Checkpont 3: Joins

April 20, 2017

Recap: Joins

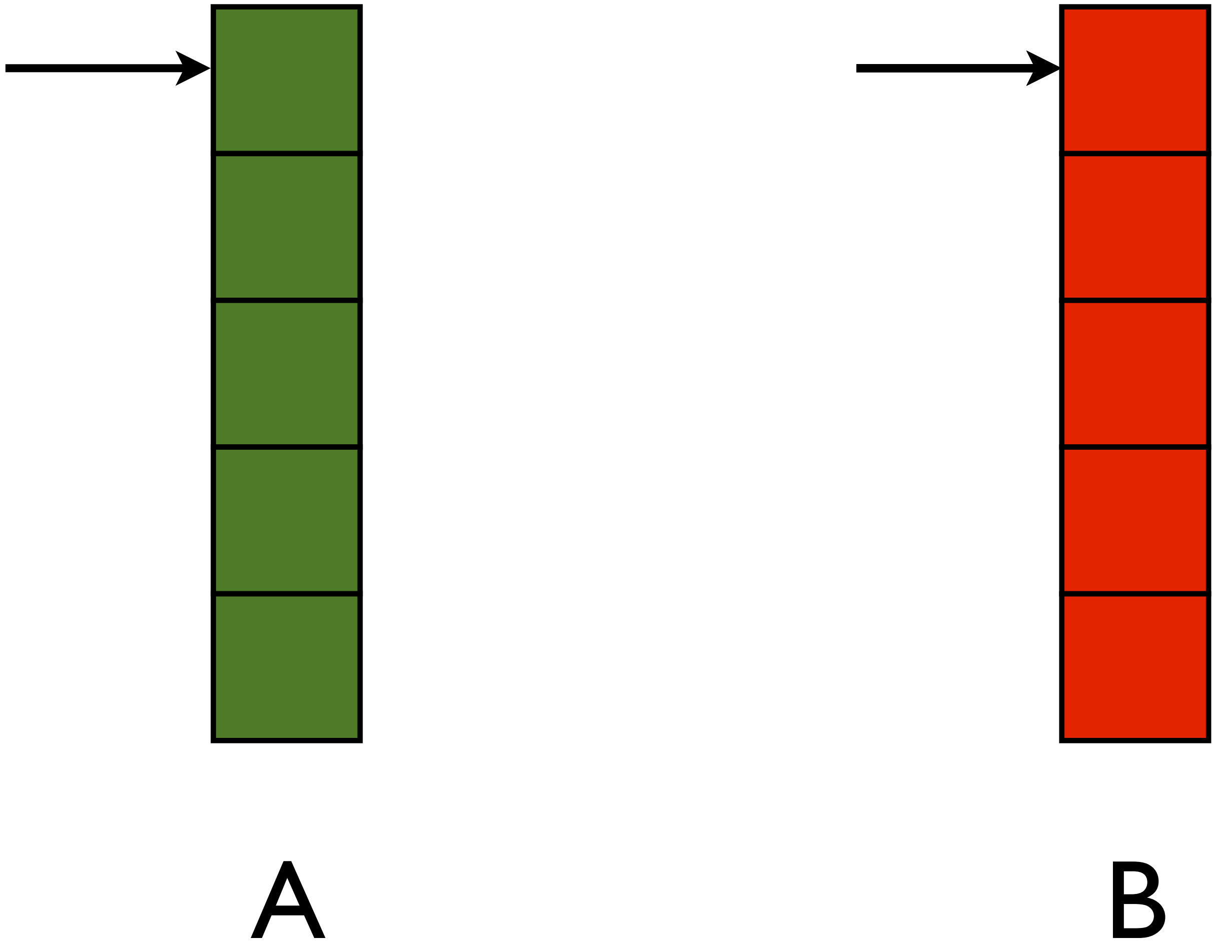
- Two General Classes of Joins
  - Equality (Equi-) Joins: `R.B = S.B`
  - Inequality (Inequi-) Joins: `R.B < S.B`

Inequi-joins are O(N^2) (as bad as NLJ)
Checkpoint 3 focuses on Equi-joins

Implementing: Joins

Solution 0 (Nested-Loop)

```
For Each (a in A) { For Each (b in B) { emit (a, b); }}
```



Implementing: Joins

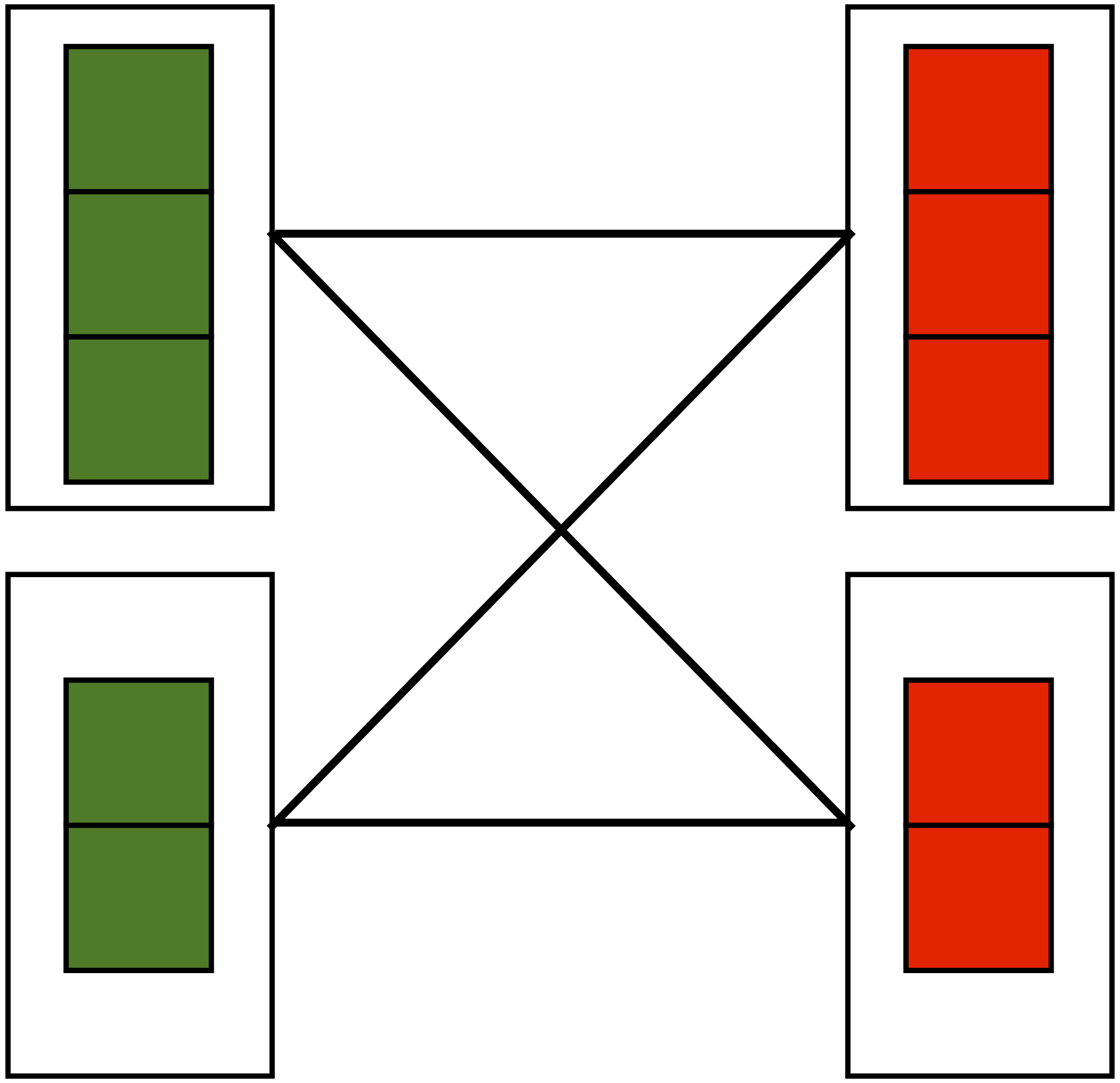
Solution 1 (Block-Nested-Loop)

Implementing: Joins

Solution 1 (Block-Nested-Loop)

1) Partition into Blocks

2) NLJ on each pair of blocks

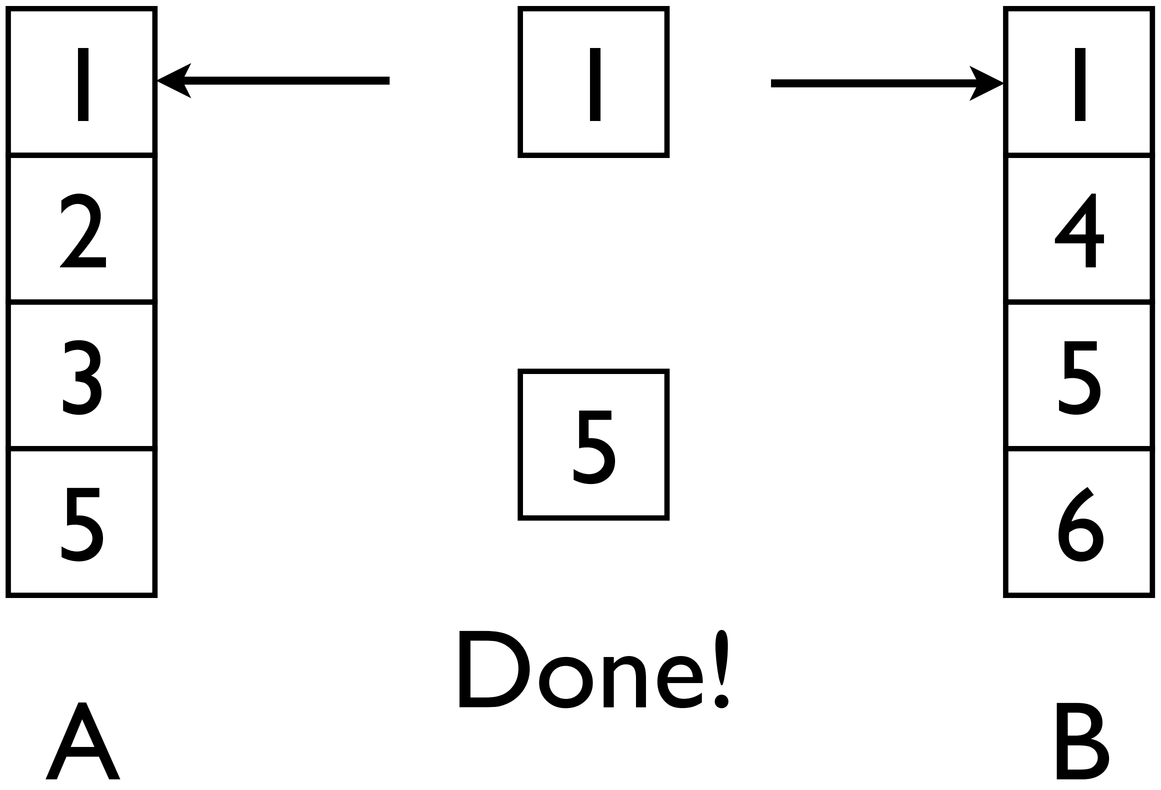


Implementing: Joins

Solution 2 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

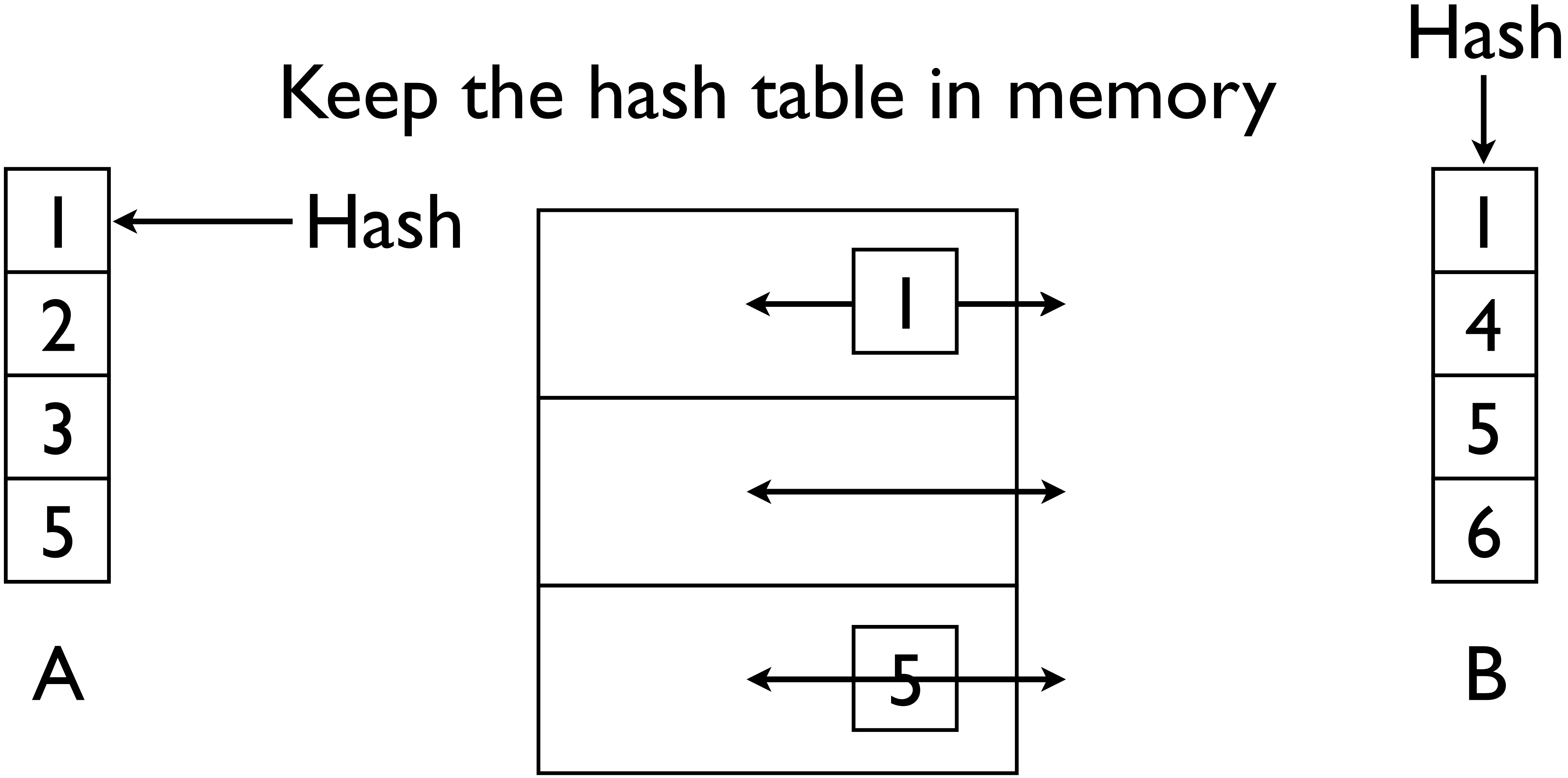
When you hit two that match, emit, then iterate both



Implementing: Joins

Solution 3 (1-Pass Hash)

Keep the hash table in memory

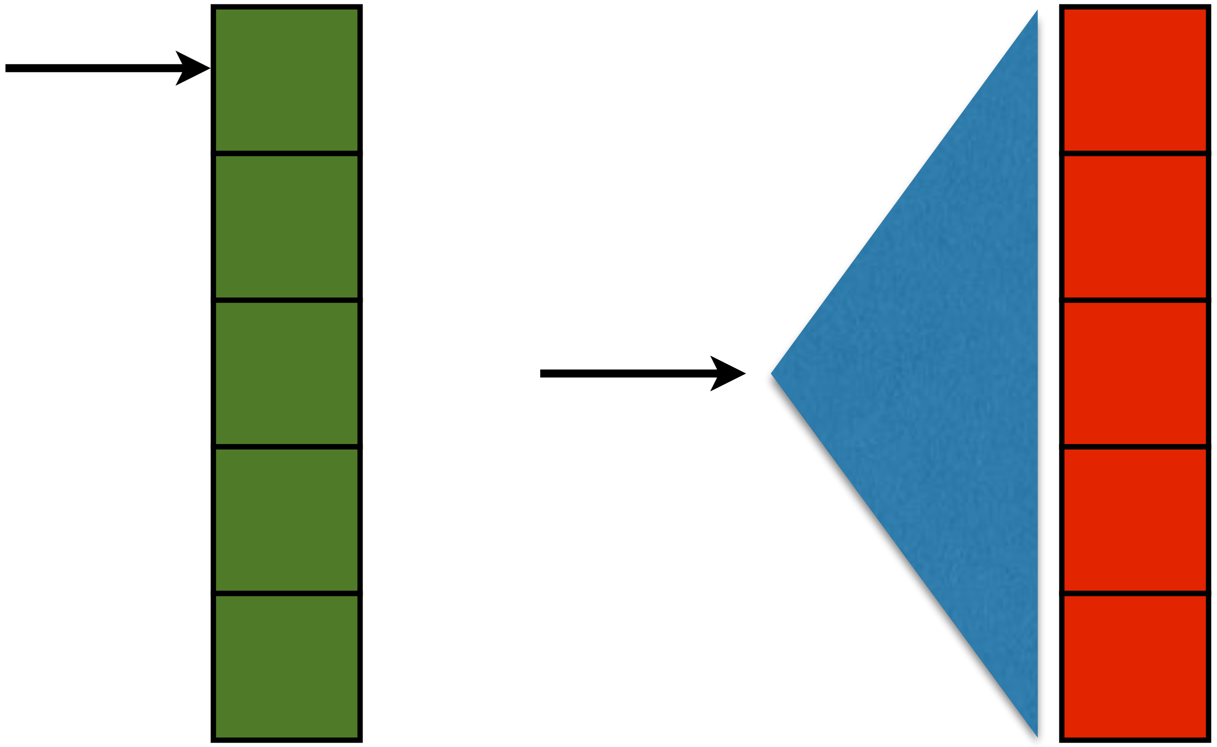


(Essentially a more efficient nested loop join)

Implementing: Joins

Solution 4 (Index-Nested-Loop)

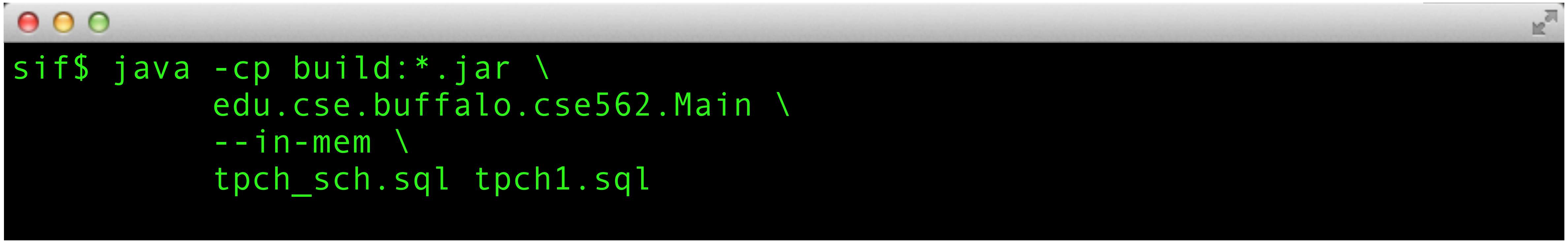
Like nested-loop, but use an index to make the inner loop much faster!



What are the tradeoffs of each algorithm?

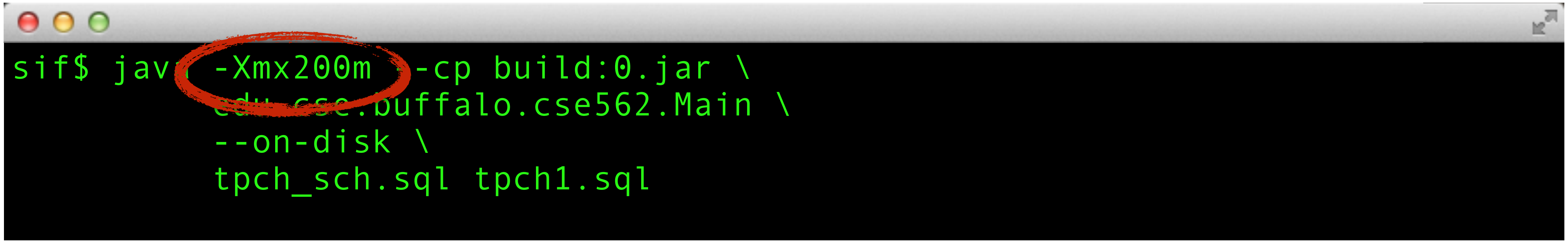
What properties
do we care about?

How do the
algorithms compare?



```
sif$ java -cp build:*.jar \
          edu.cse.buffalo.cse562.Main \
          --in-mem \
          tpch_sch.sql tpch1.sql
```

Phase 1: Identical… just needs support for joins.

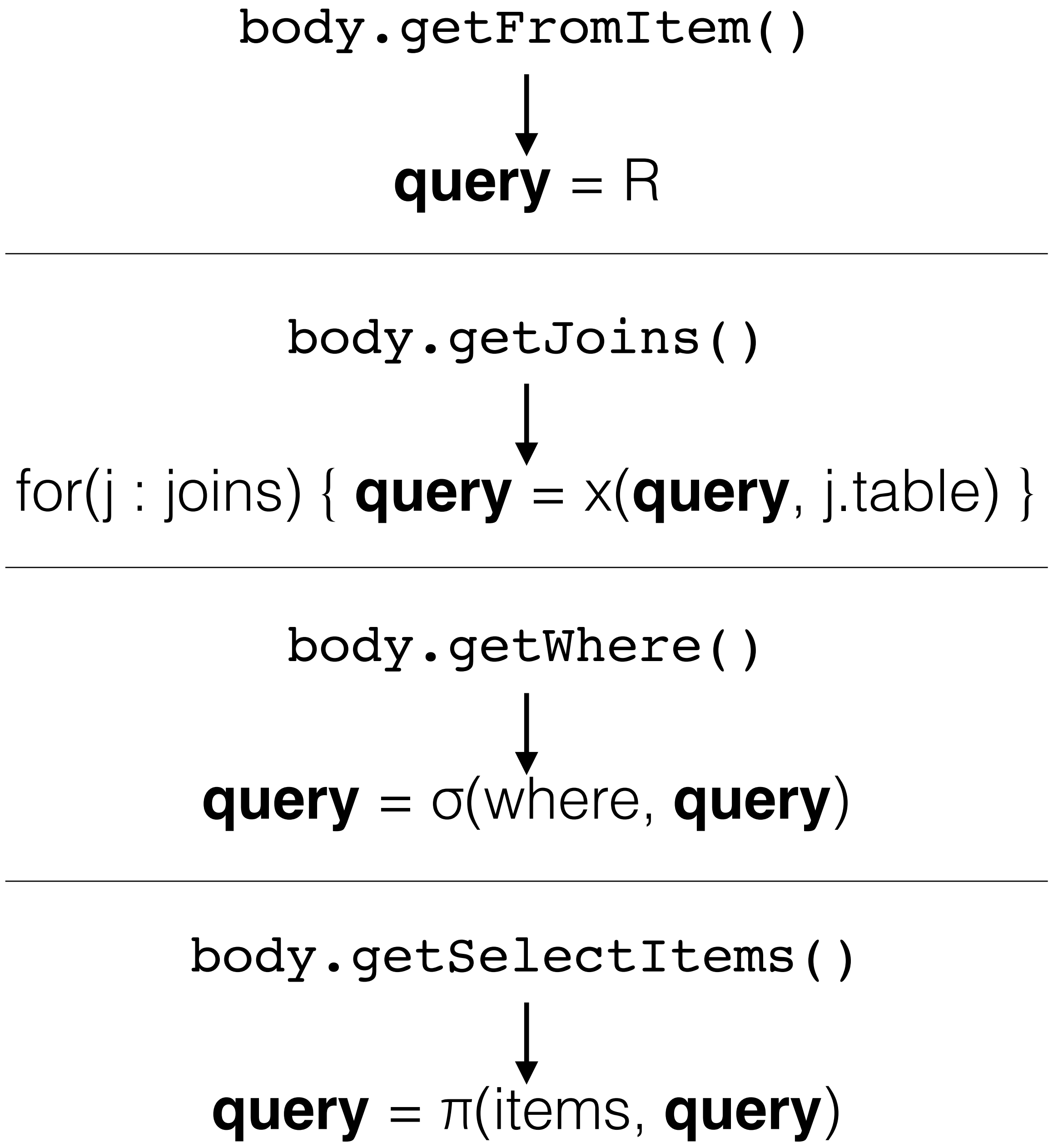


```
sif$ java -Xmx200m -cp build:0.jar \
          edu.cse.buffalo.cse562.Main \
          --on-disk \
          tpch_sch.sql tpch1.sql
```

Phase 2: Identical… just needs support for joins.

```sql
CREATE TABLE R ( A int, B int );
CREATE TABLE S ( B int, C int );

SELECT R.A, S.C FROM R, S WHERE R.B = S.B;
```



`body.getFromItem()`

query = R

`body.getJoins()`

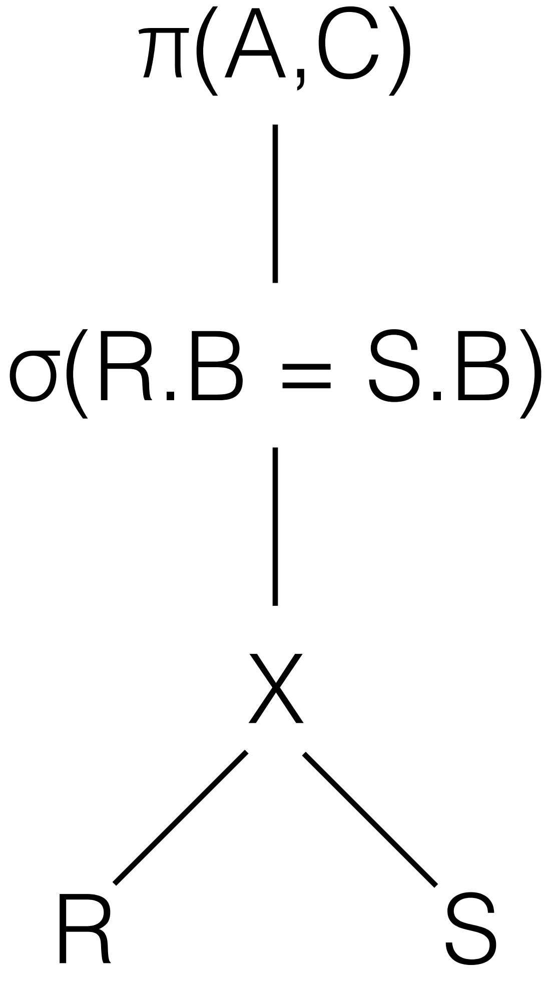
for(j : joins) { **query** = x(**query**, j.table) }

`body.getWhere()`

query = σ(where, **query**)

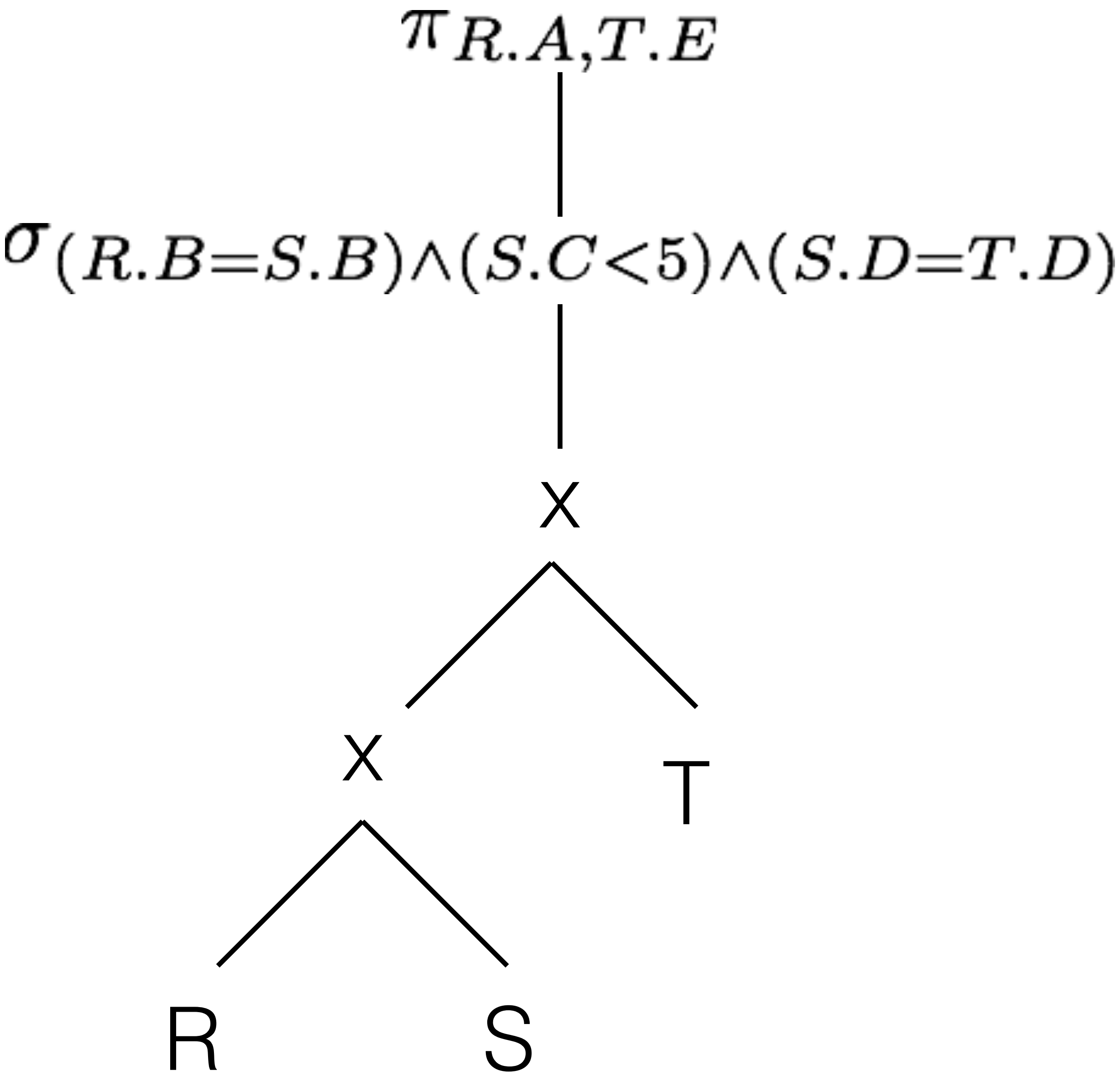
`body.getSelectItems()`

query = π(items, **query**)

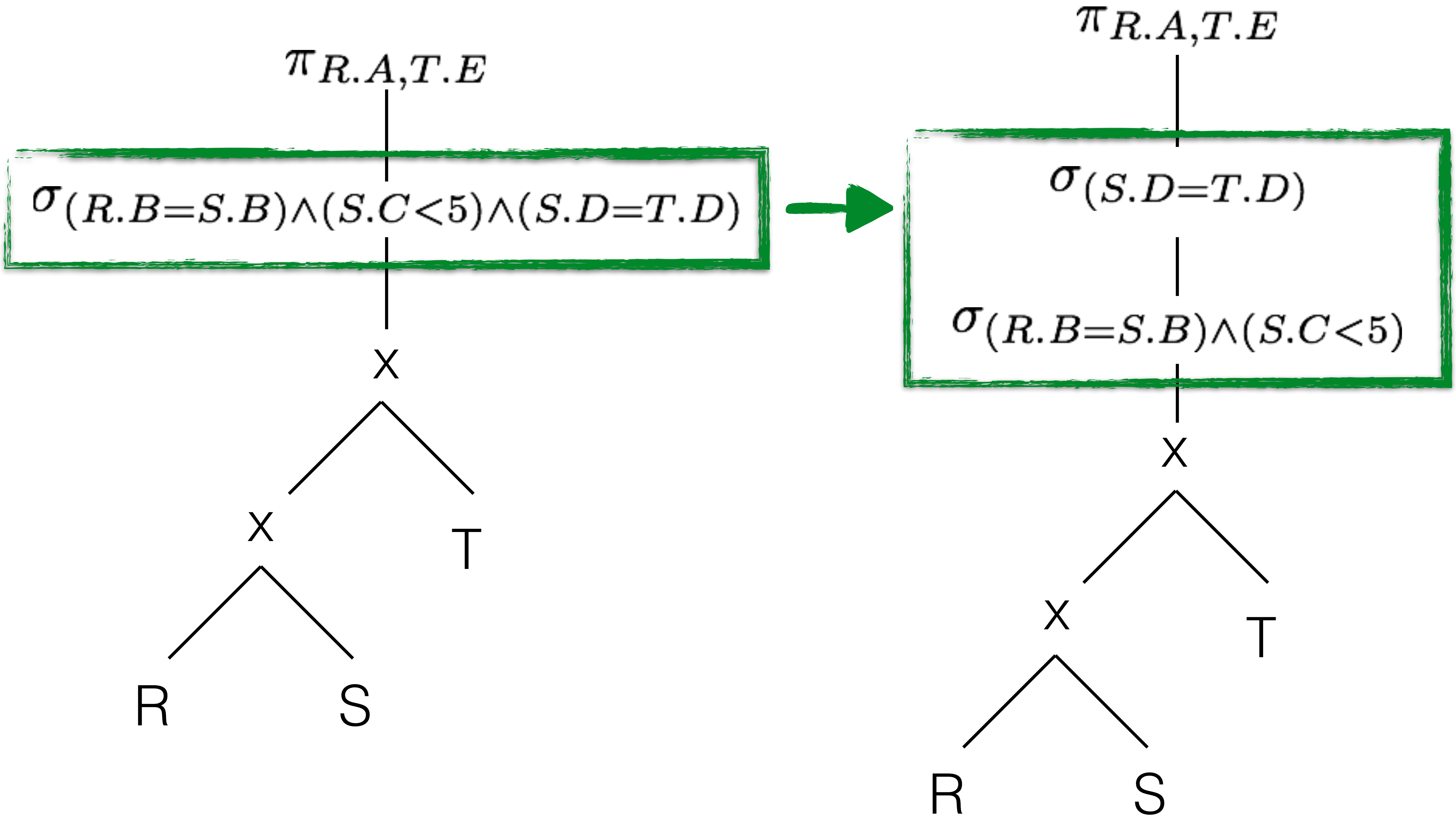


Recap: Optimization

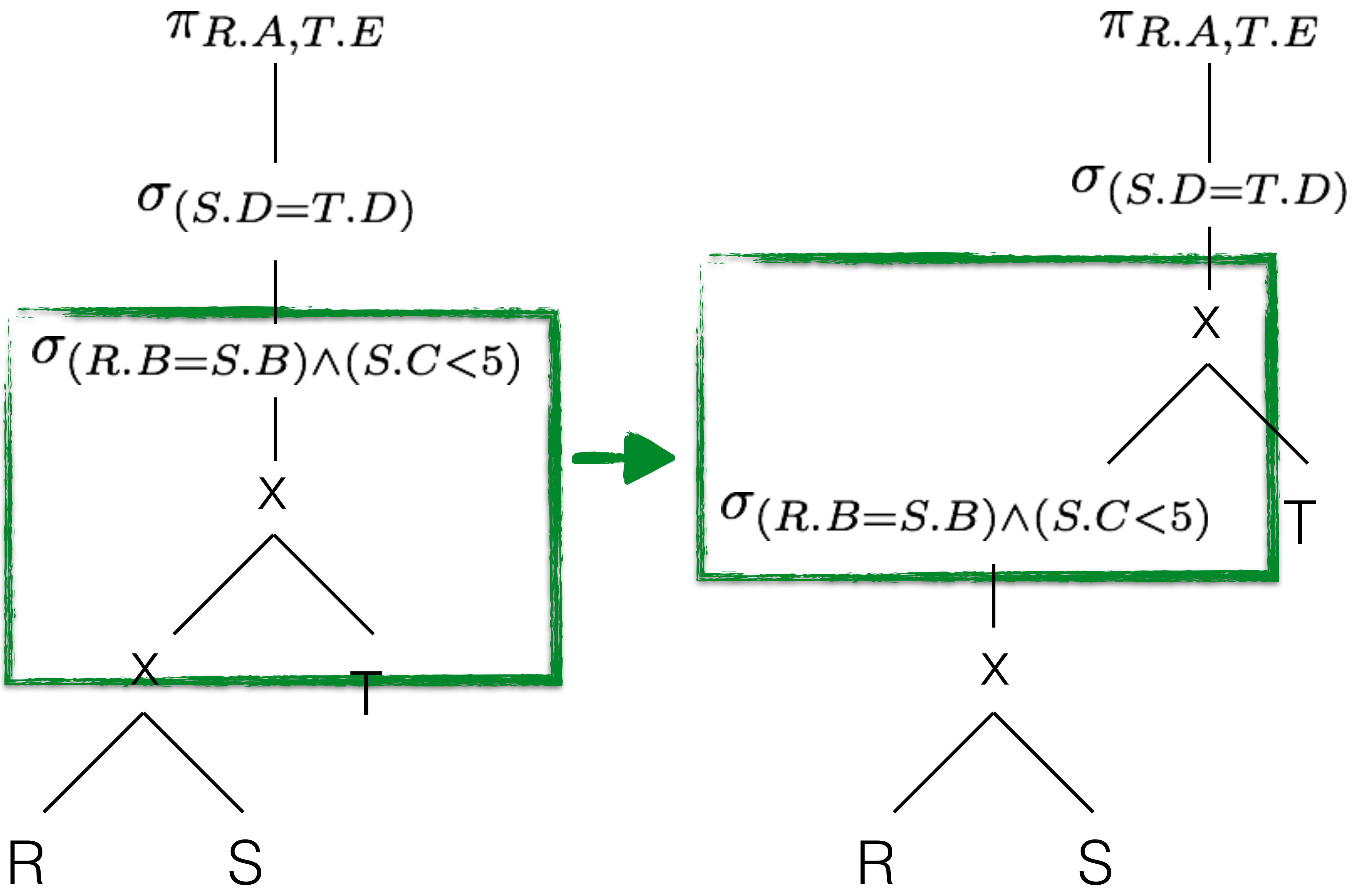
```
SELECT R.A, T.E
  FROM R, S, T
 WHERE R.B = S.B
   AND S.C < 5
   AND S.D = T.D
```



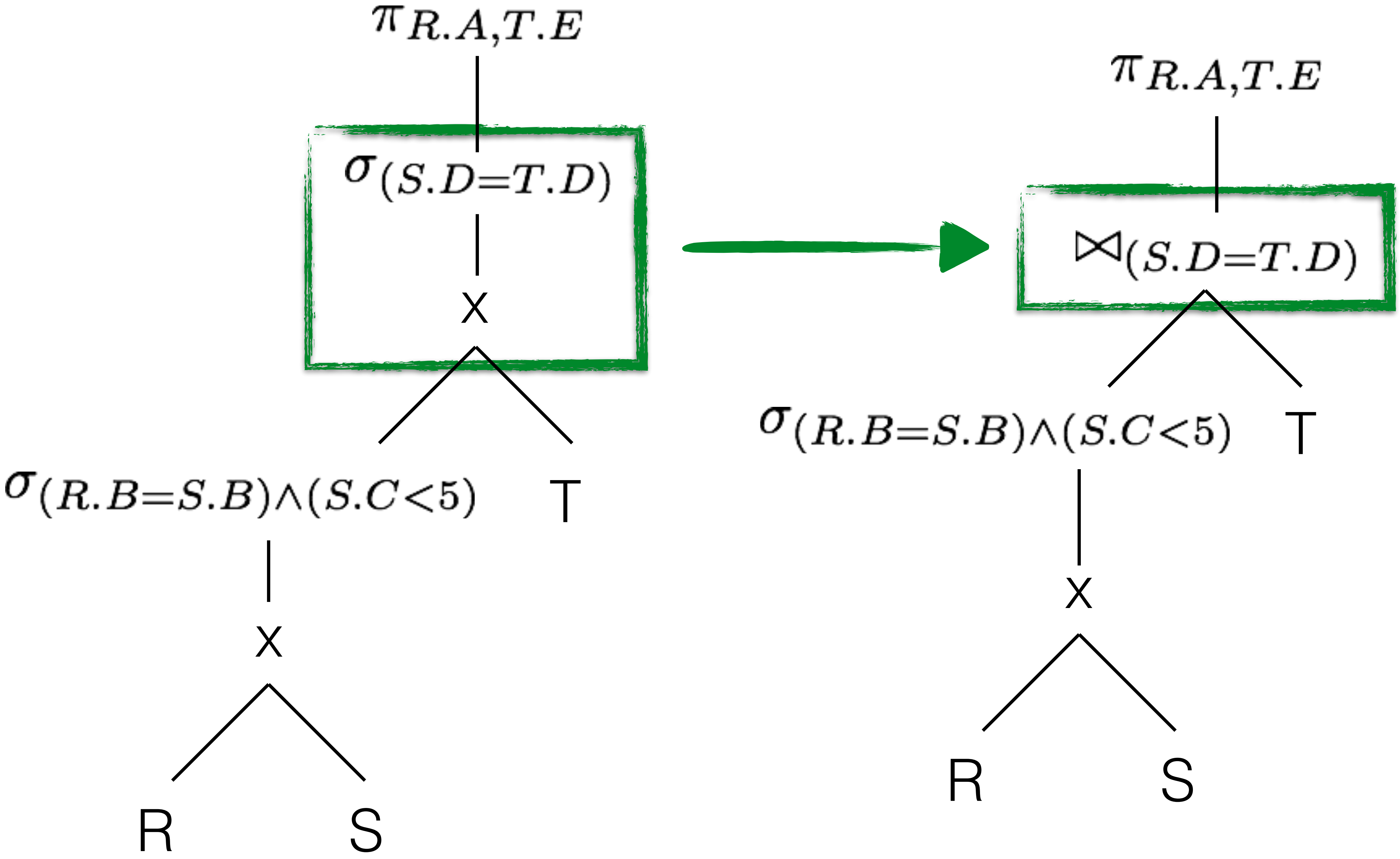
Example


$\pi_{R.A,T.E}$
$\sigma_{(R.B=S.B)\wedge(S.C<5)\wedge(S.D=T.D)}$
X
X
T
R
S
$\pi_{R.A,T.E}$
$\sigma_{(S.D=T.D)}$
$\sigma_{(R.B=S.B)\wedge(S.C<5)}$
X
X
T
R
S

Example


$\pi_{R.A,T.E}$
$\sigma_{(S.D=T.D)}$
$\sigma_{(R.B=S.B)\wedge(S.C<5)}$
X
X
T
R
S
$\pi_{R.A,T.E}$
$\sigma_{(S.D=T.D)}$
X
T
$\sigma_{(R.B=S.B)\wedge(S.C<5)}$
X
R
S

Example



$\pi_{R.A,T.E}$

$\sigma_{(S.D=T.D)}$

X

$\sigma_{(R.B=S.B)\wedge(S.C<5)}$

T

X

R

S

$\pi_{R.A,T.E}$

$\bowtie_{(S.D=T.D)}$

$\sigma_{(R.B=S.B)\wedge(S.C<5)}$

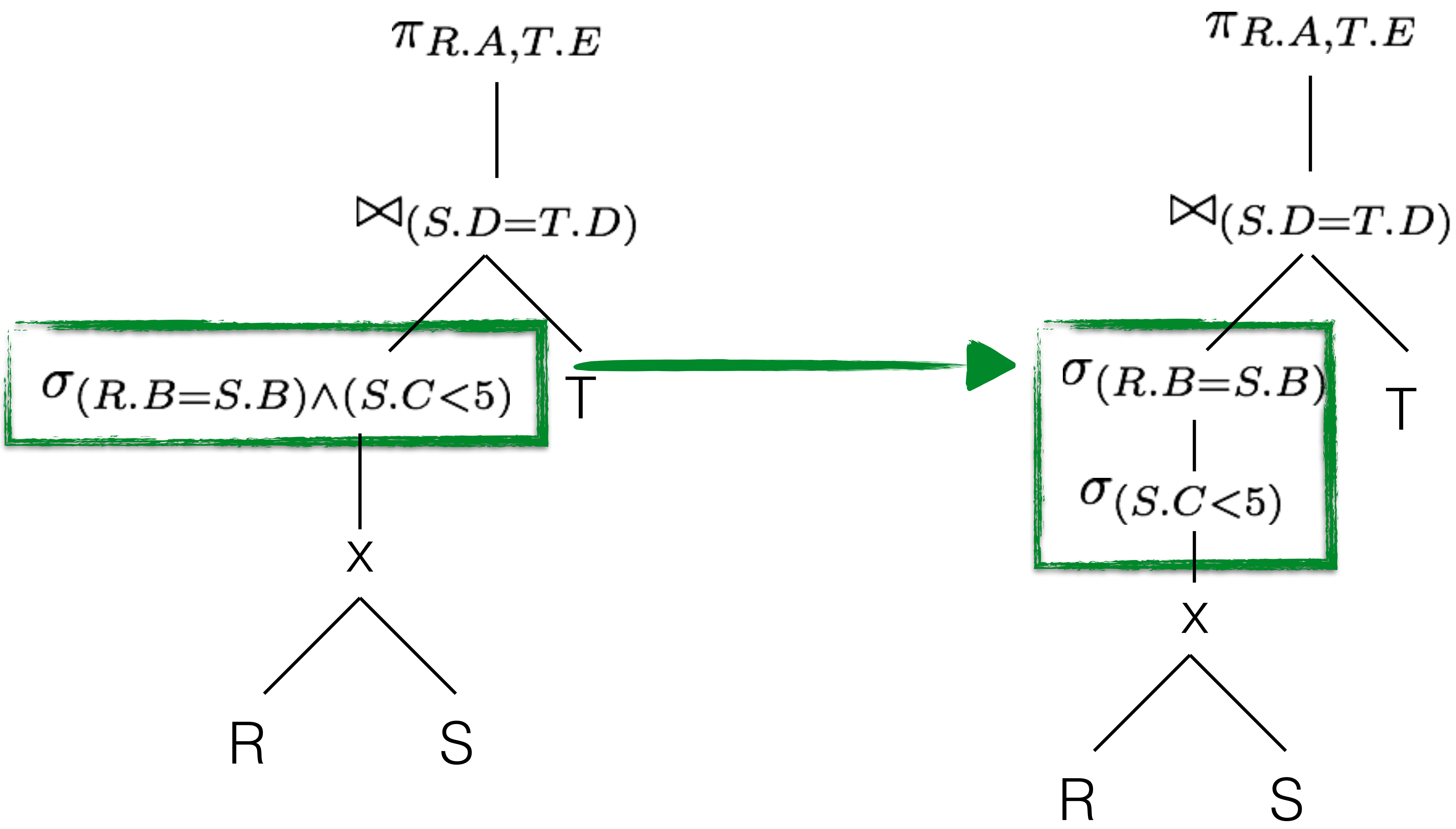
T

X

R

S

Example



$\pi_{R.A,T.E}$

$\bowtie_{(S.D=T.D)}$

$\sigma_{(R.B=S.B)\wedge(S.C<5)}$

T

X

R

S

$\pi_{R.A,T.E}$

$\bowtie_{(S.D=T.D)}$

$\sigma_{(R.B=S.B)}$

T

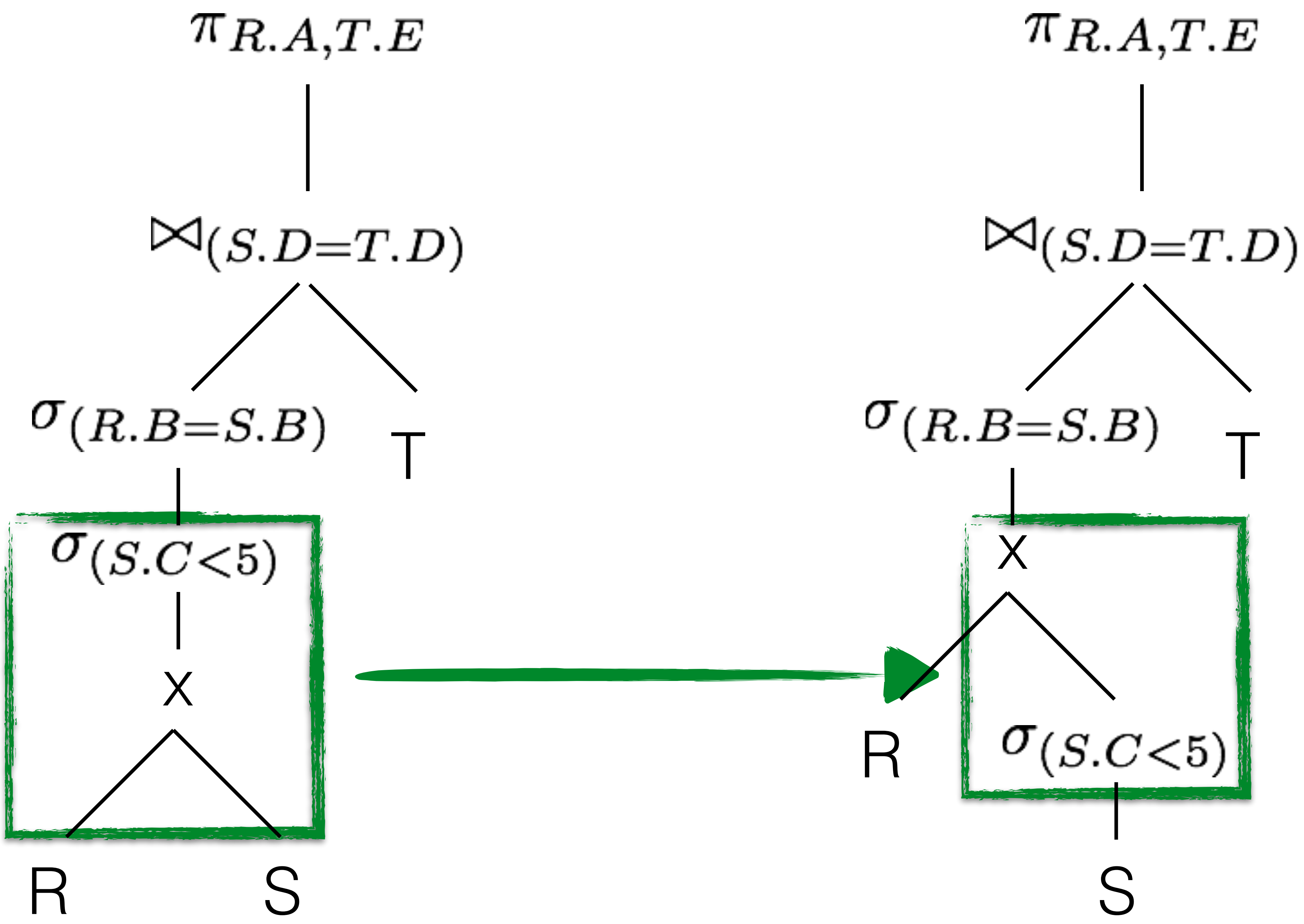
$\sigma_{(S.C<5)}$

X

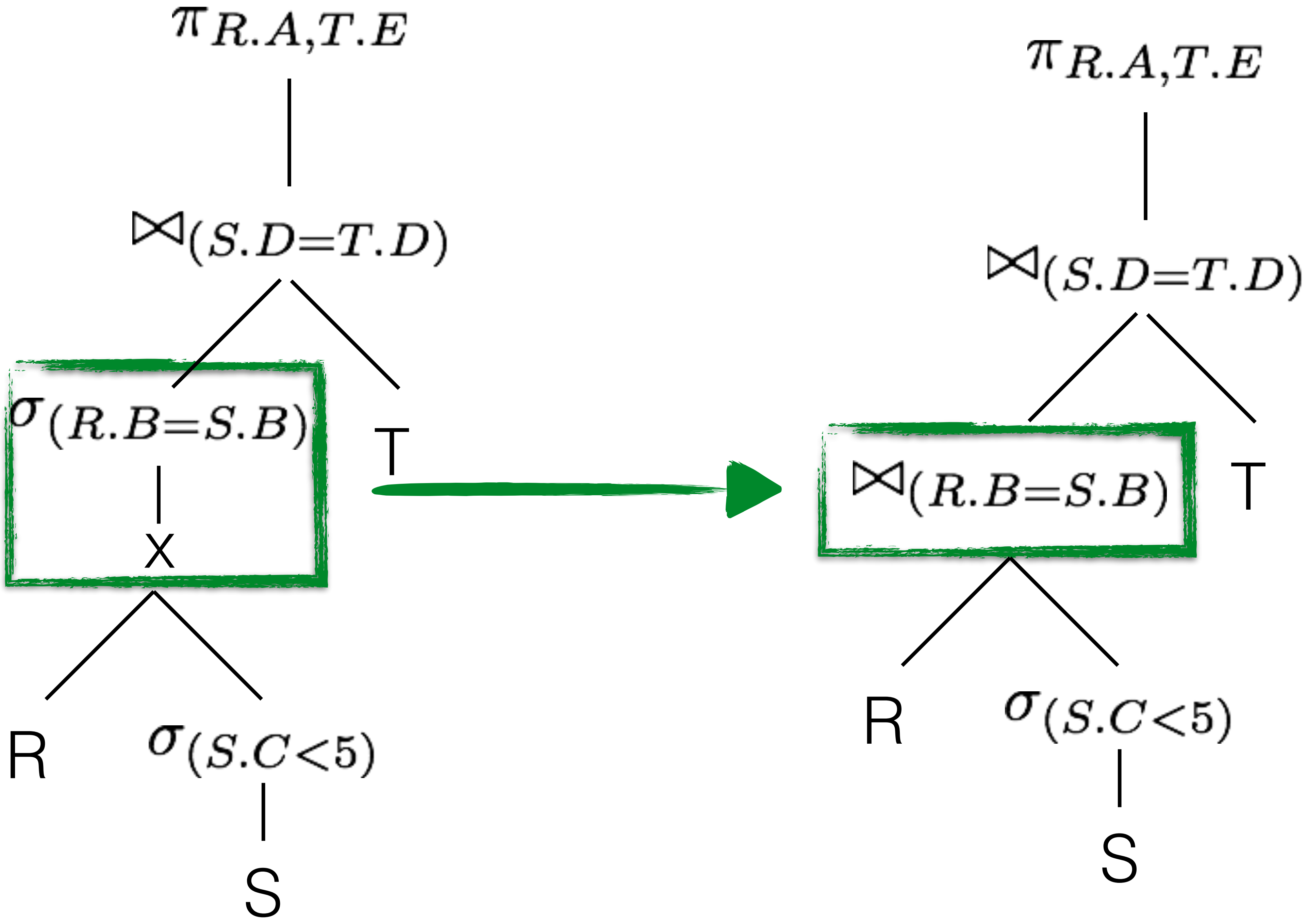
R

S

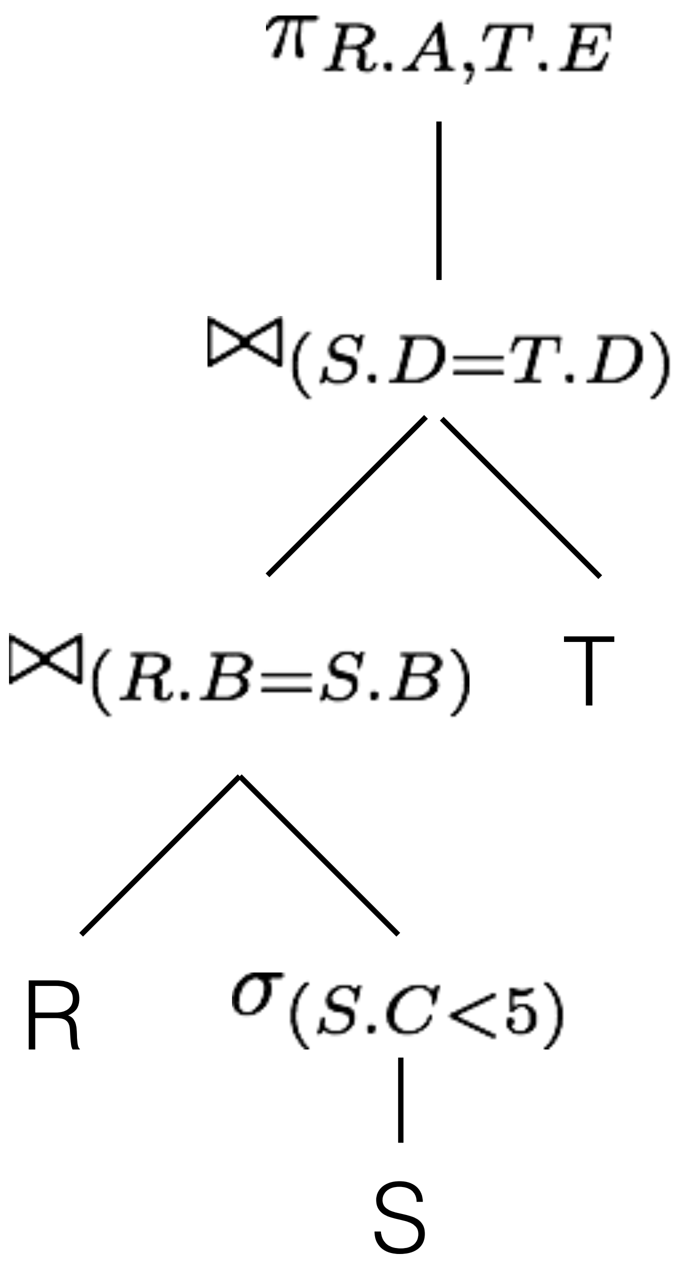
Example


$\pi_{R.A,T.E}$
$\bowtie_{(S.D=T.D)}$
$\sigma_{(R.B=S.B)}$
T
$\sigma_{(S.C<5)}$
X
R
S
$\pi_{R.A,T.E}$
$\bowtie_{(S.D=T.D)}$
$\sigma_{(R.B=S.B)}$
T
X
R
$\sigma_{(S.C<5)}$
S

Example


$\pi_{R.A,T.E}$
$\bowtie_{(S.D=T.D)}$
$\sigma_{(R.B=S.B)}$
T
X
R
$\sigma_{(S.C<5)}$
S
$\pi_{R.A,T.E}$
$\bowtie_{(S.D=T.D)}$
$\bowtie_{(R.B=S.B)}$
T
R
$\sigma_{(S.C<5)}$
S

Final Plan



```
SELECT R.A, T.E
  FROM R, S, T
 WHERE R.B = S.B
   AND S.C < 5
   AND S.D = T.D
```

Optimization

- Find a pattern in the RA-Tree that you can optimize.
- Apply the optimization.
- Repeat as necessary. (more discussion later)

Simple Optimizations (with a big impact)

- Pushdown Selections
- Build Joins
- [Replace Unbounded Memory Operators]

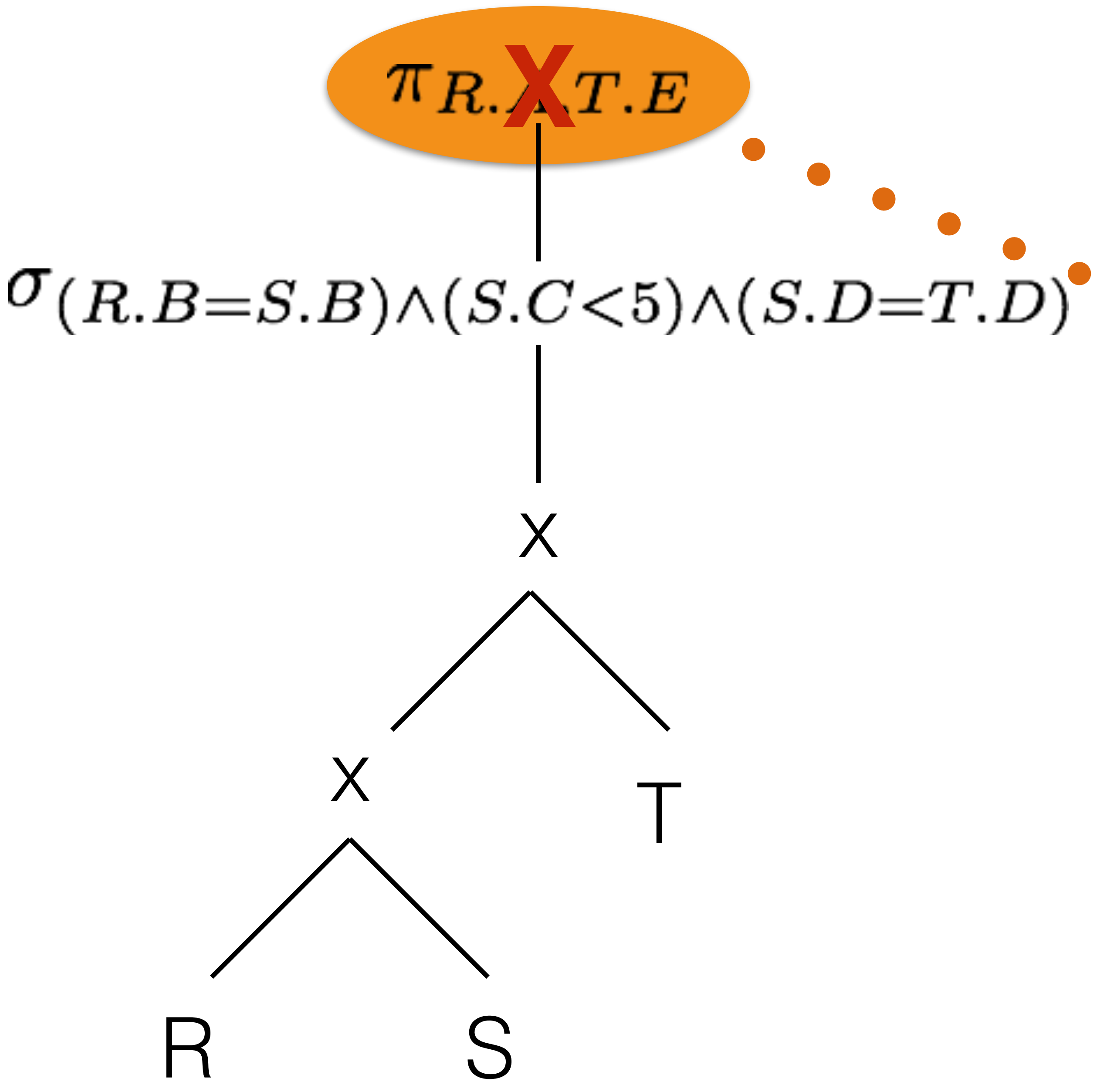
Pushdown Selections

$$\sigma_{C_R \wedge C_S \wedge C}(R \times S) \equiv \sigma_C(\sigma_{C_R}(R) \times \sigma_{C_S}(S))$$

Pattern Match/Replace

```
Operator rewrite(Operator o){
  if(o instanceof Selection) {
    Selection s = (Selection)o;
    if(s.child() instanceof CrossProduct) {
      // Magic happens here
      return new …;
    }
  }
  return o;
}
```

Pattern Match/Replace



$\pi_{R.\xcancel{A}T.E}$

$\sigma_{(R.B=S.B)\wedge(S.C<5)\wedge(S.D=T.D)}$

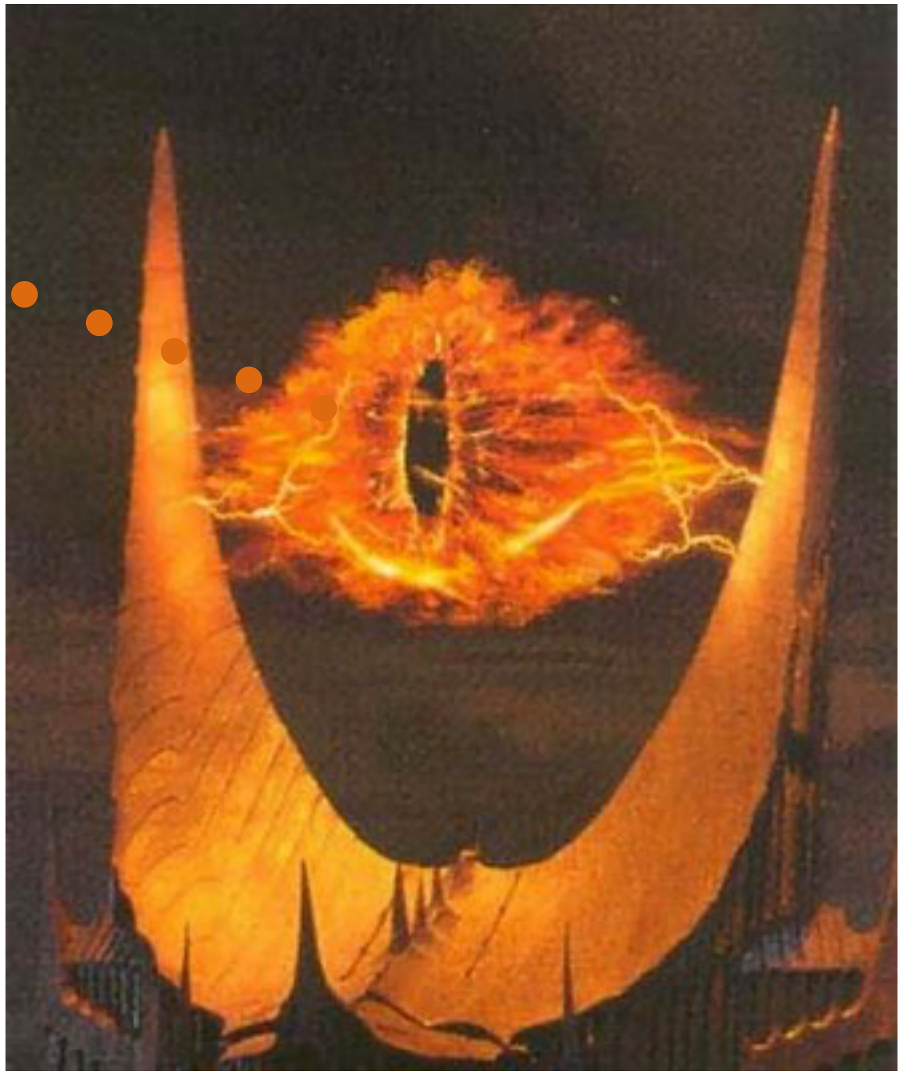
X

X

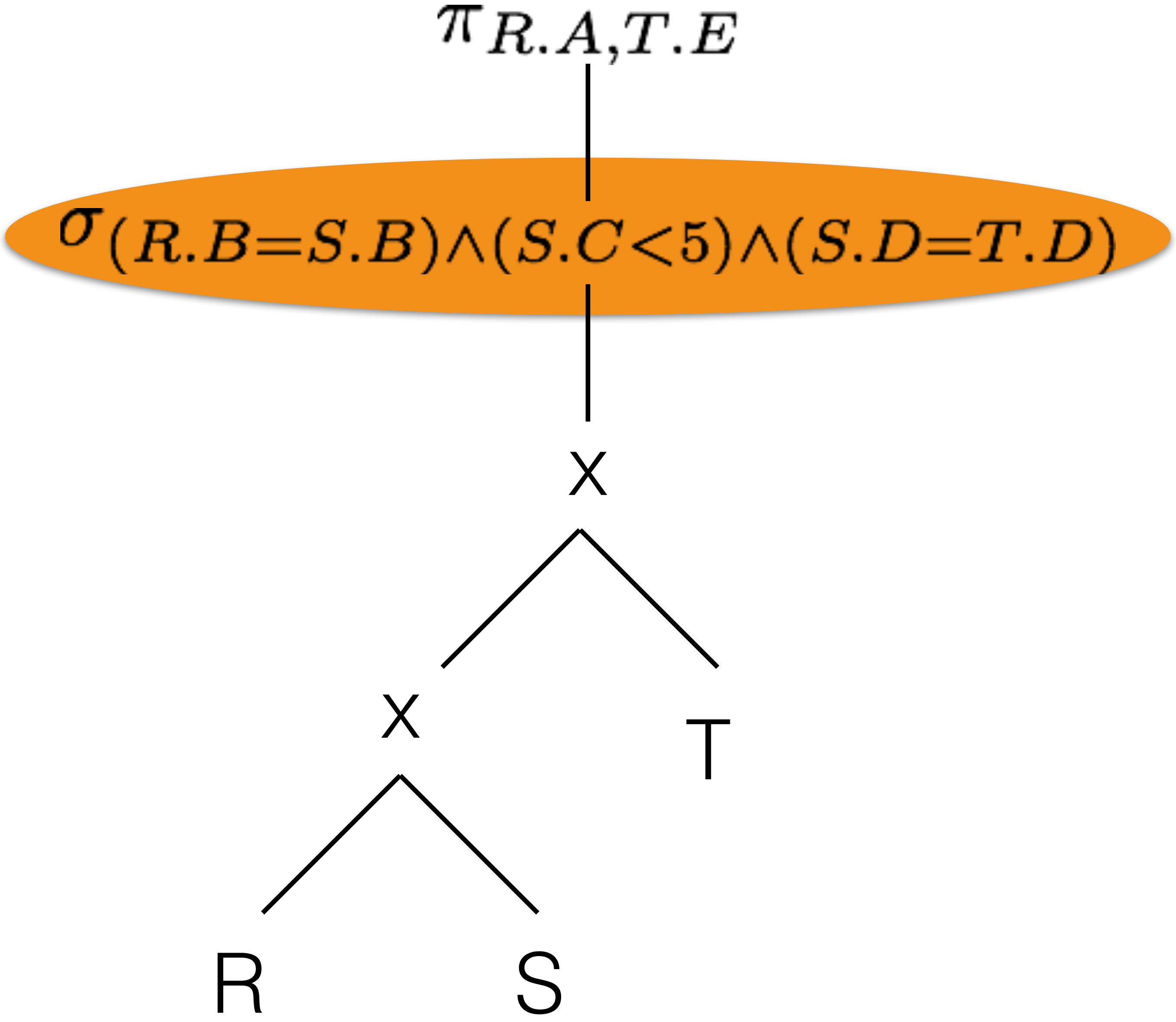
T

R

S



Pattern Match/Replace



$\pi_{R.A,T.E}$

$\sigma_{(R.B=S.B)\wedge(S.C<5)\wedge(S.D=T.D)}$

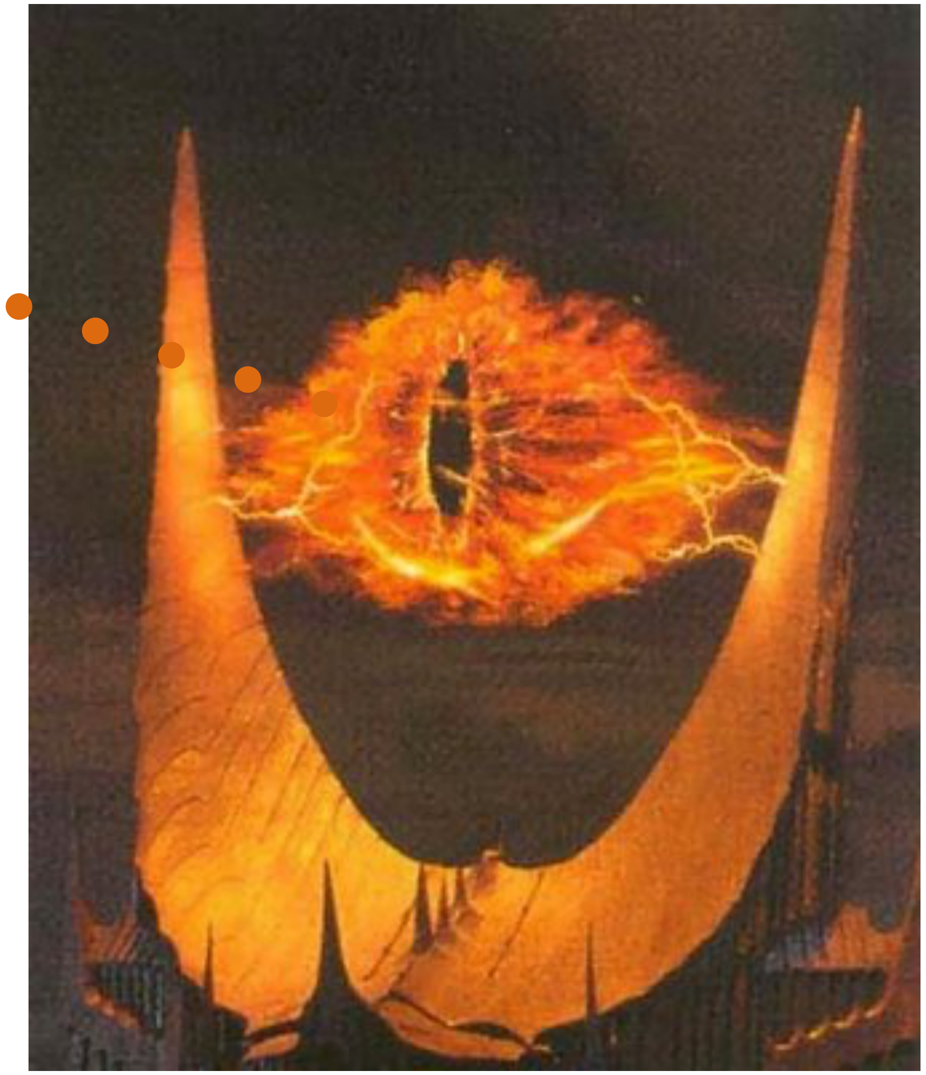
X

X

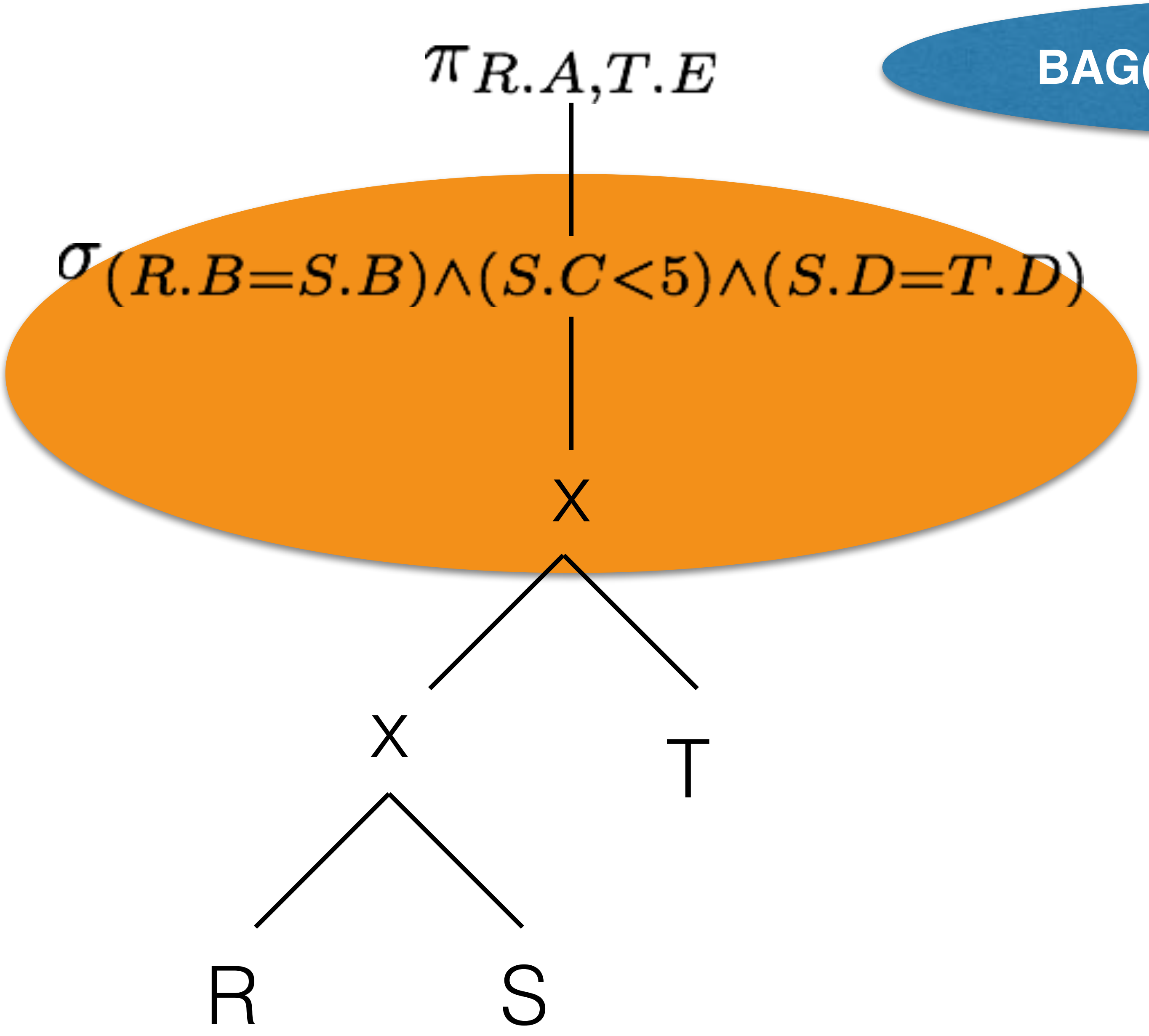
T

R

S



Pattern Match/Replace



$\pi_{R.A,T.E}$

$\sigma_{(R.B=S.B)\wedge(S.C<5)\wedge(S.D=T.D)}$

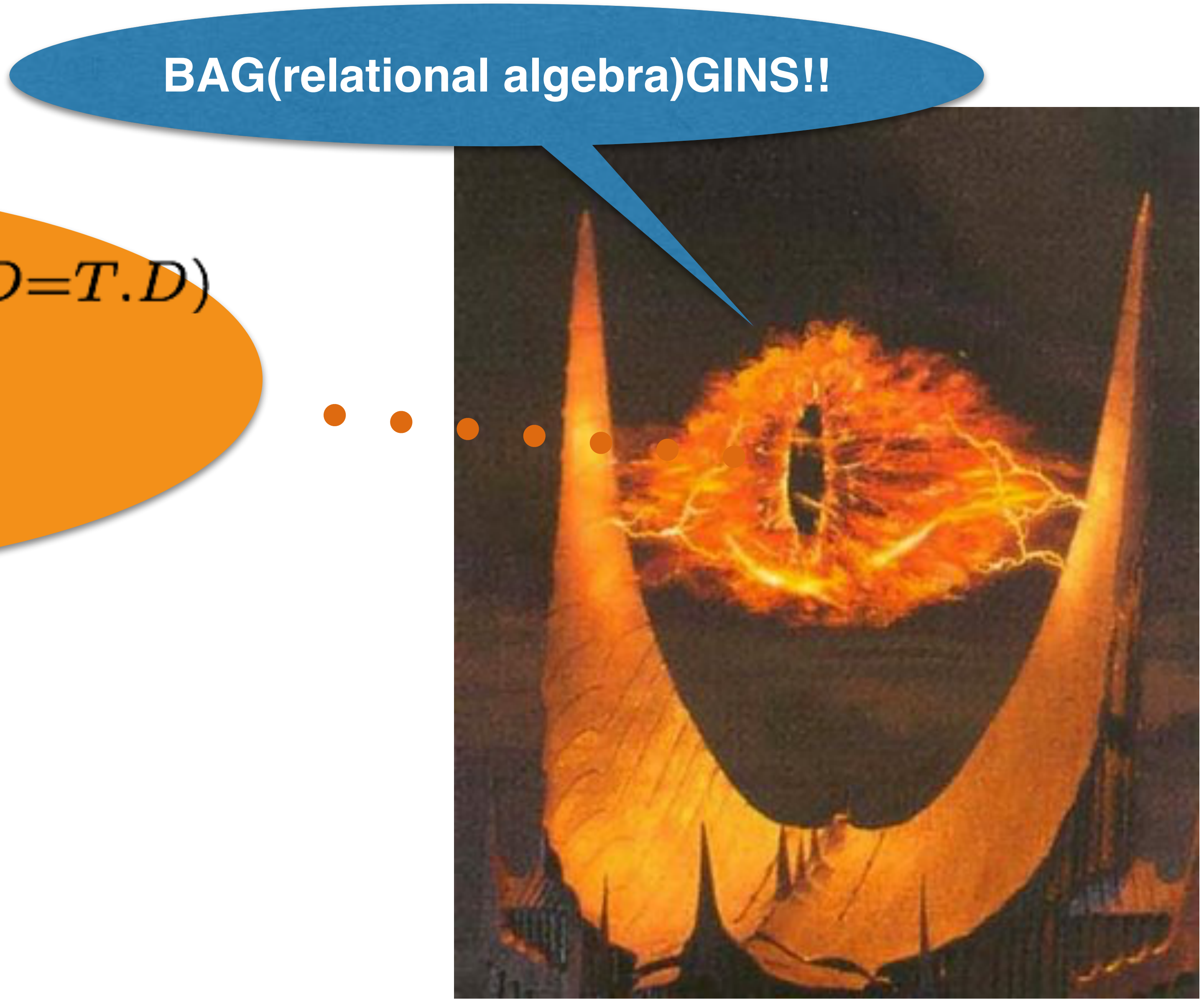
X

X

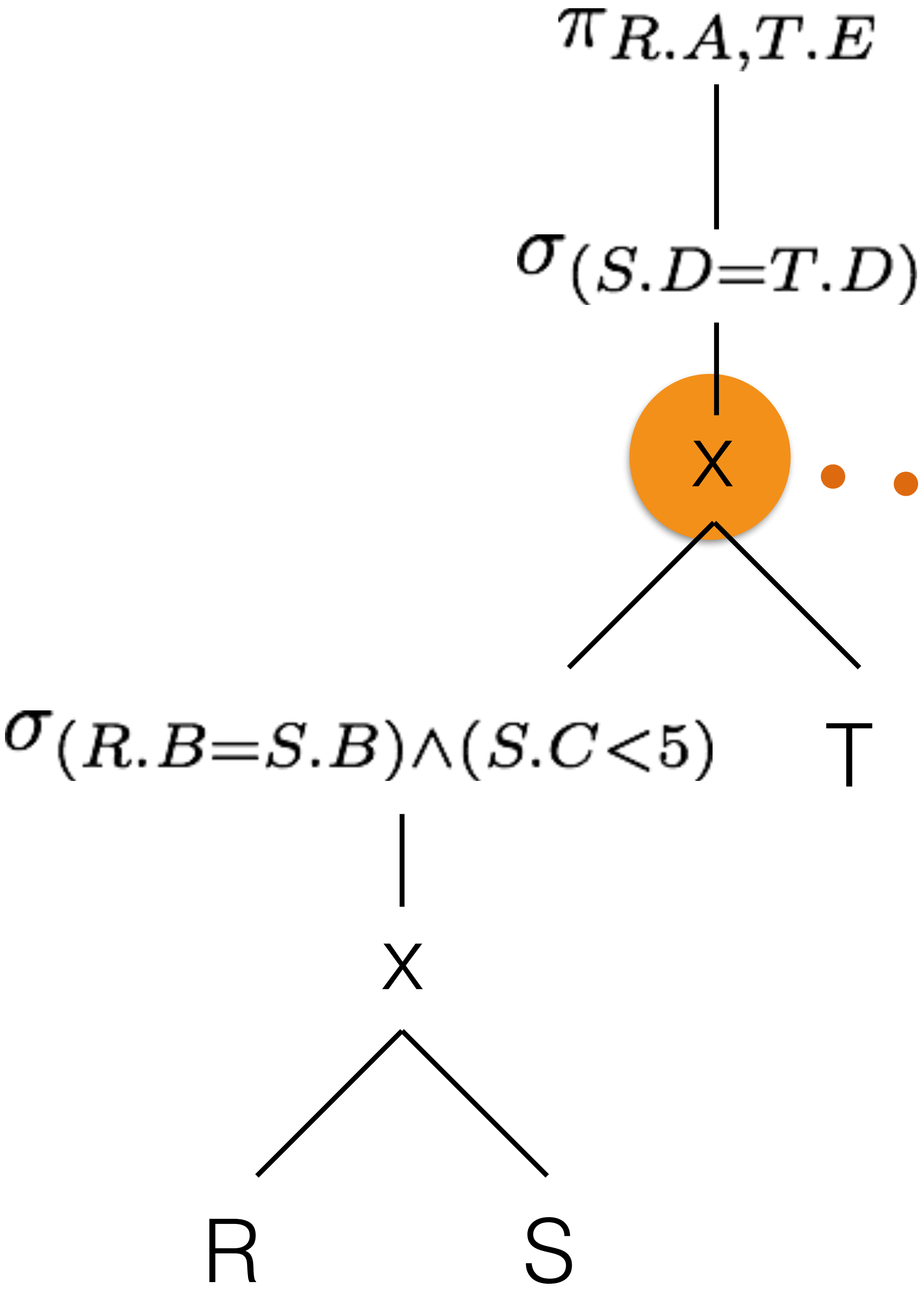
T

R

S



Pattern Match/Replace



$\pi_{R.A,T.E}$

$\sigma_{(S.D=T.D)}$

X

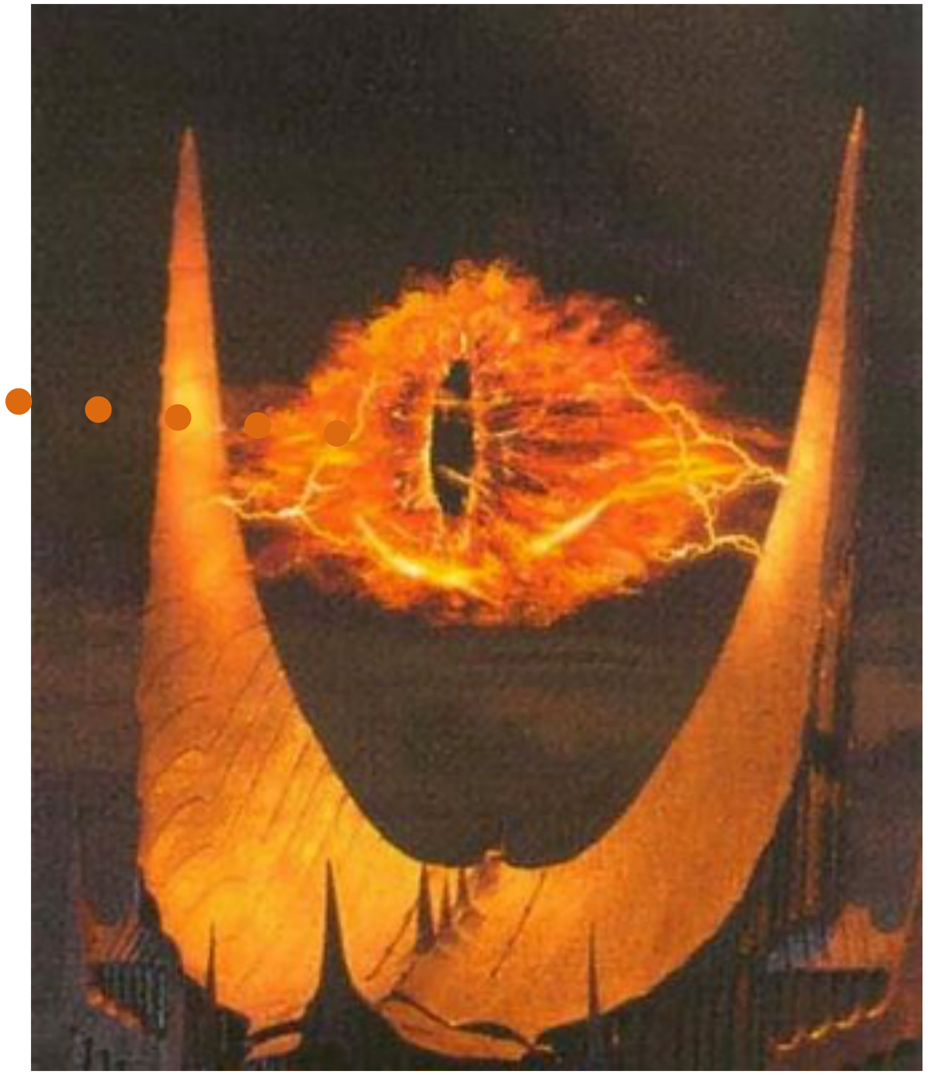
$\sigma_{(R.B=S.B)\wedge(S.C<5)}$

T

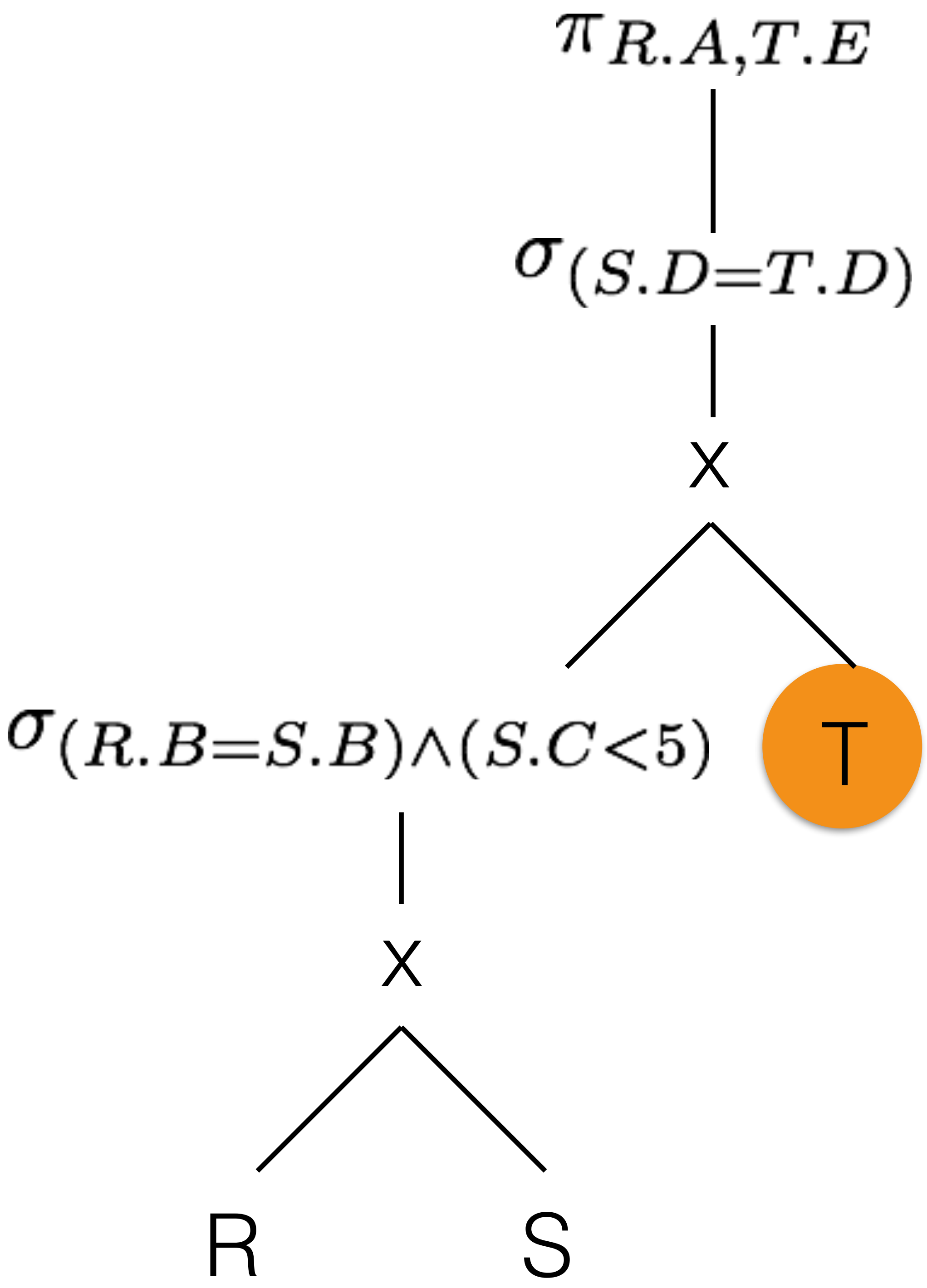
X

R

S



Pattern Match/Replace



$\pi_{R.A,T.E}$

$\sigma_{(S.D=T.D)}$

X

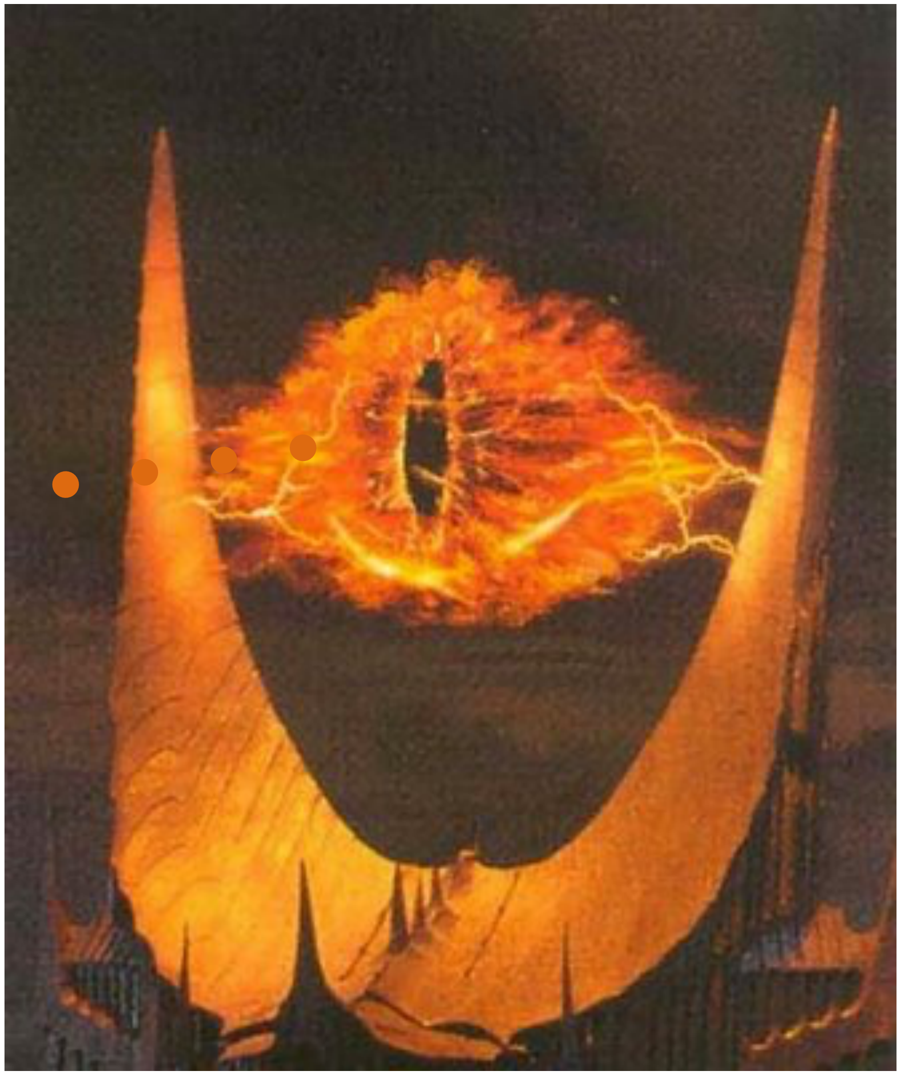
$\sigma_{(R.B=S.B)\wedge(S.C<5)}$

T

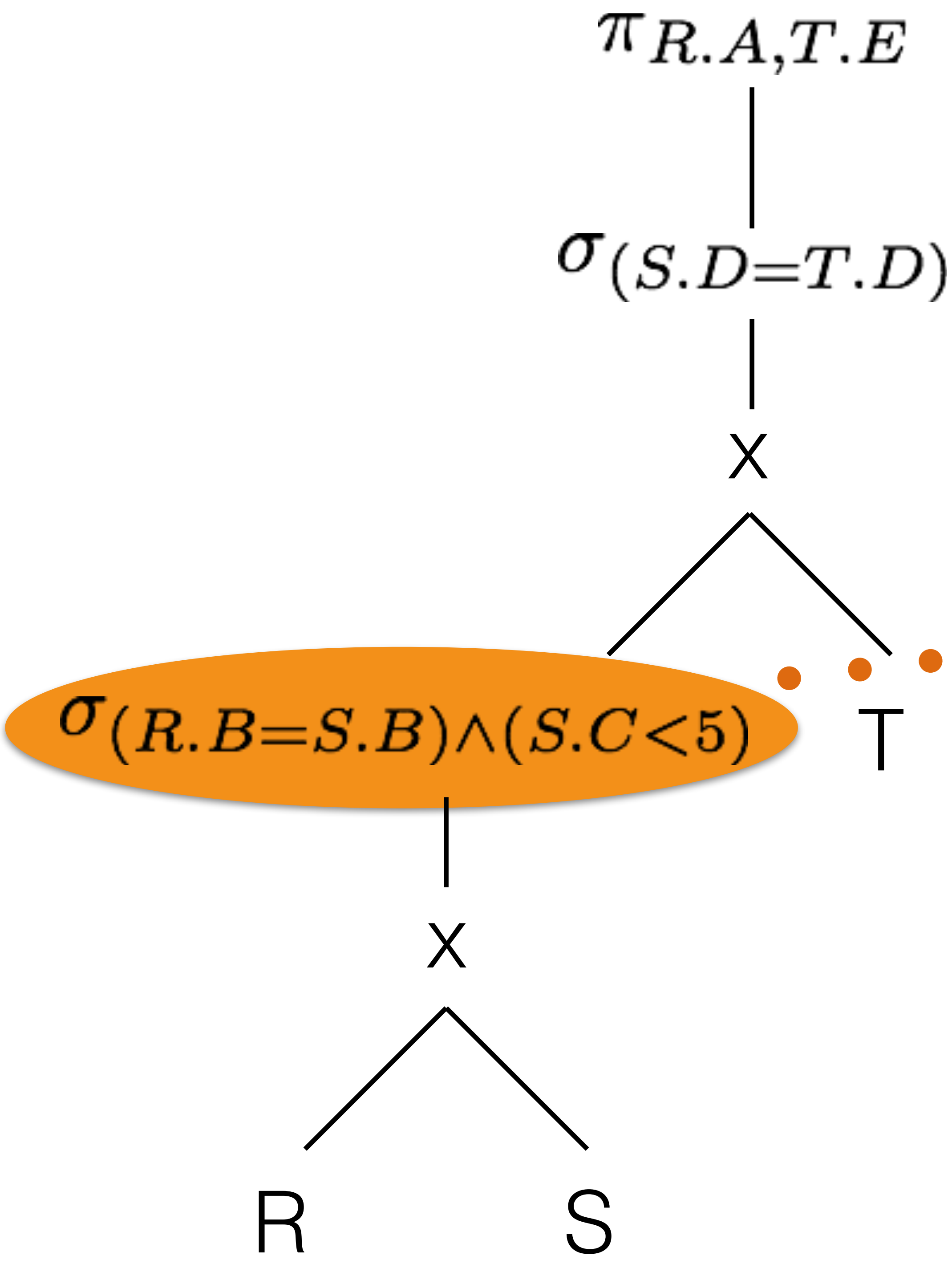
X

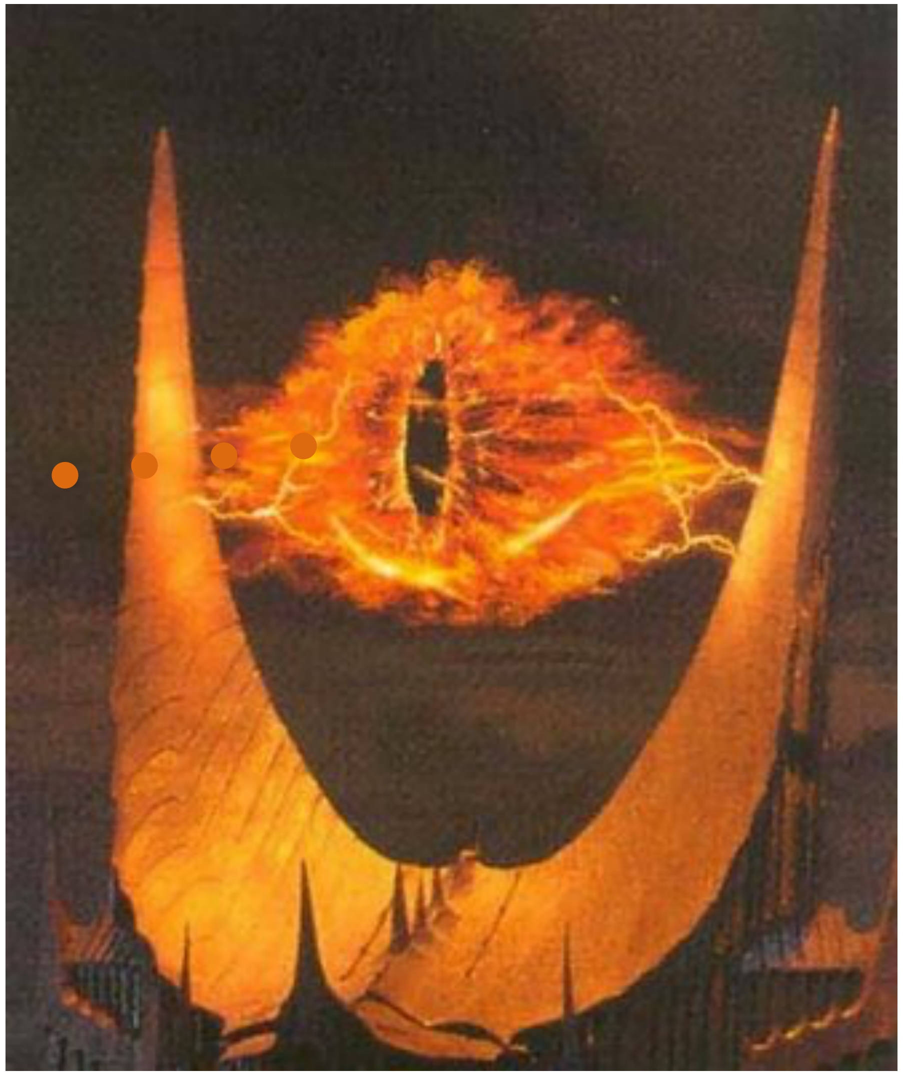
R

S

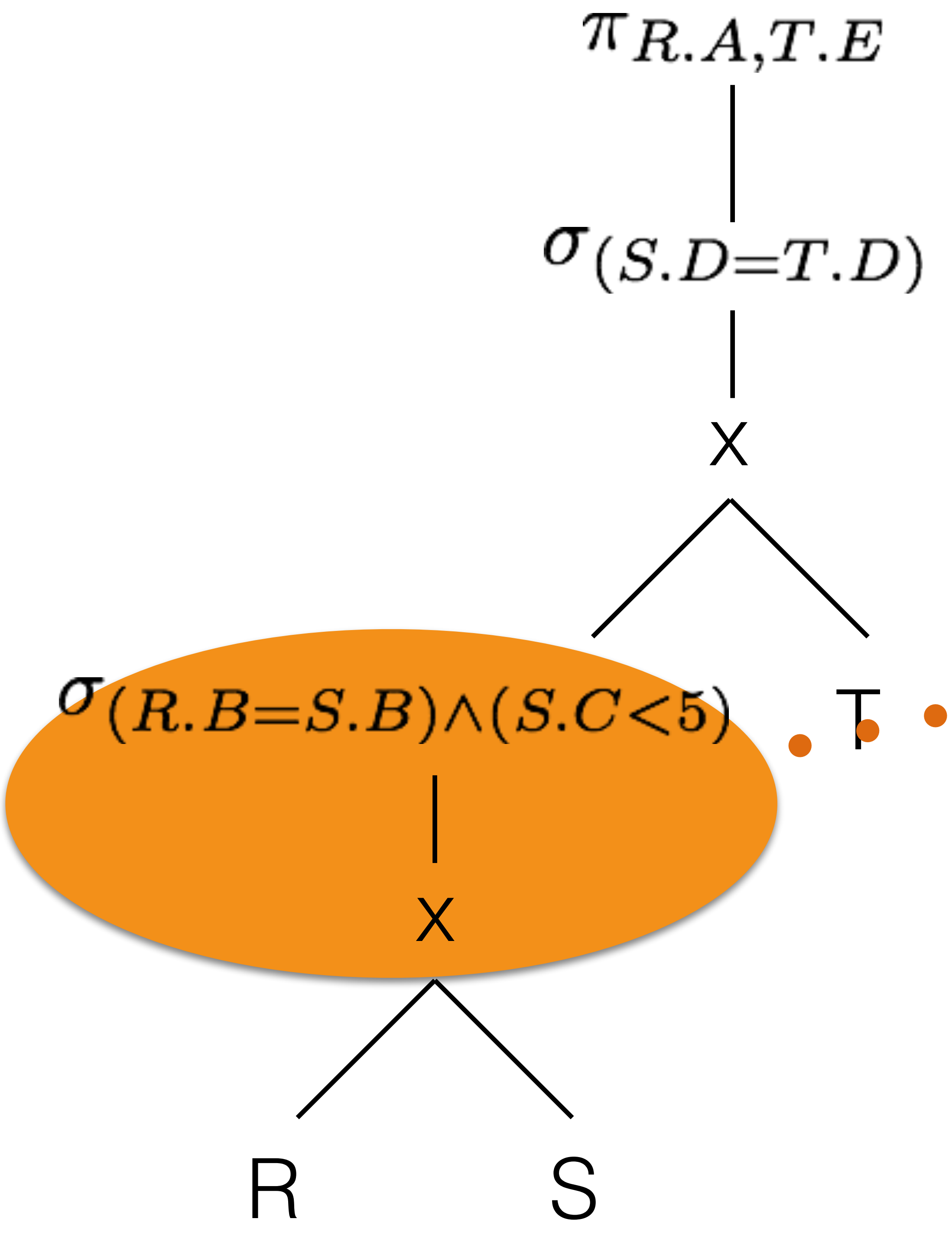


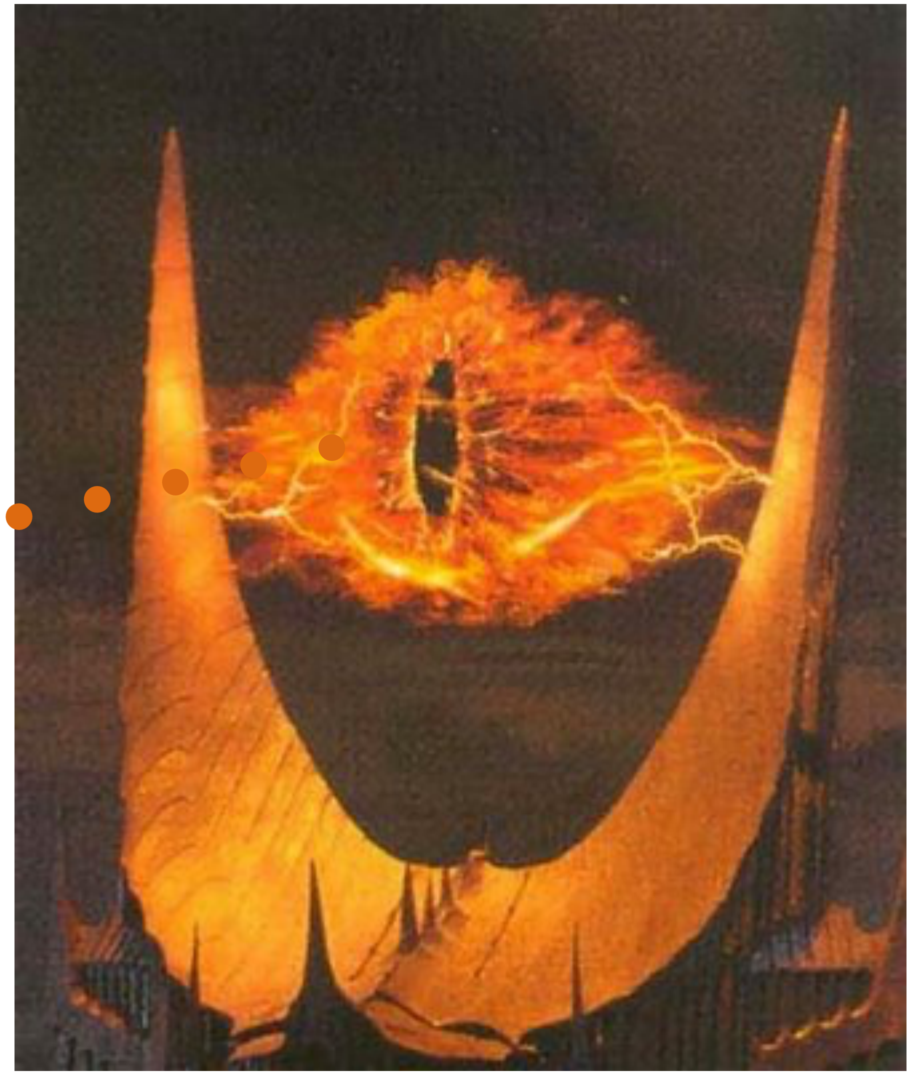
Pattern Match/Replace





Pattern Match/Replace



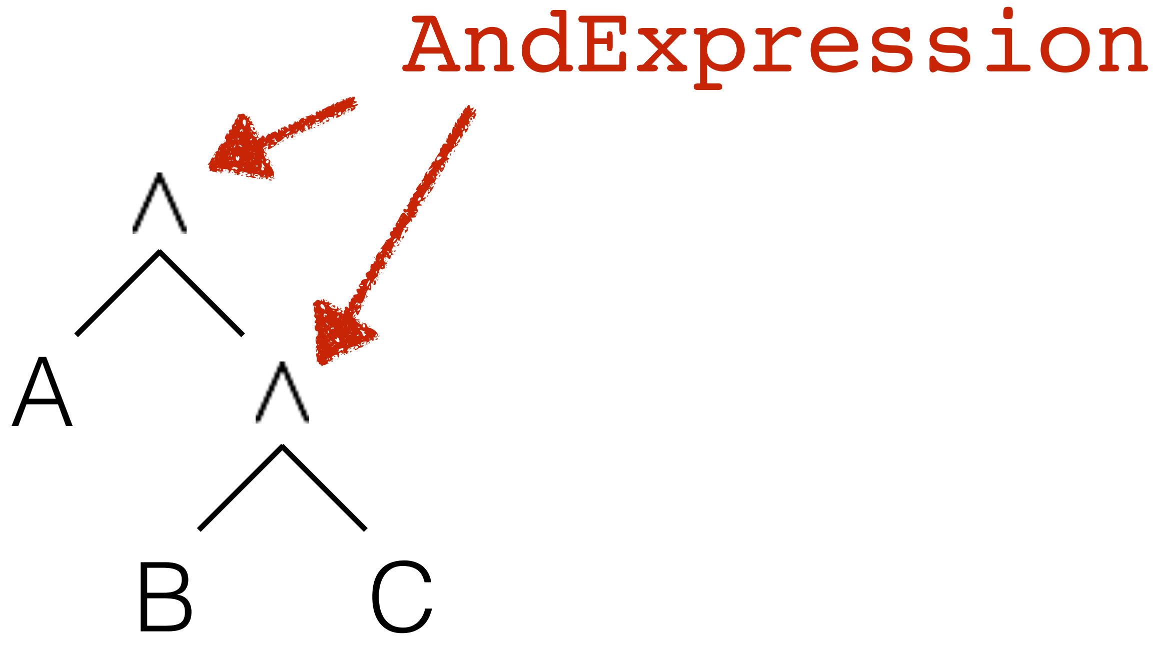


And so on…

Conjunctive Clauses

$A \land B \land C$

Conjunctive Clauses



AndExpression

Conjunctive Clauses

```
List<Expression> andClauses(Expression e){
  if(e instanceof AndExpression) {
    AndExpression a = (AndExpression)e;
    return
      andClauses(a.getLeftExpression()).
        addAll(
          andClauses(a.getRightExpression())
        );
  } else {
    return new List(e);
  }
}
```

Expression Schemas

- Does the clause include only LHS columns?
  - **Push to the left**
- Does the clause include only RHS columns?
  - **Push to the right**
- Does the clause include both?
  - **Leave in place**

Pushing Down Selection

[[Optional]]

$$\sigma_C(\pi_{A_i \leftarrow e_i}(R)) \equiv \pi_{A_i \leftarrow e_i}(\sigma_{C[A_i \backslash e_i]}(R))$$

Replace columns A_i in C with the corresponding expression e_i.

Build Joins

- **Add a New Operator:** InMemHashJoin
- Start with a simple case for selections:
    - `if(clause instanceof EqualsTo) { … }`
        - Replace Select+Product with a HashJoin
- More complex checks are possible…
    - … but you'll quickly hit diminishing returns.

Other Optimizations

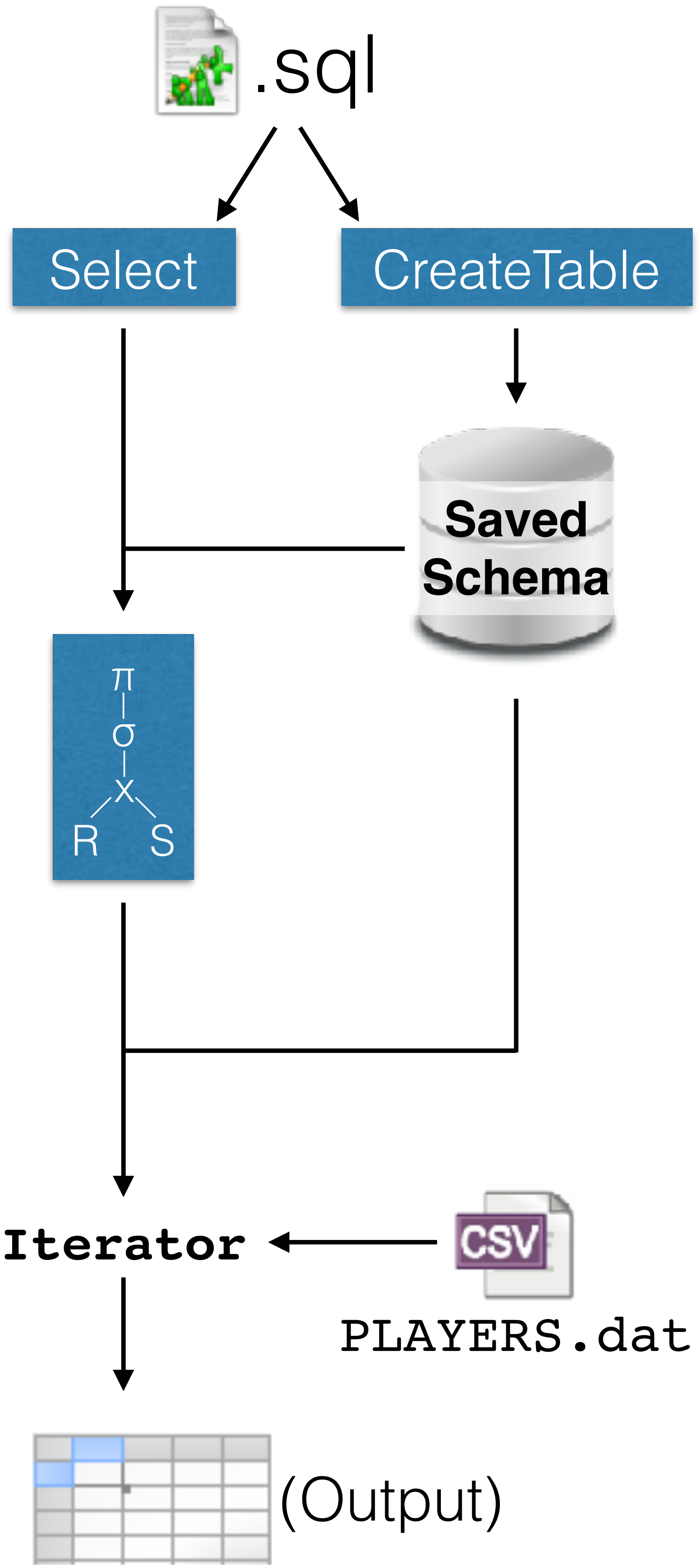
- Partially Evaluate Expressions
  - useful with pushing selections through projections
- Push Down Projections
  - useful if your relation scan operator is projection-aware
- Reorder Joins
  - hard to do unless you gather statistics…

When to “stop” optimizing

- Apply all optimization rules once **(ref impl does this)**
    - Be aware what order to apply them in.
    - Be aware of top-down vs bottom-up opts.
- Apply all rules N times.
- Apply rules up to a fixed point.

TPC-H

- http://www.tpc.org/tpch/
    - Checkpoint 2 on-disk queries taken from TPC-H
    - All Checkpoint 3 queries taken from TPC-H



.sql

Select

CreateTable

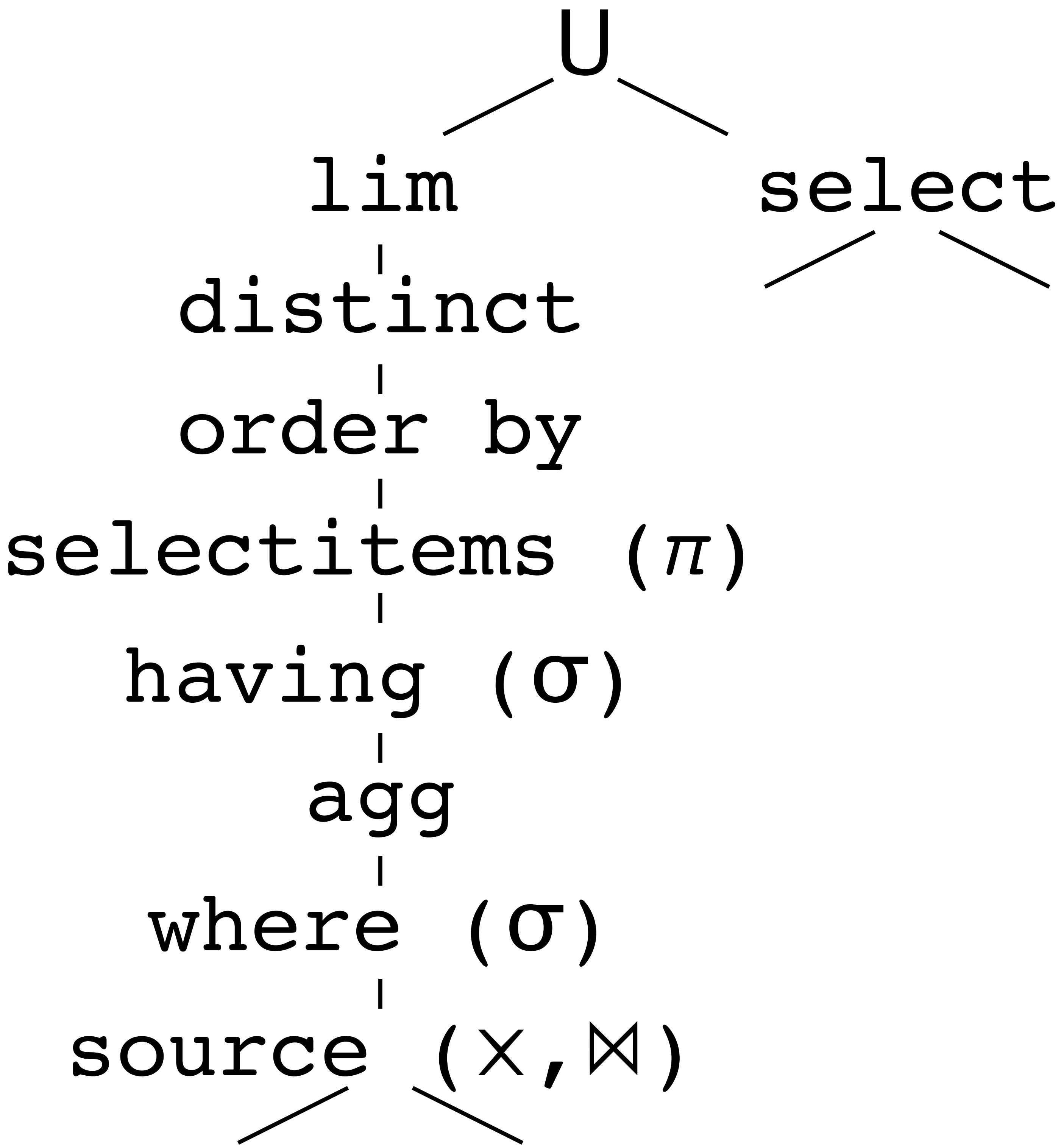
Saved Schema

Iterator

CSV

PLAYERS.dat

(Output)



∪

lim

select

distinct

order by

selectitems (π)

having (σ)

agg

where (σ)

source (×,⋈)