CS 600.316/416
Database Systems

Lecture 11, March 10th, 2014.

Advanced Query Optimization and Physical Database Design (i)

Syllabus Checkpoint

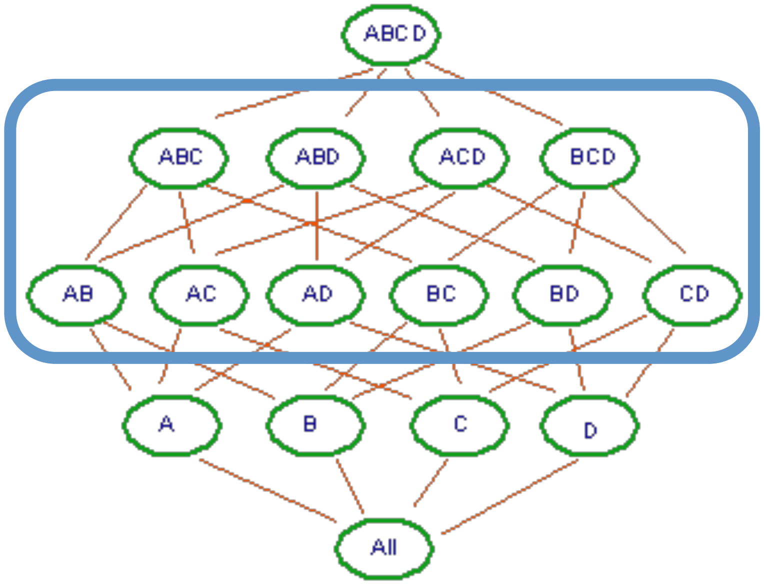
- Storage layer
  - File and page storage
  - Indexing
- Query processing and optimization
  - Join and sort algorithms
  - Dynamic programming based query optimization
  - Selectivity estimation
- Data analysis

Datacubes

- Think “combinatorics over aggregates”
- Seminal paper by Gray et. al [ICDE 96]
  - Defines the semantics of the operator
  - Years of algorithms and implementations followed

Processing decision: what “summary” tables or views, do we materialize?

Materialize upper lattice nodes, compute lower ones on the fly



Scaling up query processing via query properties (rather than data)

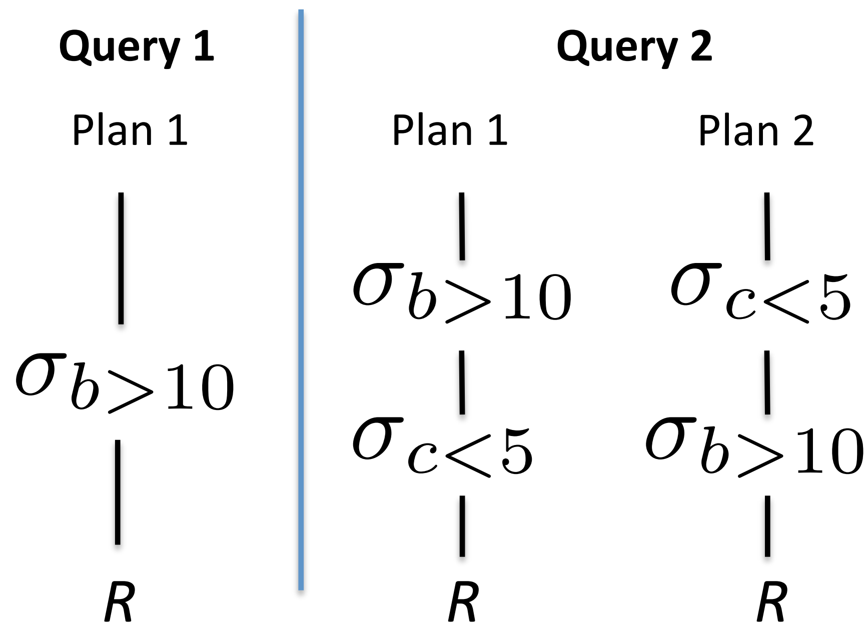
MULTIQUERY OPTIMIZATION (MQO)

Shared Query Processing

- We've looked at techniques to reuse work done in a DBMS, primarily caching:
  - Query results: reuses QP work
  - Buffer pool: reuses disk work
  - Plans: reuses parser, compiler, optimizer work
- These are all reactive and opportunistic
  - We can use heuristics to determine what to add to the cache
- Shared query processing focuses on reducing the work done by a QP *by design*, rather than opportunistically
  - By building shared query plans across multiple queries that "factor" out common work

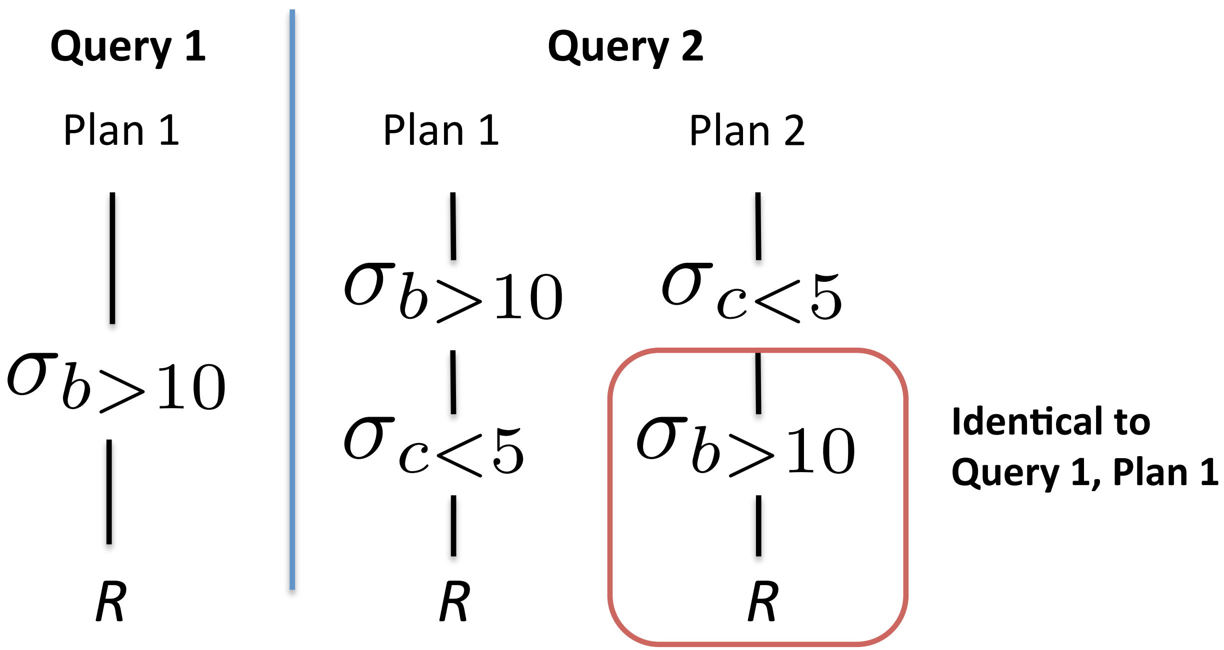
Shared Query Processing Example

- Consider the queries:
  1. select * from R where R.b > 10
  2. select * from R where R.b > 10 and R.c < 5



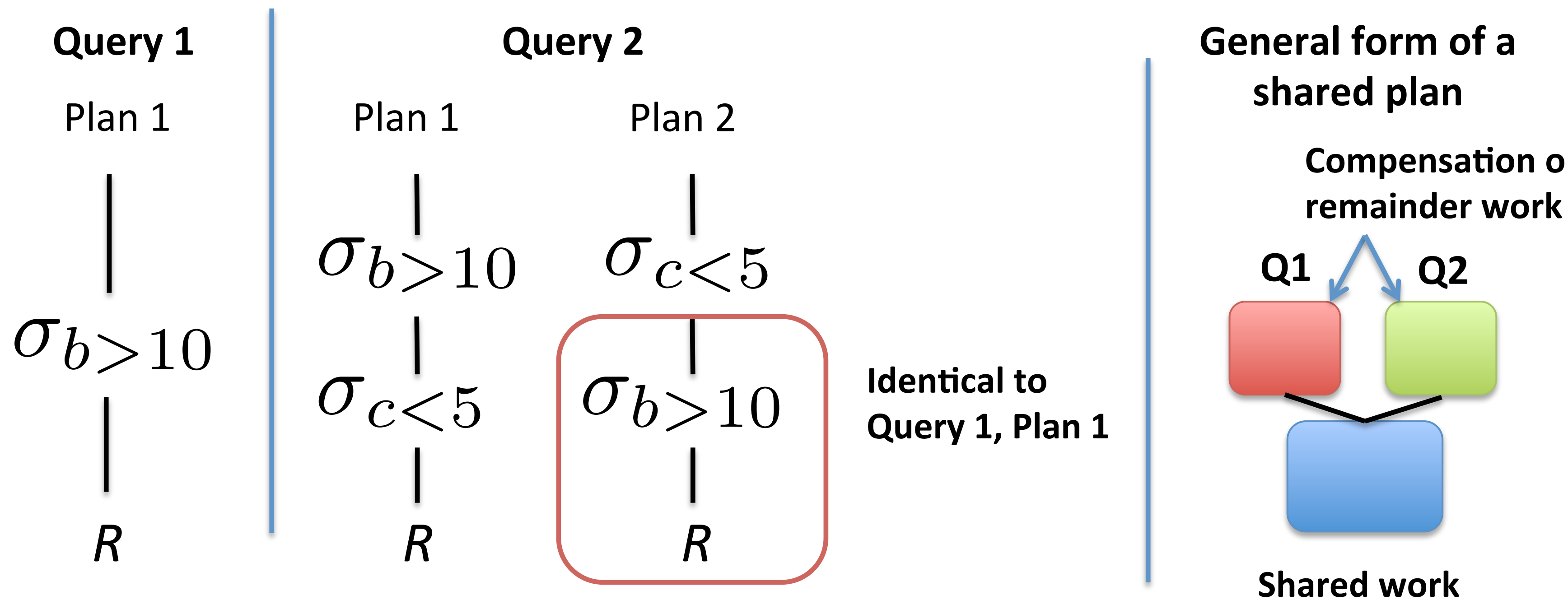
Shared Query Processing Example

- Consider the queries:
  1. select * from R where R.b > 10
  2. select * from R where R.b > 10 and R.c < 5



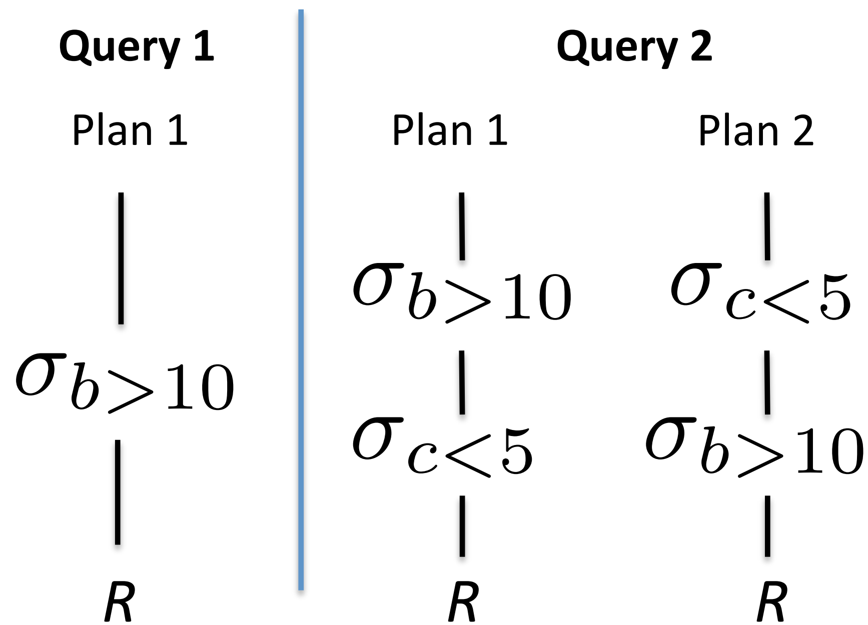
Shared Query Processing Example

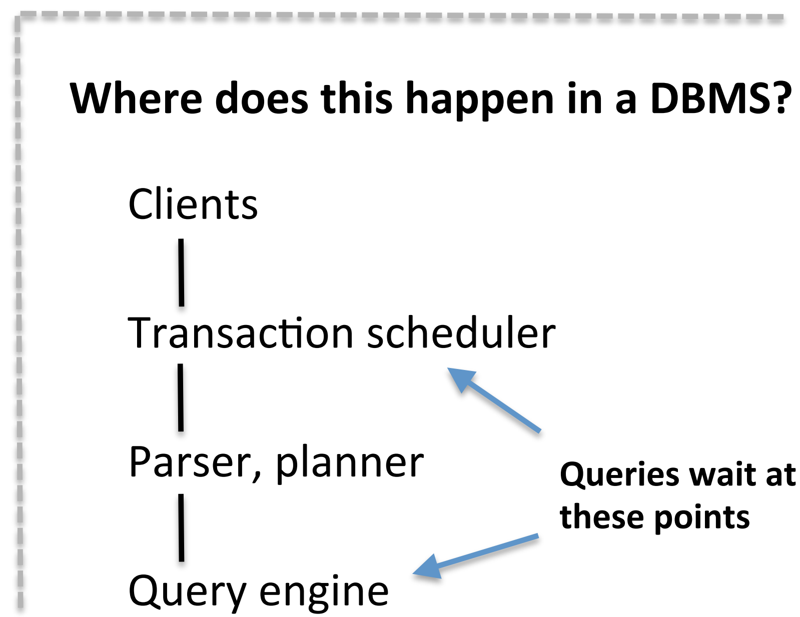
- Consider the queries:
  1. select * from R where R.b > 10
  2. select * from R where R.b > 10 and R.c < 5



Shared Query Processing Example

- Consider the queries:
  1. select * from R where R.b > 10
  2. select * from R where R.b > 10 and R.c < 5





Common Subexpressions

- Sharing requirements relational operators
  - Projections

$$\{\pi_{A_1}(Q), \pi_{A_2}(Q)\} = \begin{cases} \{\pi_{A_1 \cap A_2}(Q), \pi_{A1-A2}(Q), \pi_{A2-A1}(Q)\} \\ \qquad\qquad\qquad\qquad \text{when } A1 \cap A2 \neq \emptyset \\ \{\pi_{A_1}(Q), \pi_{A2}(Q)\} \text{ otherwise} \end{cases}$$

Assume identical for simplicity

Common Subexpressions

- Sharing requirements relational operators
  - Projections

Shared | Remainder 1 | Remainder 2

$$\{\pi_{A_1}(Q), \pi_{A_2}(Q)\} = \begin{cases} \{\pi_{A_1 \cap A_2}(Q), \pi_{A1-A2}(Q), \pi_{A2-A1}(Q)\} \\ \qquad \text{when } A1 \cap A2 \neq \emptyset \\ \{\pi_{A_1}(Q), \pi_{A2}(Q)\} \text{ otherwise} \end{cases}$$

Common Subexpressions

- Sharing requirements relational operators
  - Projections

$$\{\pi_{A_1}(Q), \pi_{A_2}(Q)\} = \begin{cases} \{\pi_{A_1 \cap A_2}(Q), \pi_{A1-A2}(Q), \pi_{A2-A1}(Q)\} \\ \qquad \text{when } A1 \cap A2 \neq \emptyset \\ \{\pi_{A_1}(Q), \pi_{A2}(Q)\} \text{ otherwise} \end{cases}$$

  - Joins

Shared | Remainder 1 | Remainder 2

$$\{Q_1 \bowtie_{\theta_1} Q_2, Q_1 \bowtie_{\theta_2} Q_2\} = \begin{cases} \{Q_C \leftarrow Q_1 \bowtie_{\theta_c} Q_2, \sigma_{\theta_1}(Q_C), \sigma_{\theta_2}(Q_C)\} \\ \qquad \text{when } \mathit{shared}(\theta_1, \theta_2) \\ \{Q_1 \bowtie_{\theta_1} Q_2, Q_1 \bowtie_{\theta_2} Q_2\} \text{ otherwise} \end{cases}$$

Common Subexpressions

- Sharing requirements relational operators
  - Join example

$$\{Q_1 \bowtie_{\theta_1} Q_2, Q_1 \bowtie_{\theta_2} Q_2\} = \begin{cases} \{Q_C \leftarrow Q_1 \bowtie_{\theta_c} Q_2, \sigma_{\theta_1}(Q_C), \sigma_{\theta_2}(Q_C)\} \\ \qquad \text{when } shared(\theta_1, \theta_2) \\ \{Q_1 \bowtie_{\theta_1} Q_2, Q_1 \bowtie_{\theta_2} Q_2\} \text{ otherwise} \end{cases}$$

$$\begin{array}{l} \{Q_1 \bowtie_{Q1.A=Q2.B \,\wedge\, Q1.C<Q2.D} Q_2, \\ \quad Q_1 \bowtie_{Q1.A=Q2.B \,\wedge\, Q1.C>Q2.E} Q_2\} \end{array} = \begin{array}{l} \{Q_C \leftarrow Q_1 \bowtie_{Q1.A=Q2.B} Q_2, \\ \quad \sigma_{Q1.C<Q2.D}(Q_C), \\ \quad \sigma_{Q1.C>Q2.E}(Q_C)\} \end{array}$$

Common Subexpressions

- Sharing requirements relational operators
  - Join example

$$\{Q_1 \bowtie_{\theta_1} Q_2, Q_1 \bowtie_{\theta_2} Q_2\} = \begin{cases} \{Q_C \leftarrow Q_1 \bowtie_{\theta_c} Q_2, \sigma_{\theta_1}(Q_C), \sigma_{\theta_2}(Q_C)\} \\ \qquad \text{when } shared(\theta_1, \theta_2) \\ \{Q_1 \bowtie_{\theta_1} Q_2, Q_1 \bowtie_{\theta_2} Q_2\} \text{ otherwise} \end{cases}$$

$$\{Q_1 \bowtie_{Q1.A=Q2.B \wedge Q1.C<Q2.D} Q_2, \; Q_1 \bowtie_{Q1.A=Q2.B \wedge Q1.C>Q2.E} Q_2\} = \{Q_C \leftarrow Q_1 \bowtie_{Q1.A=Q2.B} Q_2, \; \sigma_{Q1.C<Q2.D}(Q_C), \; \sigma_{Q1.C>Q2.E}(Q_C)\}$$

Common Subexpressions: Aggregation

- Aggregates, and group-bys
  - Let's recap some rewrite rules first!

$$\sum\left[\pi_A(Q_1) \cup \pi_A(Q_2) \cup \pi_A(Q_3)\right] =$$

$$\sum\left[\sum \pi_A(Q_1) + \sum \pi_A(Q_2) + \sum \pi_A(Q_3)\right]$$

  - In SQL:

```
select sum(a) from
(select a from Q1 union
 select a from Q2 union
 select a from Q3)
=
select sum(a) from
(select sum(a) as a from Q1 union
 select sum(a) as a from Q2 union
 select sum(a) as a from Q3)
```

Common Subexpressions: Aggregation

- Aggregates, and group-bys
  - Let's recap some rewrite rules first!

$$\sum [\pi_A(Q_1) \cup \pi_A(Q_2) \cup \pi_A(Q_3)] =$$

$$\sum [\sum \pi_A(Q_1) + \sum \pi_A(Q_2) + \sum \pi_A(Q_3)]$$

  - In SQL:

```
select sum(a) from
(select a from Q1 union
 select a from Q2 union
 select a from Q3)
=
select sum(a) from
(select sum(a) as a from Q1 union
 select sum(a) as a from Q2 union
 select sum(a) as a from Q3)
```

Why is this more efficient? →

Common Subexpressions: Aggregation

- Aggregates, and group-bys
  - Let's recap some rewrite rules first!

$$\sum [\pi_A(Q_1) \cup \pi_A(Q_2) \cup \pi_A(Q_3)] =$$

$$\sum [\sum \pi_A(Q_1) + \sum \pi_A(Q_2) + \sum \pi_A(Q_3)]$$

  - In SQL:

```
select sum(a) from
(select a from Q1 union
 select a from Q2 union
 select a from Q3)
=
select sum(a) from
(select sum(a) as a from Q1 union
 select sum(a) as a from Q2 union
 select sum(a) as a from Q3)
```

Each subquery returns fewer rows, i.e., smaller intermediates →

Common Subexpressions: Aggregation

- Aggregates, and group-bys
  - This technique is called ***partial aggregation***

$$\sum \left[\pi_A(Q_1) \cup \pi_A(Q_2) \cup \pi_A(Q_3)\right] = \sum \left[\sum \pi_A(Q_1) + \sum \pi_A(Q_2) + \sum \pi_A(Q_3)\right]$$

  - Applies to other ***distributive*** aggregates

$$\min \left[\pi_A(Q_1) \cup \pi_A(Q_2) \cup \pi_A(Q_3)\right] = \min \left(\min \pi_A(Q_1), \min \pi_A(Q_2), \min \pi_A(Q_3)\right)$$

    - See also algebraic and holistic aggregates offline
  - How is this helpful for shared query processing?
    - Exploit shared parts, and groups (for group-by-aggregation)

Common Subexpressions: Aggregation

- Aggregates

$$\left\{\sum Q1, \sum Q2, \sum Q3\right\} = \begin{cases} \{Q_C \leftarrow \sum Q_{shared\_parts}, \\ \quad Q_C + \sum Q_{rem1}, \\ \quad Q_C + \sum Q_{rem2}, \\ \quad Q_C + \sum Q_{rem3} \ \} \\ \qquad\qquad\qquad \text{when } Q_{shared\_parts} \neq \emptyset \\ \{\sum Q1, \sum Q2, \sum Q3\} \ \text{otherwise} \end{cases}$$

- Example

```
1. select sum(a) from R where R.b < 5 or R.c > 10
2. select sum(a) from R where R.b < 5 or R.d < 2
3. select sum(a) from R where R.e = 10 or R.b < 5
```

Common Subexpressions: Aggregation

- Aggregates

$$\left\{\sum Q1, \sum Q2, \sum Q3\right\} = \begin{cases} \{Q_C \leftarrow \sum Q_{shared\_parts}, \\ \quad Q_C + \sum Q_{rem1}, \\ \quad Q_C + \sum Q_{rem2}, \\ \quad Q_C + \sum Q_{rem3}\ \} \\ \qquad\qquad \text{when } Q_{shared\_parts} \neq \emptyset \\ \{\sum Q1, \sum Q2, \sum Q3\} \ \text{ otherwise} \end{cases}$$

- Example

```
1. select sum(a) from R where R.b < 5 or R.c > 10
2. select sum(a) from R where R.b < 5 or R.d < 2
3. select sum(a) from R where R.e = 10 or R.b < 5
=>
C. Qc = select sum(a) from R where R.b < 5
1. select Qc + sum(a) from R where R.c > 10
2. select Qc + sum(a) from R where R.d < 2
3. select Qc + sum(a) from R where R.e = 10
```

Query Containment

- So far, we have looked at equality predicates alone
- Consider these queries:

```
1. select sum(a) from R where R.b < 10
2. select sum(a) from R where R.b < 15

=>
C, 1. select sum(a) from R where R.b < 10
2.    select C + sum(a) from R where R.b between 10 and 15
```

- We can say query 2 subsumes query 1 above
- In general this is the query containment problem: *"how can we determine if, for all databases, all of query A's results will be contained in query B's results"*
- Query equivalence is then mutual containment

Query and Predicate Indexing

- We can build indexes to assist with shared query processing
- Key idea: index the queries rather than the data
  - Lets us return the set of queries satisfied by a single tuple
  - Need a "comparison function" over queries
  - Useful for long-running queries

MQO Algorithm Overview

- Given a query workload, ***W*** **= {Q_1…Q_N}**
- Find candidate CSEs, ***C***, in ***W***
- Build a multiquery plan that exploits CSEs
  - A single DAG that represents all of ***W***, where the DAG includes ***C***
- Optimize the multiquery plan, extending standard dynamic programming techniques

MQO Algorithm Overview

- Given a query workload, ***W*** **= {$Q_1 \ldots Q_N$}**
- **Find candidate CSEs, *C*, in *W***
  - Potentially very expensive (e.g. do we want to compute commonality of all subsets of the workload, that is its powerset?)
- Build a multiquery plan that exploits CSEs
  - A single DAG that represents all of ***W***, where the DAG includes ***C***
- Optimize the multiquery plan, extending standard dynamic programming techniques

MQO Algorithm Overview

- Given a query workload, ***W*** **= {Q_1…Q_N}**
- **Find candidate CSEs, *C*, in *W***
  - Potentially very expensive (e.g. do we want to compute commonality of all subsets of the workload, that is its powerset?)
- Build a multiquery plan that exploits CSEs
  - A single DAG that represents all of ***W***, where the DAG includes ***C***
- Optimize the multiquery plan, extending standard dynamic programming techniques

MQO Algorithm Overview

- Given a query workload, ***W*** **= {Q_1…Q_N}**
- **Find candidate CSEs, *C*, in *W***
  - Potentially very expensive (e.g. do we want to compute commonality of all subsets of the workload, that is its powerset?)
- Build a multiquery plan that exploits CSEs
  - A single DAG that represents all of ***W***, where the DAG includes ***C***
- Optimize the multiquery plan, extending standard dynamic programming techniques

- **Question: why do we use candidate CSEs and a multiquery plan up front, rather than keeping track of commonality during regular single query optimization?**

MQO Algorithm Overview

- Given a query workload, ***W*** **=** $\{Q_1 \ldots Q_N\}$
- **Find candidate CSEs, *C*, in *W***
  - Potentially very expensive (e.g. do we want to compute commonality of all subsets of the workload, that is its powerset?)
- Build a multiquery plan that exploits CSEs
  - A single DAG that represents all of ***W***, where the DAG includes ***C***
- Optimize the multiquery plan, extending standard dynamic programming techniques

- **Question: why do we use candidate CSEs and a multiquery plan up front, rather than keeping track of commonality during regular single query optimization?**
  - **Highly localized pruning may eliminate CSEs during single query optimization**

Signature-Based Query Matching

- Query signatures provide a coarse-grained matching heuristic to indicate if two queries have any commonality
- We can use this in a simple candidate generation algorithm:
  1. initialize empty candidate pool
  2. for i = 2 to n
  3. generate CSE candidates for query subsets of size i by combining subsets from previous round and verifying commonality
  4. prune subsets with no commonality
  5. heuristically prune candidates to add to the pool

Matching n-sets

$\{Q_1, Q_2, \ldots, Q_n\}$

Matching triples

$\{Q_1, Q_2, Q_5\}$ $\{Q_6, Q_{10}, Q_{13}\}$

Matching pairs

$\{Q_1, Q_2\}$ $\{Q_6, Q_{10}\}$

$\{Q_1, Q_2, Q_3 \ldots Q_n\}$

Signature-Based Query Matching

- What is a suitable query signature?

Operator	Table Signature
Table/View (t)	$S_t = [\mathrm{F}; t]$
Select (σ)	$S_{\sigma(e)} = S_e$, *if* $G_e = \mathrm{F}$
Project (π)	$S_{\pi(e)} = S_e$, *if* $G_e = \mathrm{F}$
Join ($\bowtie$)	$S_{e_1 \bowtie e_2} = [\mathrm{F}; T_{e_1} \cup T_{e_2}]$, *if* $G_{e_1} = G_{e_2} = \mathrm{F}$
Group-by (γ)	$S_{\gamma(e)} = [\mathrm{T}; T_e]$, *if* $G_e = \mathrm{F}$

- Two queries ***may*** have a common subexpression iff their signatures match and they are join-compatible

Signature-Based Query Matching

- What is a suitable query signature?

Operator	Table Signature
Table/View (t)	$S_t = [\mathrm{F}; t]$
Select (σ)	$S_{\sigma(e)} = S_e$, *if* $G_e = \mathrm{F}$
Project (π)	$S_{\pi(e)} = S_e$, *if* $G_e = \mathrm{F}$
Join ($\bowtie$)	$S_{e_1 \bowtie e_2} = [\mathrm{F}; T_{e_1} \cup T_{e_2}]$, *if* $G_{e_1} = G_{e_2} = \mathrm{F}$
Group-by (γ)	$S_{\gamma(e)} = [\mathrm{T}; T_e]$, *if* $G_e = \mathrm{F}$

- Examples:

  Sig(select * from R) = [F; R]

  Sig(select a,b,c from R,S where R.b = S.b) = [F; R,S]

  Sig(select a,sum(b) from R group by a) = [T; R]

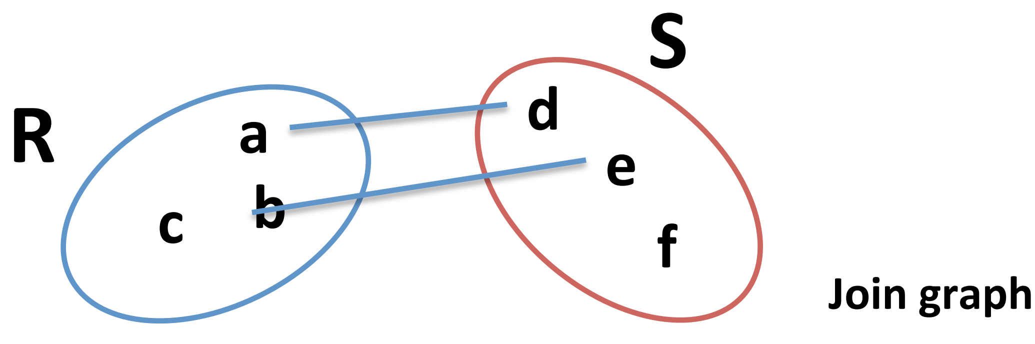
Signature-Based Query Matching

- Two queries are join-compatible iff the equijoin graph constructed from their equivalence classes is connected

Signature-Based Query Matching

- Two queries are join-compatible iff the **<u>equijoin graph</u>** constructed from their intersected equivalence classes is connected
- Example, with schemas R(a,b,c), S(d,e,f):

$$Q1 : R \bowtie_{R.a=S.d \wedge R.b=S.e} S$$



Signature-Based Query Matching

- Two queries are join-compatible iff the equijoin graph constructed from their intersected **equivalence classes** is connected
- Example, with schemas R(a,b,c), S(d,e,f)

$$Q1 : R \bowtie_{R.a=S.d \wedge R.b=S.e} S \Rightarrow \{\{R.a, S.d\}, \{R.b, S.e\}\}$$

$$Q2 : R \bowtie_{R.a=S.d \wedge R.c=S.f} S \Rightarrow \{\{R.a, S.d\}, \{R.c, S.f\}\}$$

Equivalence classes

Signature-Based Query Matching

- Two queries are join-compatible iff the equijoin graph constructed from their intersected **equivalence classes** is connected
- Example, with schemas R(a,b,c), S(d,e,f)

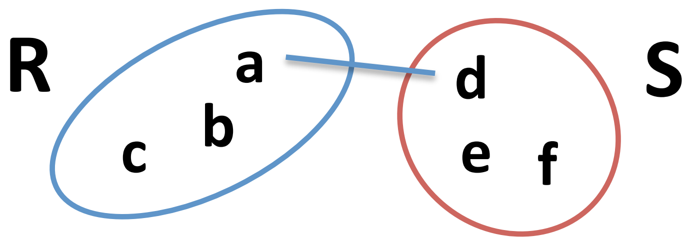
$$Q1 : R \bowtie_{R.a=S.d \wedge R.b=S.e} S \Rightarrow \{\{R.a, S.d\}, \{R.b, S.e\}\}$$

$$Q2 : R \bowtie_{R.a=S.d \wedge R.c=S.f} S \Rightarrow \{\{R.a, S.d\}, \{R.c, S.f\}\}$$

Intersecting equivalence classes:

$$\{\{R.a, S.d\}, \{R.b, S.e\}\} \cap \{\{R.a, S.d\}, \{R.c, S.f\}\} = \{\{R.a, S.d\}\}$$

Looking at the join graph: connected!

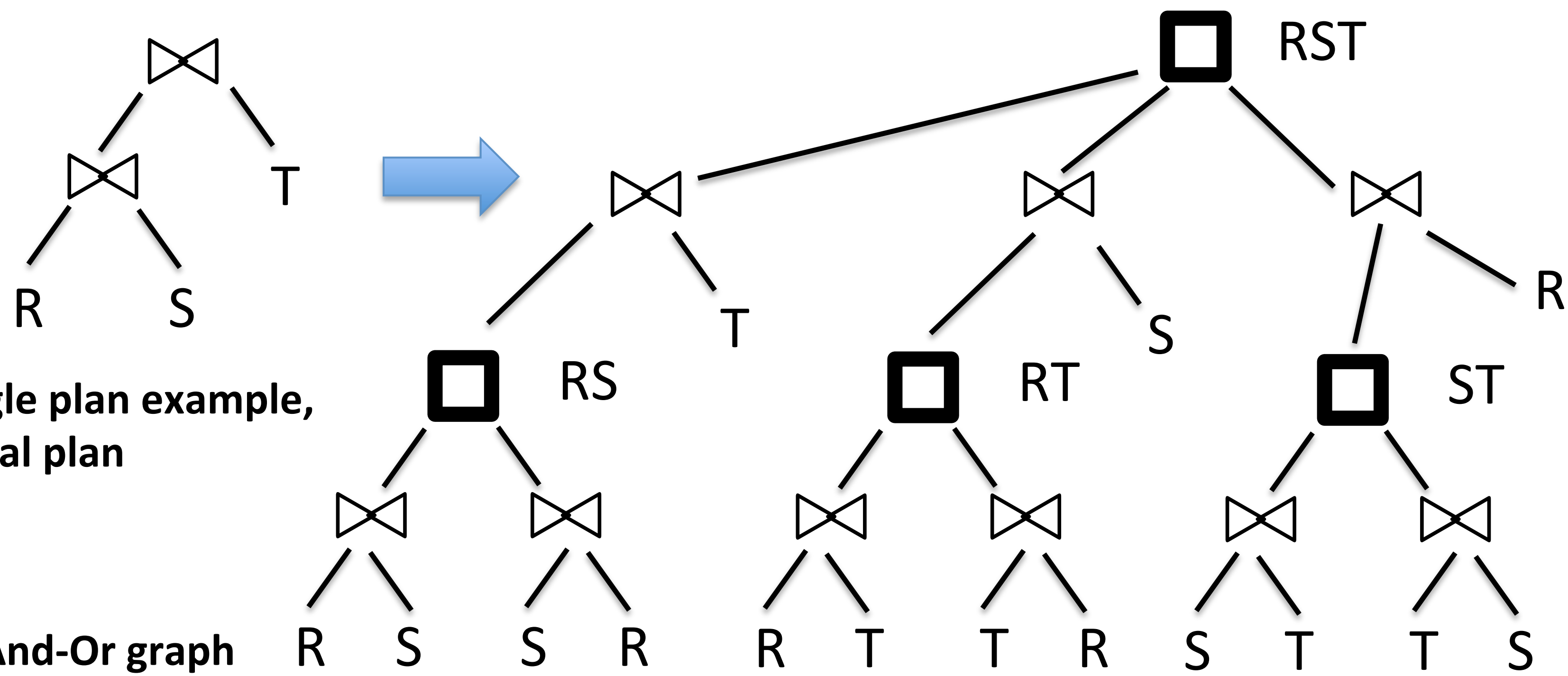


MQO Algorithm Overview

- Given a query workload, ***W*** **=** $\{Q_1 \ldots Q_N\}$
- Find candidate CSEs, *C*, in *W*
- **Build a multiquery plan that exploits CSEs**
  - A single DAG that represents all of ***W***, where the DAG includes ***C***
- Optimize the multiquery plan, extending standard dynamic programming techniques

MQ Plans: And-Or Graphs

- Boxes are “or” nodes
- Each input to an “or” node indicates an “equivalent” plan
- Existing operators (e.g. SPJAG) are “and” nodes

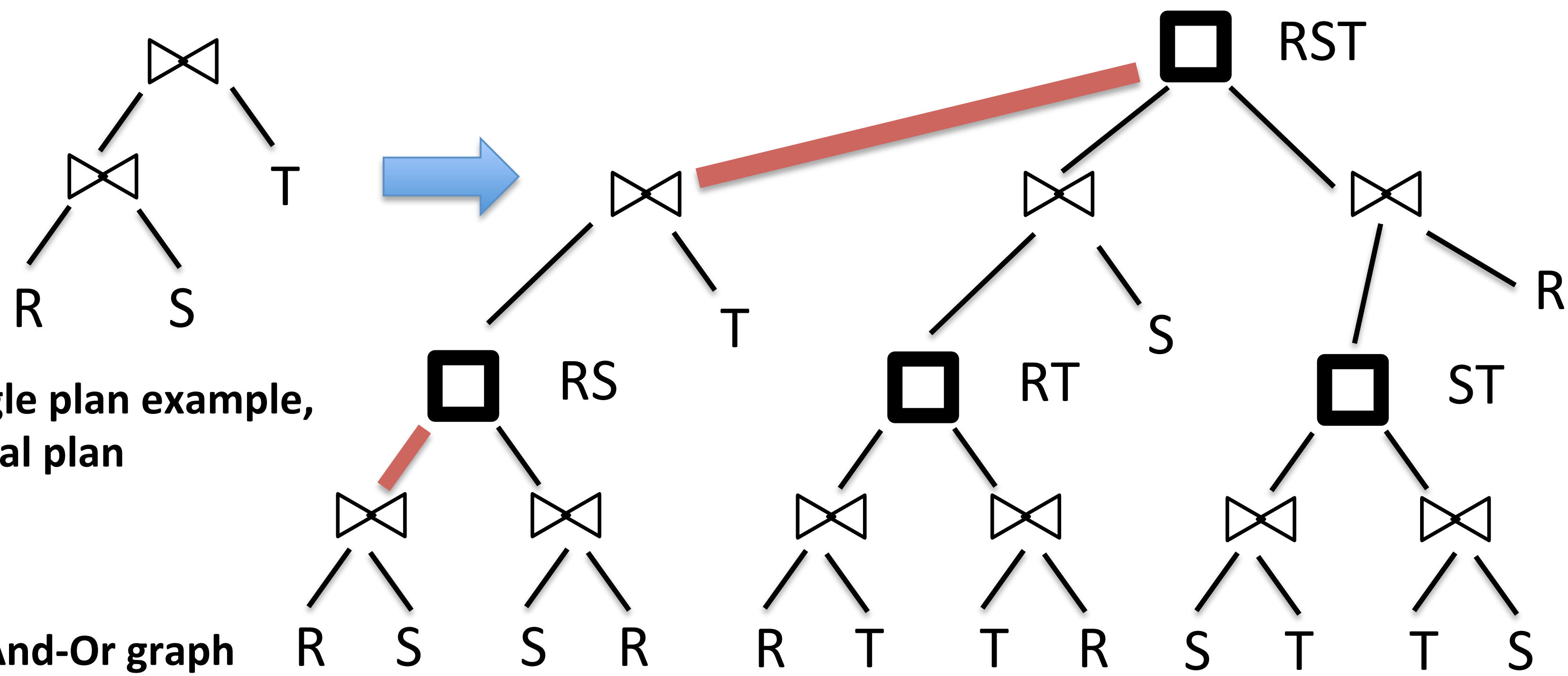


Single plan example, initial plan

And-Or graph

MQ Plans: And-Or Graphs

- Boxes are “or” nodes
- Each input to an “or” node indicates an “equivalent” plan
- Existing operators (e.g. SPJAG) are “and” nodes



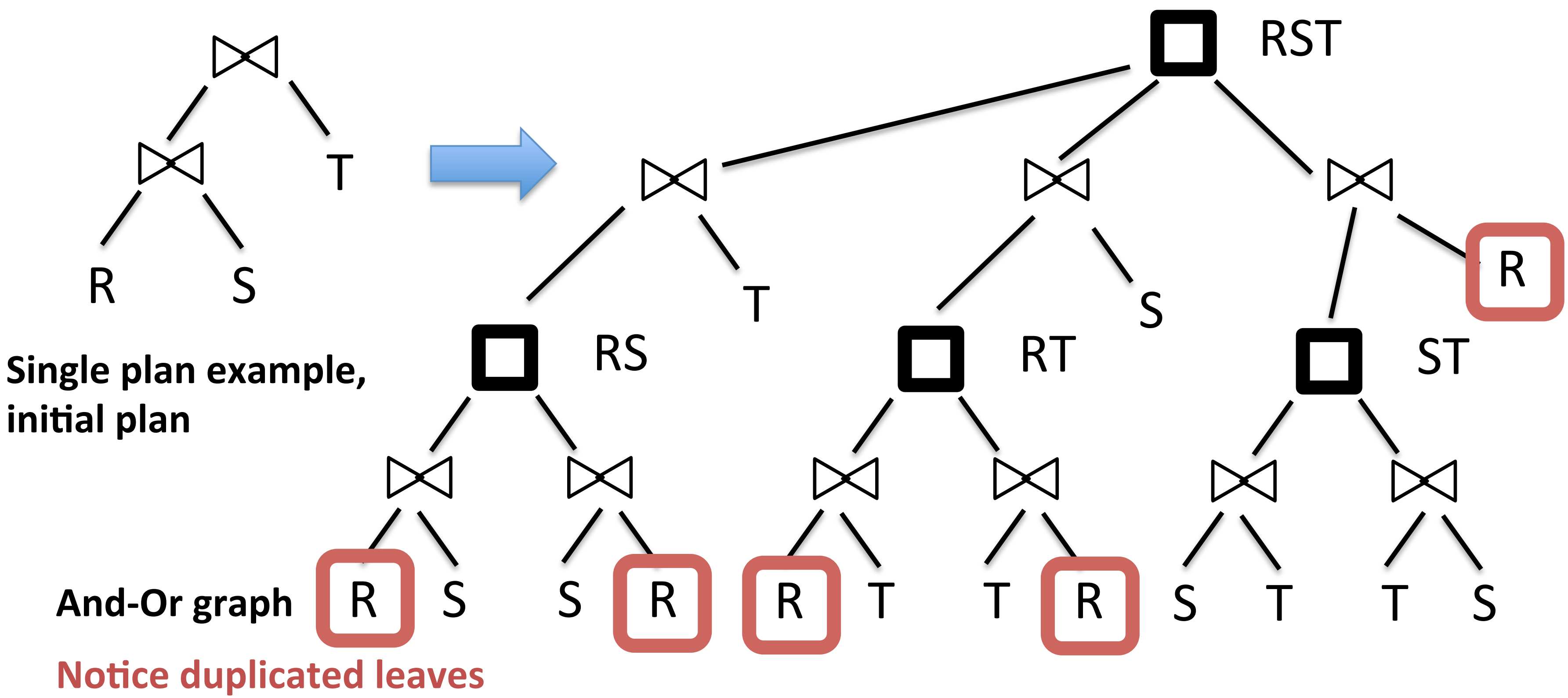
Single plan example, initial plan

And-Or graph

Highlighted branch indicates plan choice

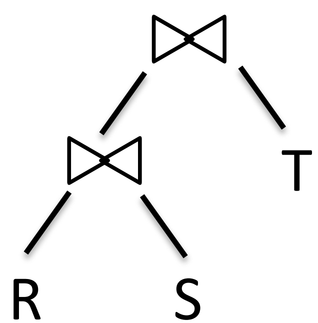
MQ Plans: And-Or Graphs

- Boxes are “or” nodes
- Each input to an “or” node indicates an “equivalent” plan
- Existing operators (e.g. SPJAG) are “and” nodes

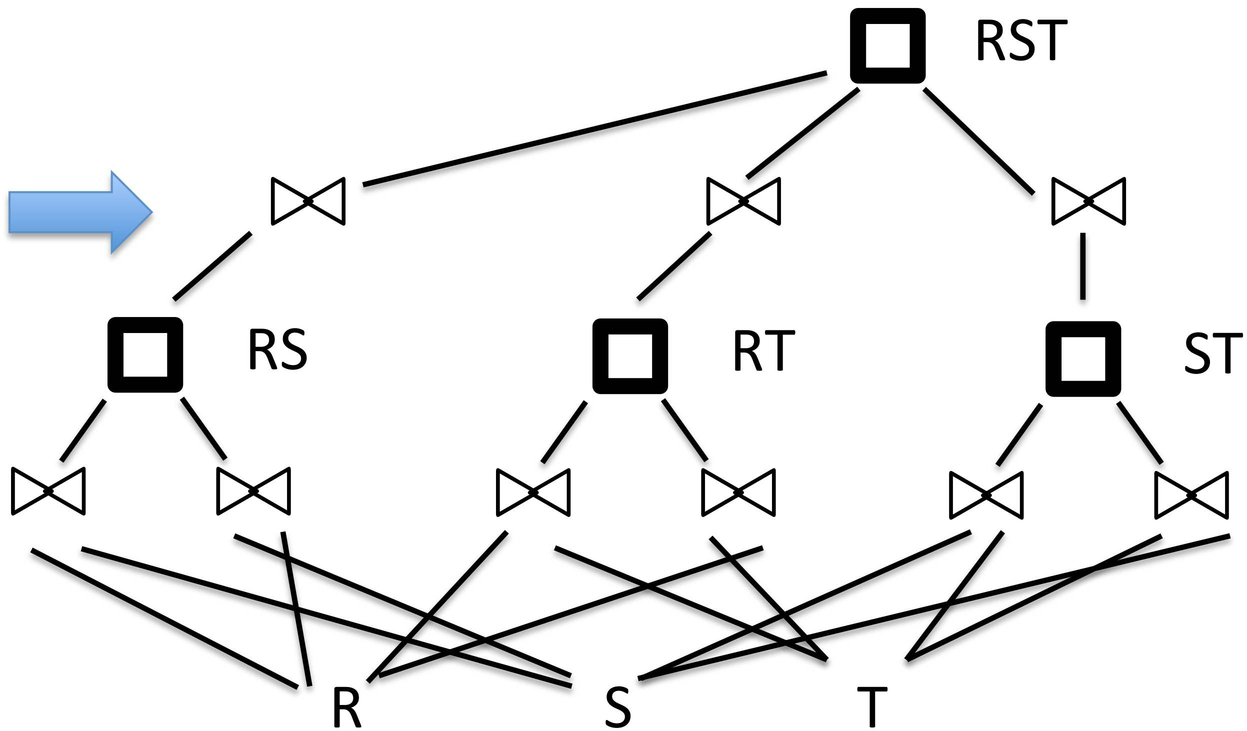


MQ Plans: And-Or Graphs

- Boxes are “or” nodes
- Each input to an “or” node indicates an “equivalent” plan
- Existing operators (e.g. SPJAG) are “and” nodes



Single plan example, initial plan

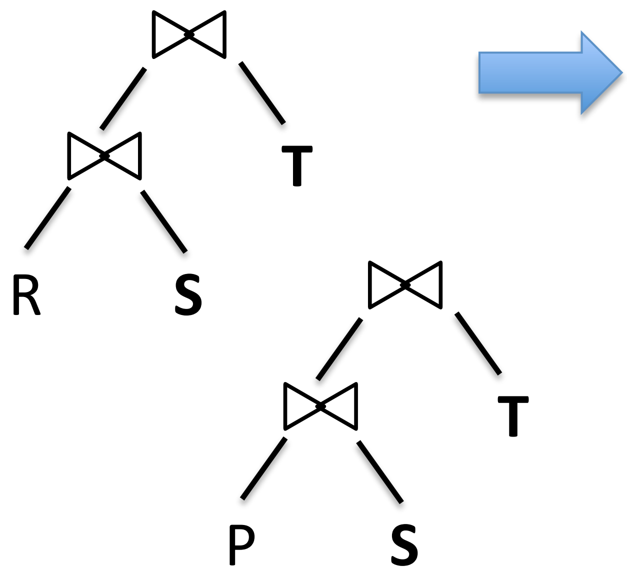


And-Or graph

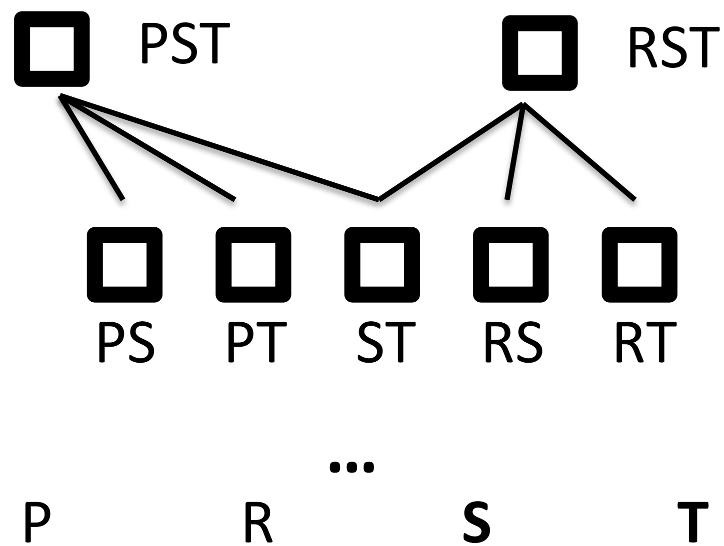
No more duplicate leaves. Notice lattice structure, just as with datacube lattice

MQ Plans: And-Or Graphs

- Boxes are “or” nodes
- Each input to an “or” node indicates an “equivalent” plan
- Existing operators (e.g. SPJAG) are “and” nodes



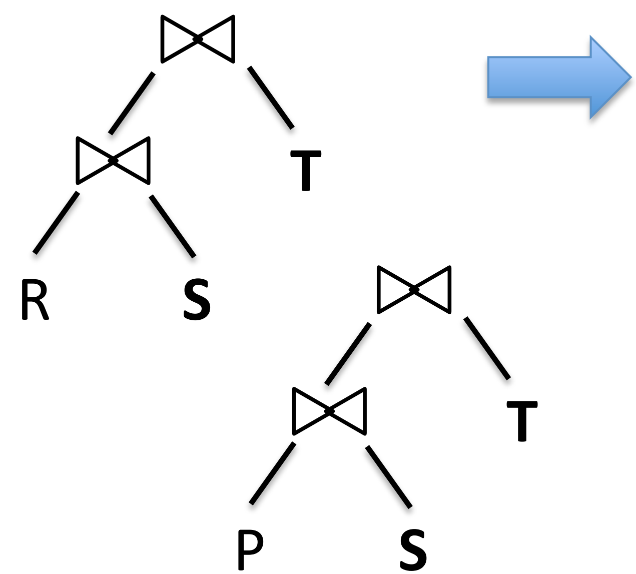
Multiple plan example



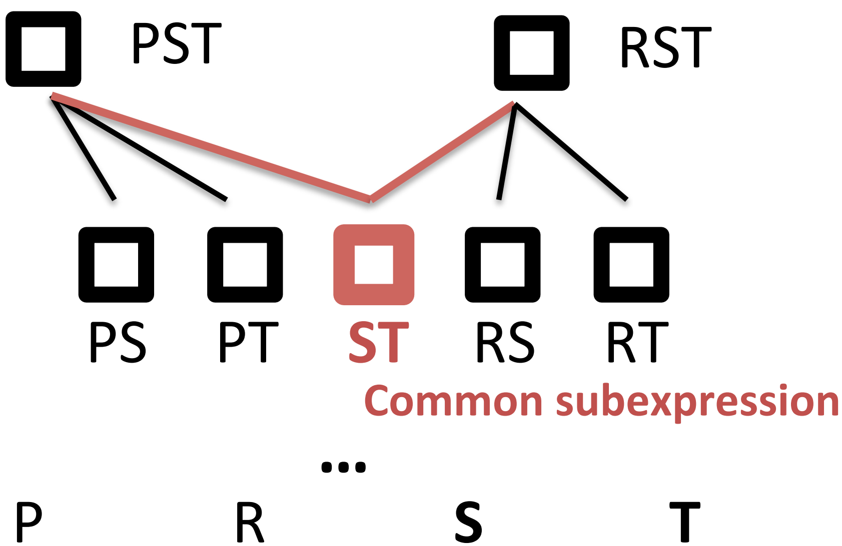
And-Or graph (only “or” nodes shown)

MQ Plans: And-Or Graphs

- Boxes are “or” nodes
- Each input to an “or” node indicates an “equivalent” plan
- Existing operators (e.g. SPJAG) are “and” nodes



Multiple plan example



And-Or graph (only “or” nodes shown)

MQO Algorithm

- Cost models for and-or graphs
- And-nodes

$$cost(o) = exec\_cost(o) + \sum_{e_i \in \text{children(o)}} cost(e_i)$$

- Or-nodes

$$cost(e) = \min\{cost(o_i) | o_i \in \text{children(e)}\}$$

MQO Algorithm Overview

- Given a query workload, ***W*** **= {Q_1…Q_N}**
- Find candidate CSEs, ***C***, in ***W***
- Build a multiquery plan that exploits CSEs
  - A single DAG that represents all of ***W***, where the DAG includes ***C***
- Optimize the multiquery plan, extending standard dynamic programming techniques

MATERIALIZED VIEWS

Database Views

- A view is a table derived as the result of a query, that may optionally be stored or materialized to disk
- A view is created by a defining query, and available for use in queries just like any other relation in the DBMS

```
create view Rmax as
select b, max(a) as ma
from R
group by R.b
```

View definition query

```
select sum(S.d)
from Rmax,S
where Rmax.b = S.b
and S.d < R.ma
```

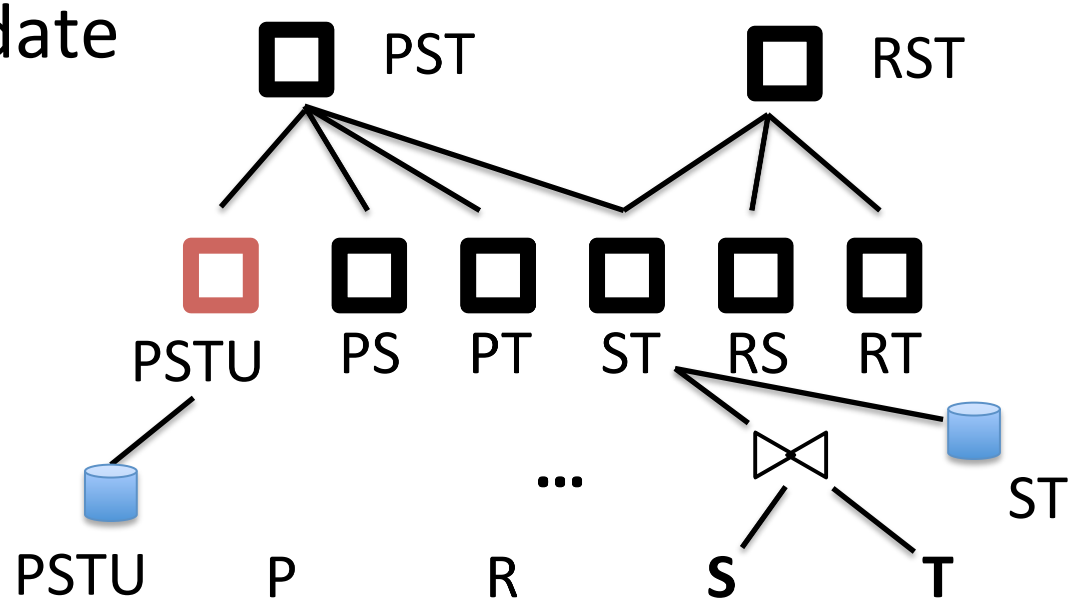
Example query using the defined view

Why Are Views Useful?

- They are the main mechanism for abstraction in a DBMS (both logical and physical)
  - e.g. physical abstractions may implement the same relation with different storage schemes
  - e.g. logical abstractions may implement the same relation with different normalization
- They allow derived relations to be named, referenced, and shared

Using Views for MQO

- We can use views as candidate CSEs in addition to those present in the workload
- We can decide to materialize candidate CSEs as views
- Much of the machinery for detecting sharing is useful for views



And-Or graph (only "or" nodes shown)

View Materialization

- Materialization: storing the results of a view computation to disk
  - The view results can be reused when we see the same query again.
  - How is this different to caching? Views are ***maintained***, caches are simply invalidated
- Partial materialization: we need not store the entire view on disk, but only a subset of its rows
  - How do we pick which rows to keep?
  - There are many algorithms, that typically depend on row "heat", i.e. how useful a row is to a query workload

View Matching

- Question: given a query, how can we determine if we can use a view to answer it?
- We may be able to use a view
  - if the view completely answers the query, i.e. the query is contained in the view (i.e. the view subsumes the query)
  - if the view partially answers the query (i.e. if there is some commonality between the query and the view)
- We can use similar techniques (i.e. signatures) as with MQO

View Maintenance

- The maintenance problem: if my base relations are updated, how do I refresh my view so that query answering remains up-to-date?
- Two high-level approaches:
  - Full refresh: recompute the query from scratch on every update
    - A general-purpose technique that works for all kinds of queries
    - But, it is inefficient since many rows in the view may be unaffected by the update
    - No need to do this on every update (i.e. ***eager***), instead we can be ***lazy*** and do this periodically (queries may have different freshness requirements)
  - Incremental refresh: recompute only those parts of the view that are affected by the updates
    - Relies on the concept of ***delta*** queries

Incremental View Maintenance Algorithms

- Delta queries are computed by a program transformation, which symbolically replaces a relation in the query with a single tuple
- Example:

```
q= select l.ordkey, o.sprior,  delta =>
          sum(l.extprice)
   from  Lineitem l, Orders o
   where l.ordkey = o.ordkey
   group by l.ordkey, o.sprior;
```

```
select l.ordkey, o.sprior,
       sum(l.extprice)
from values(@ok,@ep)
     as l(ordkey,extprice),
     Orders o
where l.ordkey = o.ordkey
group by l.ordkey, o.sprior
```

Incremental View Maintenance Algorithms

- Delta queries are computed by a program transformation, which symbolically replaces a relation in the query with a single tuple
- Example:

```
q= select l.ordkey, o.sprior,
          sum(l.extprice)
   from  Lineitem l, Orders o
   where l.ordkey = o.ordkey
   group by l.ordkey, o.sprior;
```

delta =>

```
select l.ordkey, o.sprior,
       sum(l.extprice)
from values(@ok,@ep)
     as l(ordkey,extprice),
     Orders o
where l.ordkey = o.ordkey
group by l.ordkey, o.sprior
```

simplify =>

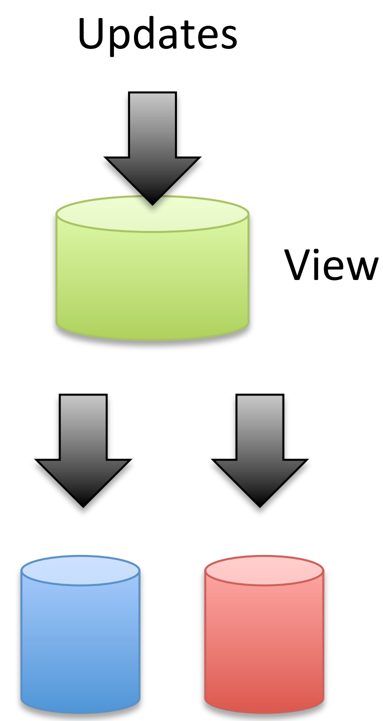
```
dq =
select @ok, sprior, @ep*v
from
(select o.sprior, sum(1) as v
 from  Orders o
 where @ok = o.ordkey
 group by o.sprior)
```

Incremental update:

$q_{new} = q_{old}$ union dq

View Update

- So far we have treated views as read-only derived data
  - This makes sense for many classes of OLAP queries; statistics are "read-only"
- The view update problem: how do I support writeable views, so that updates to my view are propagated back to the base relation?
  - This is generally difficult, it requires an inverse or bidirectional query
  - e.g., how do you invert a join or aggregate?



Base tables, inconsistent with the view after updates to the view

Next Lecture: Physical DB Design

- How do we automatically pick good views to maintain for a query workload ***W***
- Also, how do we pick good indexes?
- How do these data structure selection mechanisms interact with query optimization?