100,   B=B-100   END
T2:  BEGIN  A=1.06*A,  B=1.06*B  END
```

Isolation

A user should be able to understand a transaction without considering the effect of any other concurrently running transaction

- Even if the DBMS interleaves their actions
- Transaction are “isolated or protected” from other transactions

```
T1:  BEGIN  A=A+100,  B=B-100  END
T2:  BEGIN  A=1.06*A,  B=1.06*B  END
```

Durability

Once the DBMS informs the user that a transaction has been successfully completed, its effect should persist

- even if the system crashes before all its changes are reflected on disk

```
T1:  BEGIN   A=A+100,   B=B-100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

Next, how we maintain all these four properties on a high level

Ensuring Consistency

- User's responsibility to maintain the integrity constraints, as the DBMS may not be able to catch such errors in user program's logic
  - e.g. if the credit is (debit – 1)
- However, the DBMS may be in inconsistent state "during a transaction" between actions
  - which is ok, but it should leave the database at a consistent state when it commits or aborts
- Database consistency follows from transaction consistency, isolation, and atomicity

Ensuring Isolation

- DBMS guarantees isolation
- If T1 and T2 are executed concurrently, either the effect would be T1->T2 or T2->T1 (and from a consistent state to a consistent state)
- But DBMS provides no guarantee on which of these order is chosen
- Often ensured by "locks" but there are other methods too

Ensuring Atomicity

Transactions can be incomplete due to several reasons

- Aborted (terminated) by the DBMS because of some anomalies during execution
    - in that case automatically restarted and executed anew
- The system may crash (e.g., no power supply)
- A transaction may decide to abort itself encountering an unexpected situation
    - e.g., read an unexpected data value or unable to access disks

Ensuring Atomicity

- A transaction interrupted in the middle can leave the database in an inconsistent state
- DBMS has to remove the effects of partial transactions from the database
- DBMS ensures atomicity by “undoing” the actions of incomplete transactions
- DBMS maintains a “log” of all changes to do so

Ensuring Durability

- The log also ensures durability
- If the system crashes before the changes made by a completed transactions are written to the disk, the log is used to remember and restore these changes when the system restarts
- "recovery manager"
    - takes care of atomicity and durability

Using Transactions in SQL

- SQL allows the programmer to group several statements in a single *transaction*
- Either all operations are performed or none are
- A single SQL statement is always considered to be atomic.

```
START TRANSACTION

UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;

UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;

COMMIT;
```

Marks beginning of transaction

Causes transaction to end successfully

Using Transactions in SQL

- ROLLBACK causes the transaction to abort and undo any changes

We find that there are insufficient funds to make transfer

```
START TRANSACTION

UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;



ROLLBACK;
```

Using Transactions in SQL

```
SET [GLOBAL | SESSION] TRANSACTION
        transaction_characteristic [,
transaction_characteristic] ...

transaction_characteristic: {
        ISOLATION LEVEL level
        | access_mode }

level: {
        REPEATABLE READ
       | READ COMMITTED
       | READ UNCOMMITTED
       | SERIALIZABLE}

access_mode: {
        READ WRITE
        | READ ONLY }
```

Using Transactions in SQL

```
SET [GLOBAL | SESSION] TRANSACTION
        transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic: {
        ISOLATION LEVEL level
        | access_mode }

level: {
        REPEATABLE READ
        | READ COMMITTED
        | READ UNCOMMITTED
        | SERIALIZABLE}

access_mode: {
        READ WRITE
         | READ ONLY }
```

Isolation Levels

- With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
- However, if any transaction executes at a lower level, then serializability may be violated.

Using Transactions in SQL

```
SET [GLOBAL | SESSION] TRANSACTION
        transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic: {
        ISOLATION LEVEL level
        | access_mode }

level: {
        REPEATABLE READ
        | READ COMMITTED
        | READ UNCOMMITTED
        | SERIALIZABLE}

access_mode: {
        READ WRITE
        | READ ONLY }
```

Access Mode

- The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.

Read-only transactions

- Transactions that only read data and do not write can be executed in parallel
- Tell SQL system before running transaction:

```
SET TRANSACTION READ ONLY;
```

Dirty reads

Reading data written by a transaction that has not yet committed

Consider this seat selection example:

1. Find available seat and reserve by setting *seatStatus* to ‘occupied’
2. Ask customer for approval of seat
   a. If so, commit
   b. If not, release seat by setting seatStatus to ‘available’ and repeat Step (1)

Dirty read

- If we allow dirty reads, this can happen

time

User 1 finds seat 22A empty and reserves it (22A is occupied)

User 2 is told that seat 22A is already occupied (dirty read)

User 1 disapproves the 22A reservation

Dirty reads

- If this result is acceptable, the transaction processing can be done faster
  - DBMS does not have to prevent dirty reads
  - Allows more parallelism
- Tell SQL system:

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```