DATA ANALYTICS USING DEEP LEARNING

GT 8803 // FALL 2019 // JOY ARULRAJ

LECTURE #09:QUERY OPTIMIZATION

Georgia Tech

CREATING THE NEXT®

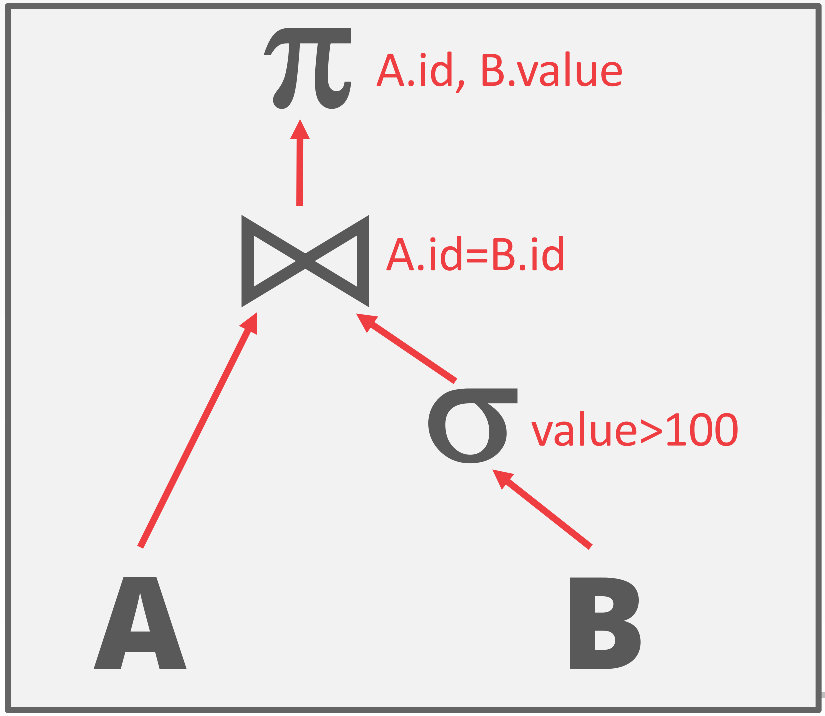
ADMINISTRIVIA

- Reminders
  - Assignment 1: postponed to next Monday
  - Sign up for discussion slots on Thursday
  - Proposal presentations on next Wednesday

LAST CLASS

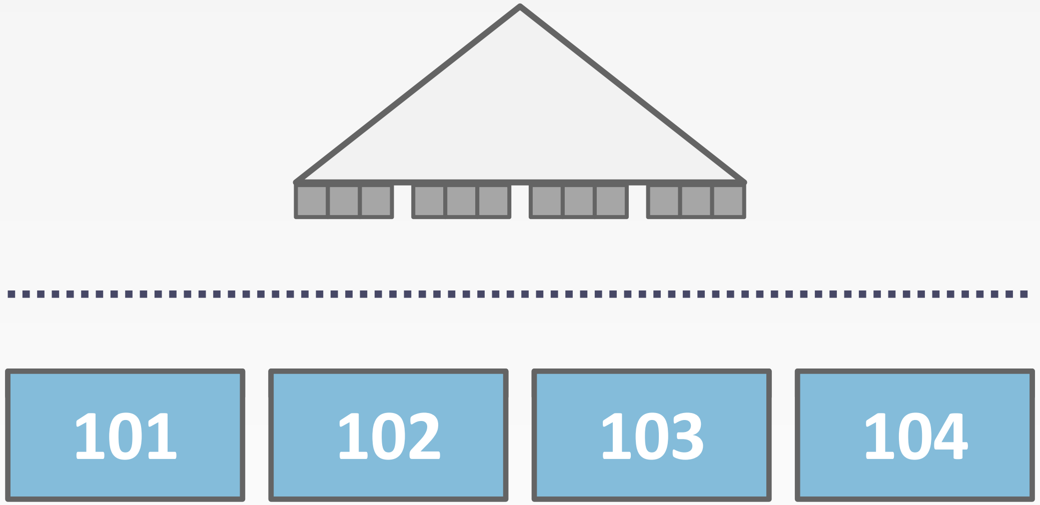
- Query execution models
  - Tuple-at-a-time
  - Operator-at-a-time
  - Vector-at-a-time

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



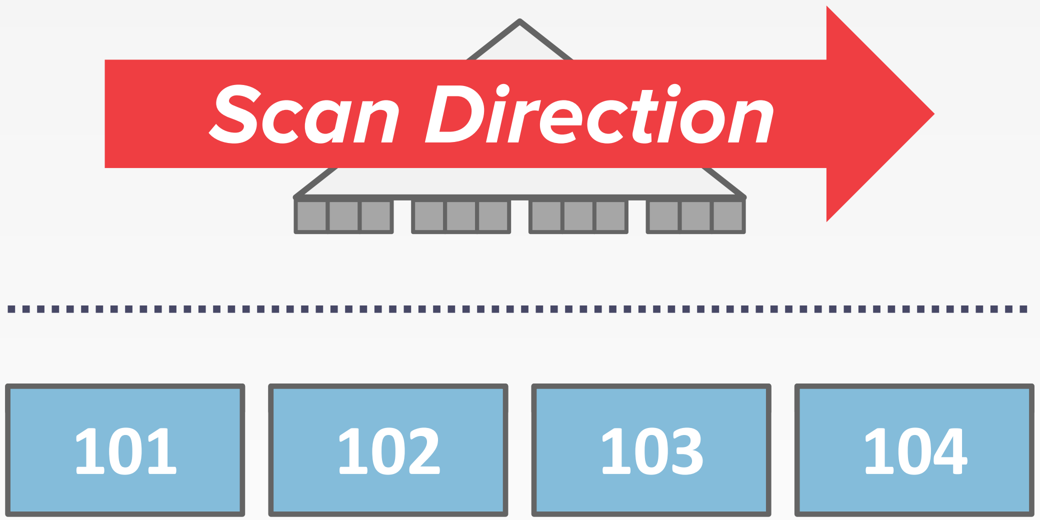
LAST CLASS

- Access methods
  - Sequential scan
  - Index scan
  - Multi-index scan



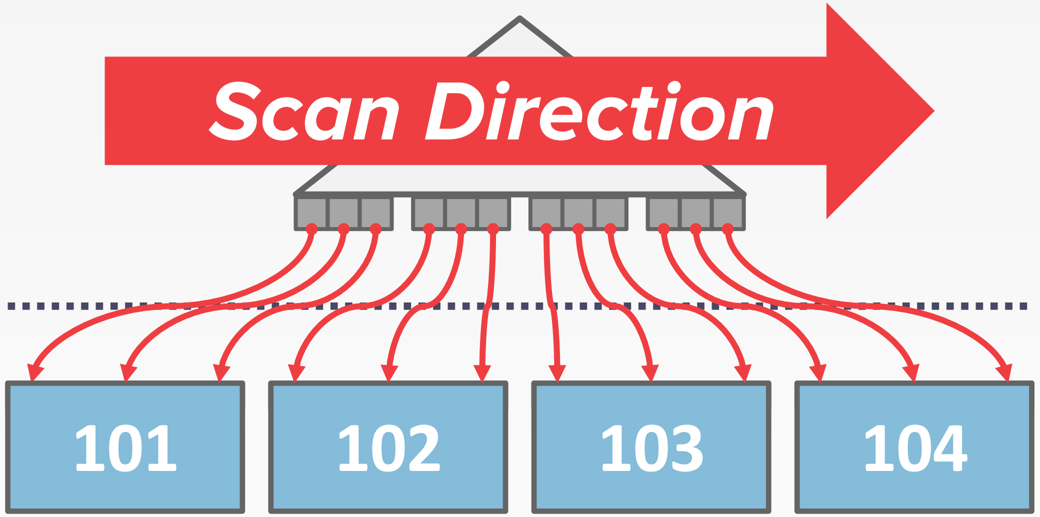
LAST CLASS

- Access methods
  - Sequential scan
  - Index scan
  - Multi-index scan



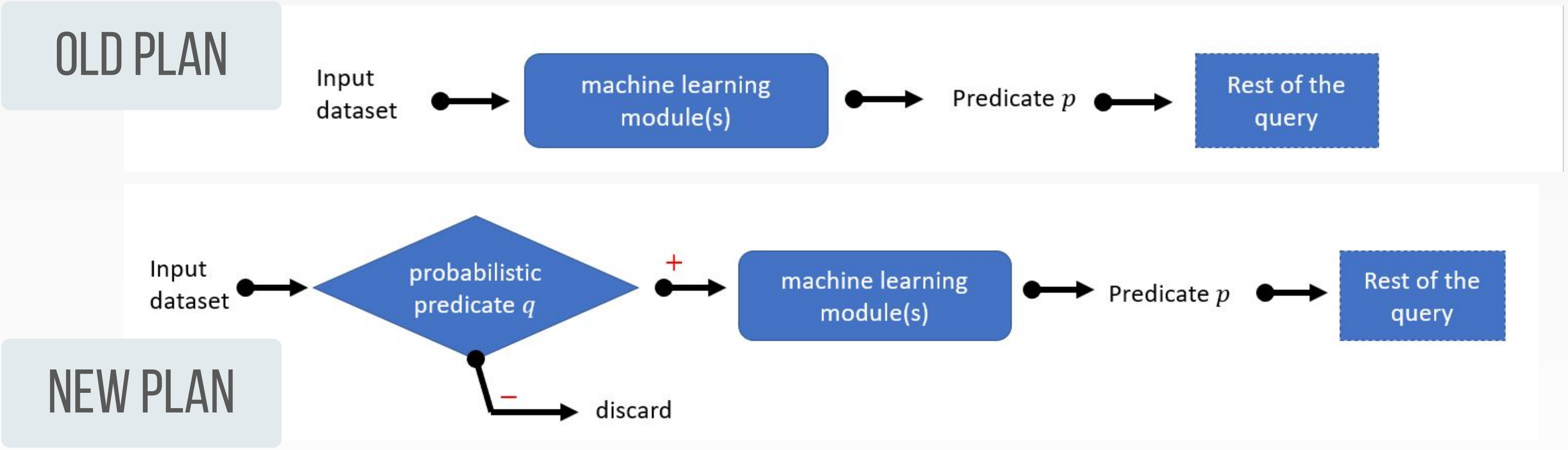
LAST CLASS

- Access methods
  – Sequential scan
  – Index scan
  – Multi-index scan



LAST CLASS

- Visual Query Execution Engine
  - Filtering classifier, Sampling



TODAY'S AGENDA

- Relational Algebra Equivalences
- Plan Cost Estimation
- Plan Enumeration
- Visual Query Optimizer

RELATIONAL ALGEBRA EQUIVALENCES

ANATOMY OF A DATABASE SYSTEM

Query		
	Connection Manager + Admission Control	*Process Manager*
	Query Parser	*Query Processor*
	Query Optimizer	
	Query Executor	
	Lock Manager (Concurrency Control)	*Transactional Storage Manager*
	Access Methods (or Indexes)	
	Buffer Pool Manager	
	Log Manager	
	Memory Manager + Disk Manager	*Shared Utilities*
	Networking Manager	

Source: Anatomy of a Database System

QUERY OPTIMIZATION

- Remember that SQL is declarative.
  - User tells the DBMS what answer they want, not how to get the answer.
- There can be a big difference in performance based on plan is used:
  - 1.3 hours vs. 0.45 seconds

IBM SYSTEM R

- First implementation of a query optimizer. People argued that the DBMS could never choose a query plan better than what a human could write.

- A lot of the concepts from **System R**'s optimizer are still used today.

QUERY OPTIMIZATION

- **Rule-based Optimizer**
  - Rewrite the query to remove inefficient things.
  - Does not require a cost model.

- **Cost-based Optimizer**
  - Use a cost model to evaluate multiple equivalent plans and pick the one with the lowest cost.

QUERY OPTIMIZATION: OVERVIEW

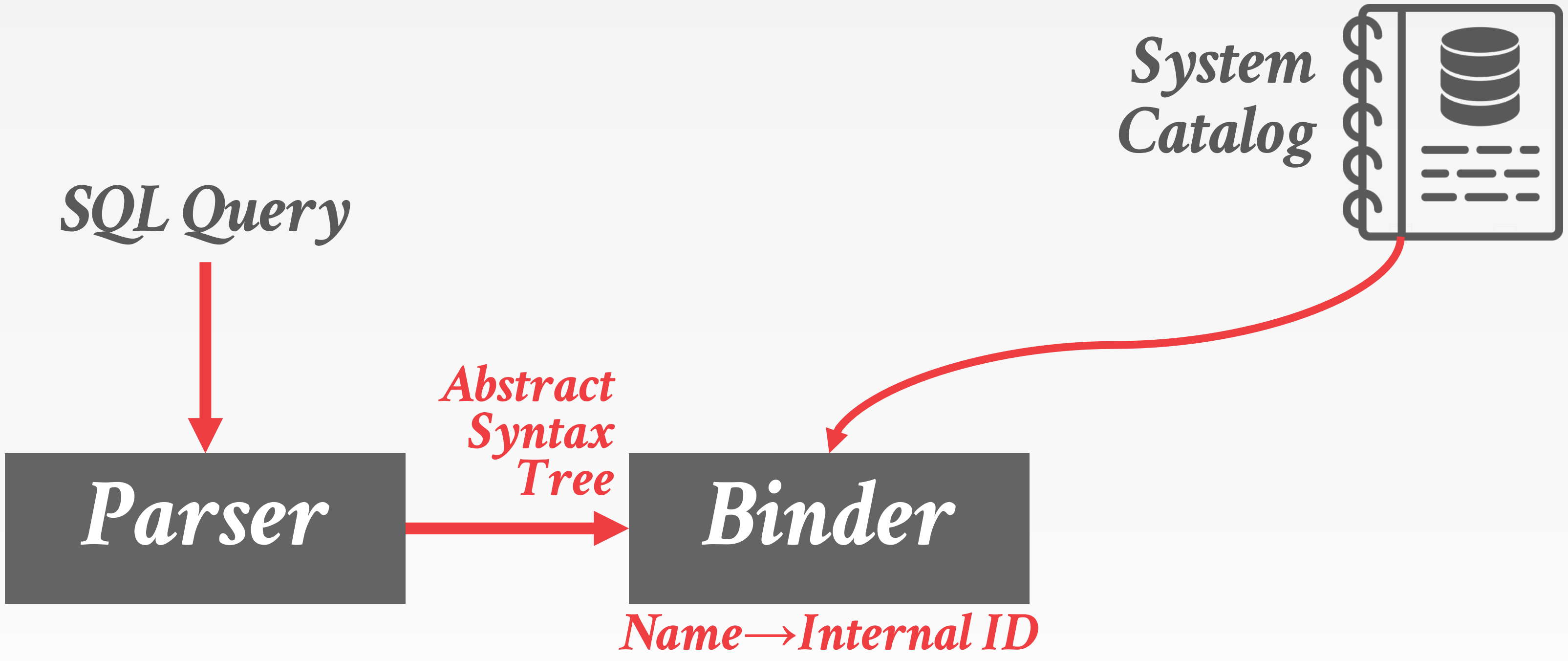
QUERY OPTIMIZATION: OVERVIEW

SQL Query

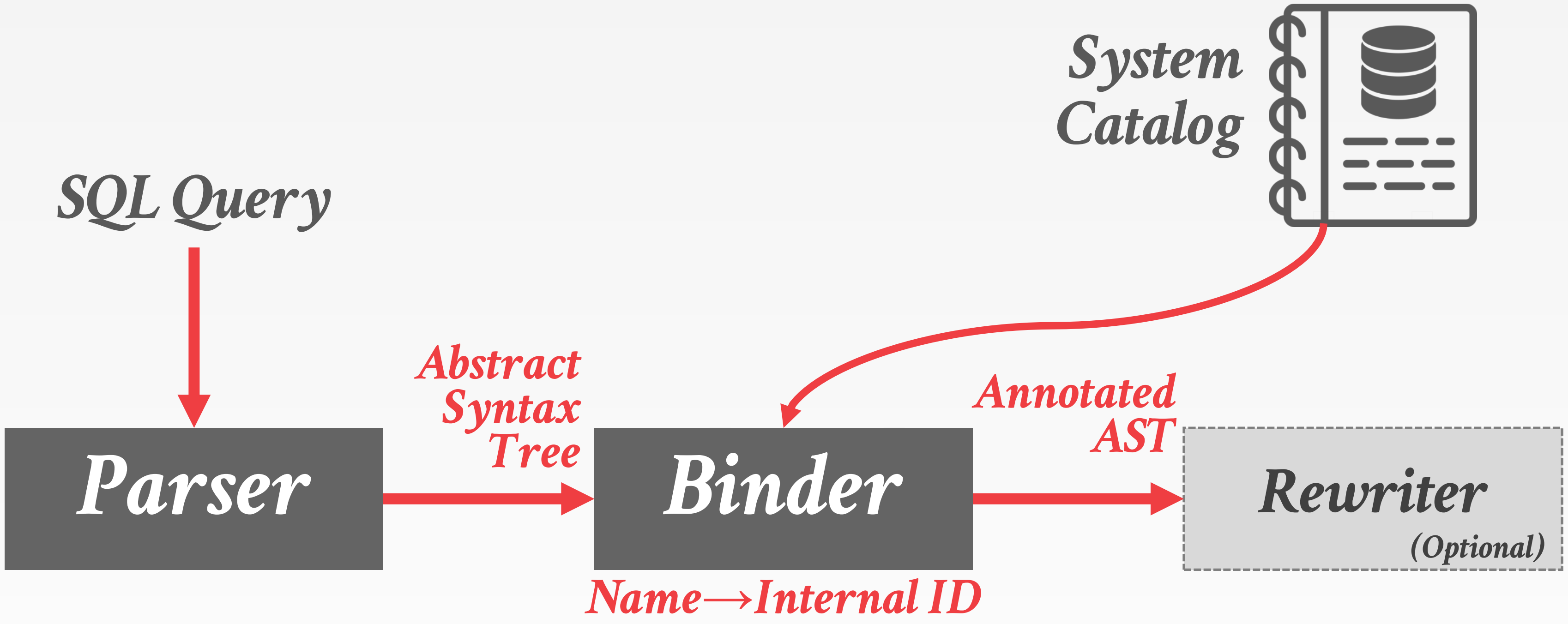
Parser

QUERY OPTIMIZATION: OVERVIEW

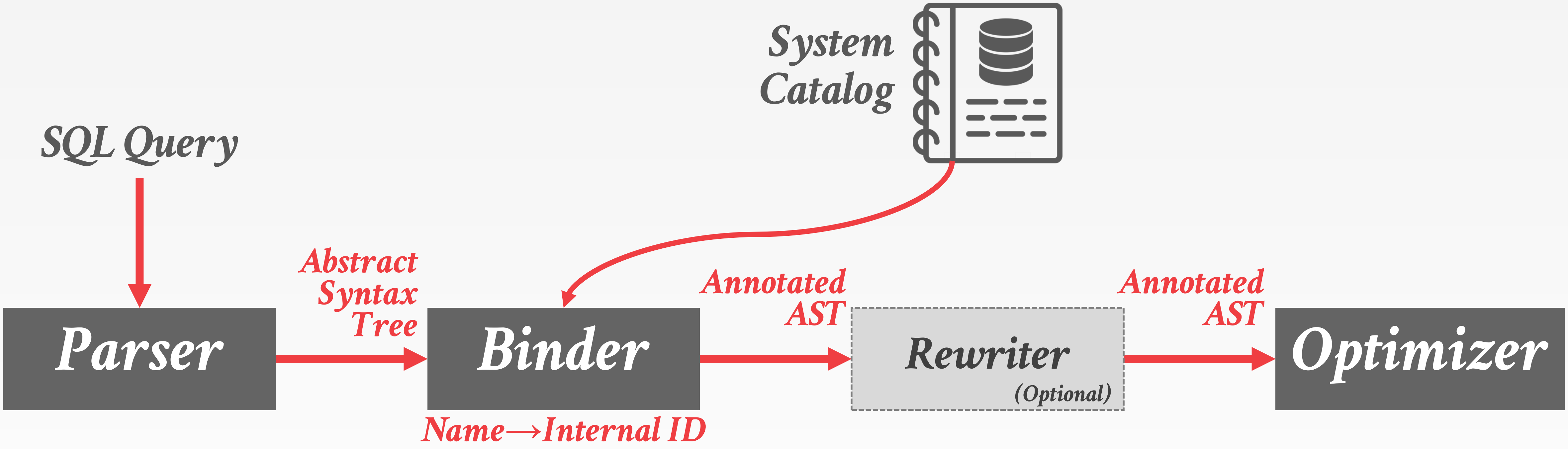
QUERY OPTIMIZATION: OVERVIEW



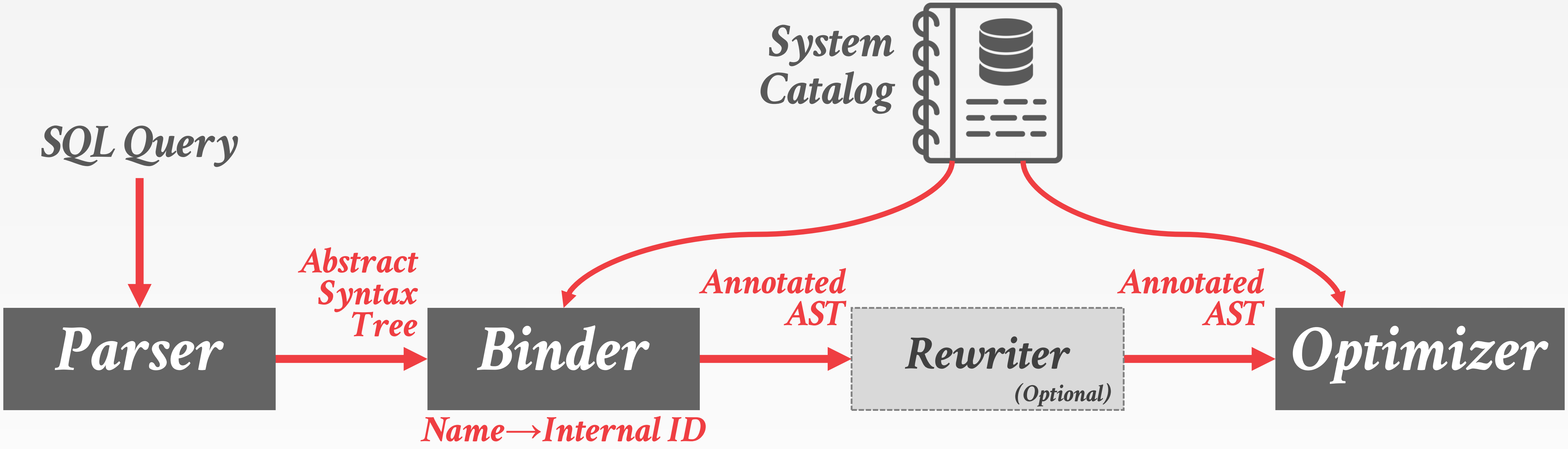
QUERY OPTIMIZATION: OVERVIEW



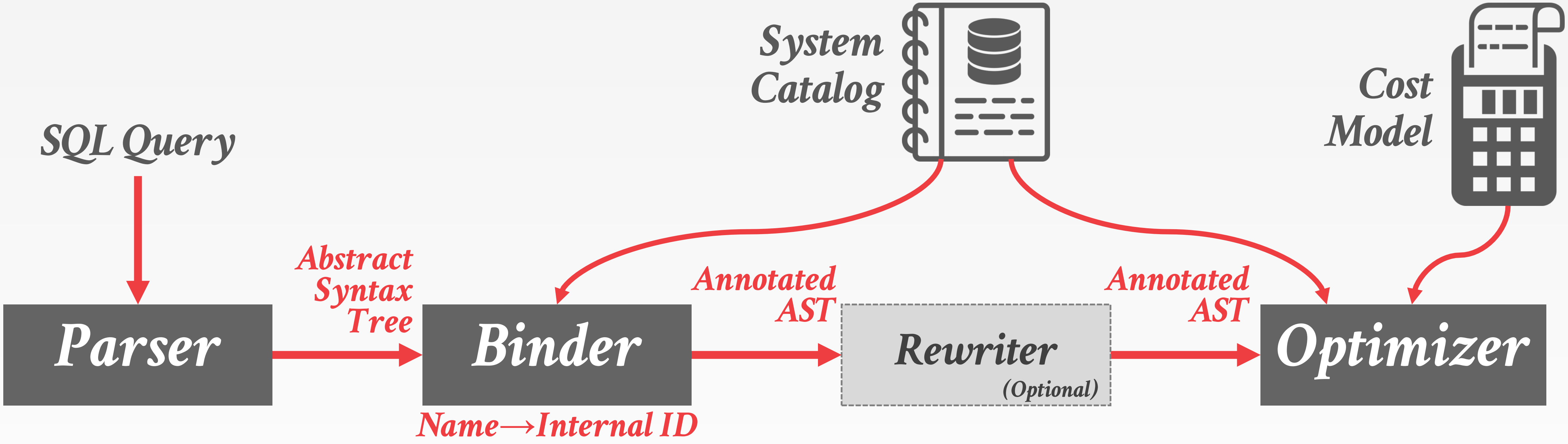
QUERY OPTIMIZATION: OVERVIEW



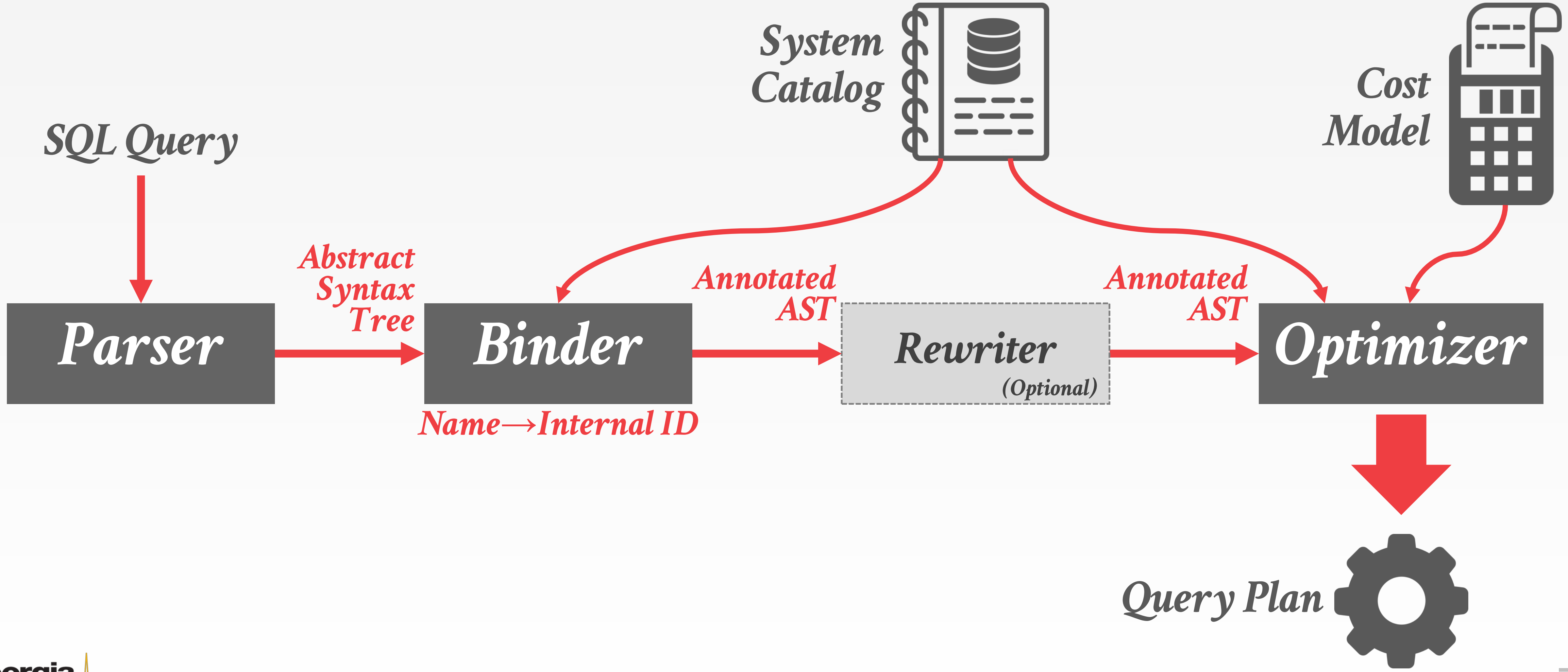
QUERY OPTIMIZATION: OVERVIEW



QUERY OPTIMIZATION: OVERVIEW



QUERY OPTIMIZATION: OVERVIEW



QUERY OPTIMIZATION IS NP-HARD

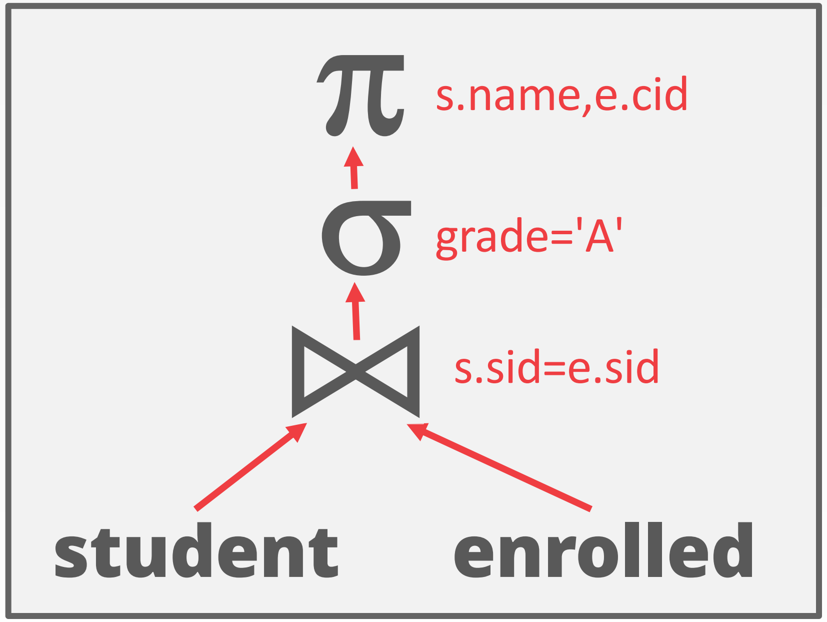
- This is the hardest part of building a DBMS.
- If you are good at this, you will get paid.
- People are starting to look at employing ML to improve the accuracy and efficacy of optimizers.

RELATIONAL ALGEBRA EQUIVALENCES

- Two relational algebra expressions are equivalent if they generate the same set of tuples.
  - The DBMS can identify better query plans **without** a cost model.
  - This is often called **query rewriting**.

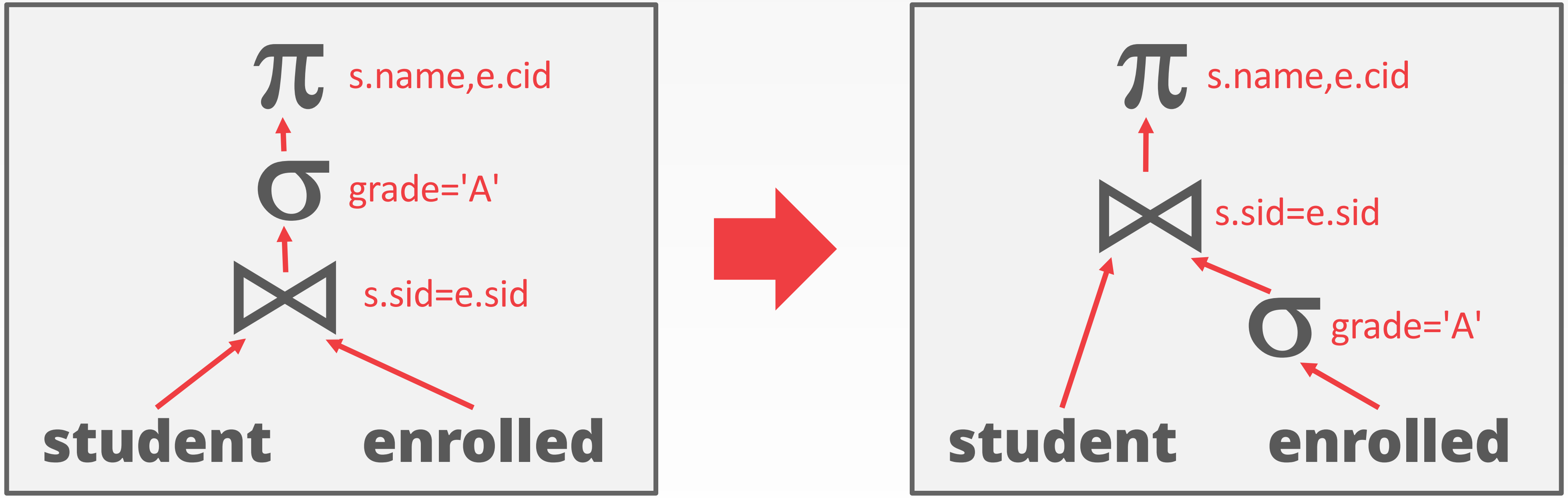
PREDICATE PUSHDOWN

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```



PREDICATE PUSHDOWN

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```



RELATIONAL ALGEBRA EQUIVALENCES

```sql
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

$$\pi_{\text{name, cid}}(\sigma_{\text{grade='A'}}(\text{student} \bowtie \text{enrolled}))$$

$$=$$

$$\pi_{\text{name, cid}}(\text{student} \bowtie (\sigma_{\text{grade='A'}}(\text{enrolled})))$$

RELATIONAL ALGEBRA EQUIVALENCES

- **Selections:**
  - Perform filters as early as possible.
  - Reorder predicates so that the DBMS applies the most selective one first.
  - Break a complex predicate, and push down

    $$\sigma_{p1 \wedge p2 \wedge \ldots pn}(\mathbf{R}) = \sigma_{p1}(\sigma_{p2}(\ldots\sigma_{pn}(\mathbf{R})))$$

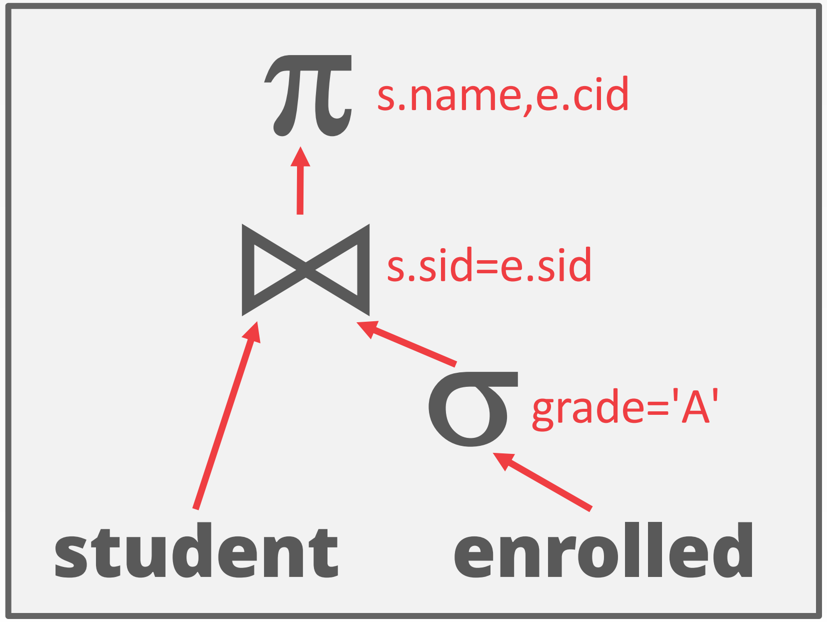
- Simplify a complex predicate
  - **(X=Y AND Y=3) → X=3 AND Y=3**

RELATIONAL ALGEBRA EQUIVALENCES

- **Projections:**
  - Perform them early to create smaller tuples and reduce intermediate results (if duplicates are eliminated)
  - Project out all attributes except the ones requested or required (e.g., joining keys)

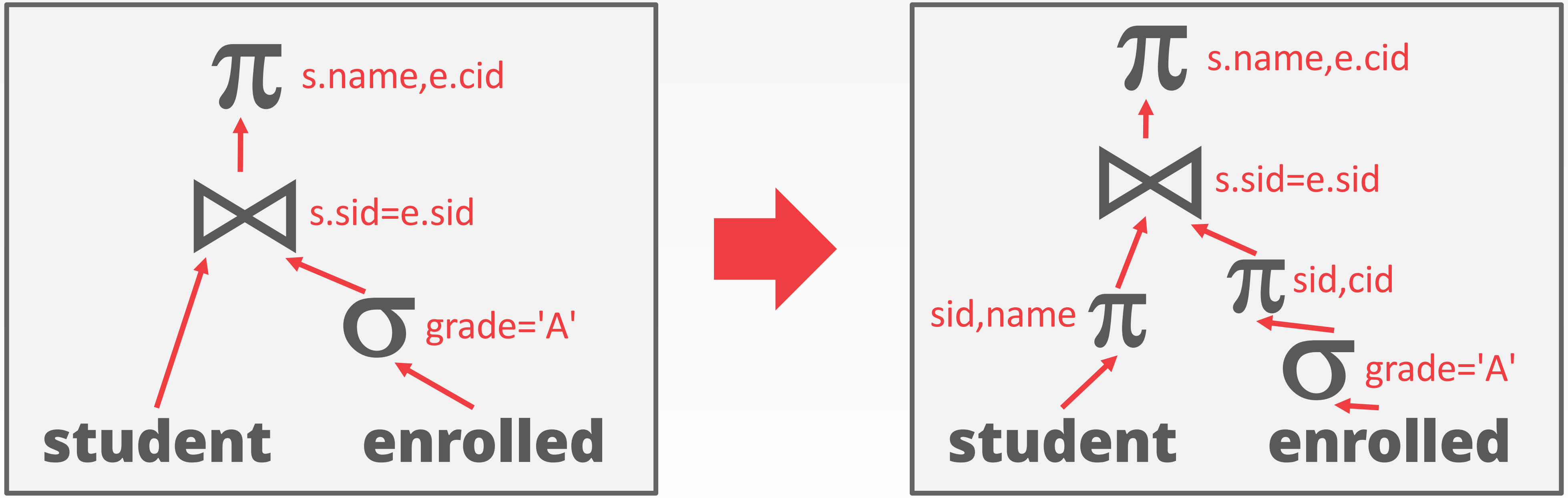
PROJECTION PUSHDOWN

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```



PROJECTION PUSHDOWN

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```



```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```
SELECT * FROM A WHERE 1 = 0;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```
SELECT * FROM A WHERE 1 = 0;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```
SELECT * FROM A WHERE 1 = 0;
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```sql
SELECT * FROM A WHERE 1 = 0;
```

X

```sql
SELECT * FROM A WHERE 1 = 1;
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```sql
SELECT * FROM A WHERE 1 = 0;
```

X

```sql
SELECT * FROM A WHERE 1 = 1;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```
SELECT * FROM A WHERE 1 = 0; X
```

```
SELECT * FROM A;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
SELECT * FROM A;
```

- Join Elimination

```
SELECT A1.*
  FROM A AS A1 JOIN A AS A2
    ON A1.id = A2.id;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
SELECT * FROM A;
```

- Join Elimination

```
SELECT A1.*
  FROM A AS A1 JOIN A AS A2
    ON A1.id = A2.id;
```

Source: Lukas Eder

Georgia Tech

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

X

```
SELECT * FROM A;
```

- Join Elimination

```
SELECT * FROM A;
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```sql
SELECT * FROM A AS A1
 WHERE EXISTS(SELECT * FROM A AS A2
               WHERE A1.id = A2.id);
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```
SELECT * FROM A AS A1
 WHERE EXISTS(SELECT * FROM A AS A2
               WHERE A1.id = A2.id);
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

```sql
SELECT * FROM A;
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

- Ignoring Projections

```sql
SELECT * FROM A;
```

- Merging Predicates

```sql
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150;
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

- Ignoring Projections

```sql
SELECT * FROM A;
```

- Merging Predicates

```sql
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150;
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

- Ignoring Projections

```sql
SELECT * FROM A;
```

- Merging Predicates

```sql
SELECT * FROM A
 WHERE val BETWEEN 1 AND 150;
```

Source: Lukas Eder

RELATIONAL ALGEBRA EQUIVALENCES

- **Joins:**
  - Commutative, associative

    $R \bowtie S = S \bowtie R$

    $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
  - How many different orderings are there for an *n*-way join?

RELATIONAL ALGEBRA EQUIVALENCES

- How many different orderings are there for an *n*-way join?
- **Catalan number** $\approx 4^n$
  - Exhaustive enumeration will be too slow.
- We'll see in a second how an optimizer limits the search space.

PLAN COST ESTIMATION

COST ESTIMATION

- How long will a query take?
  - CPU: Small cost; tough to estimate
  - Disk: # of block transfers
  - Memory: Amount of DRAM used
- How many tuples will be read/written?
- What statistics do we need to keep?

STATISTICS

- The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.
- Different systems update them at different times.
- Manual invocations:
  - Postgres/SQLite: **ANALYZE**
  - SQL Server: **UPDATE STATISTICS**

STATISTICS

- For each relation **R**, the DBMS maintains the following information:
  - N_R: Number of tuples in **R**.
  - **V(A,R)**: Number of distinct values for attribute **A**.

DERIVABLE STATISTICS

DERIVABLE STATISTICS

- The **<u>selection cardinality</u>** **SC(A,R)** is the average number of records with a value for an attribute **A** given **N_R/V(A,R)**

- Note that this assumes ***data uniformity.***
  - 10,000 students, 10 colleges – how many students in SCS?

SELECTION STATISTICS

SELECTION STATISTICS

```
SELECT * FROM people
 WHERE id = 123
```

SELECTION STATISTICS

- Equality predicates on unique keys are easy to estimate.

```
SELECT * FROM people
 WHERE id = 123
```

- What about more complex predicates? What is their selectivity?

```
SELECT * FROM people
 WHERE val > 1000
```

```
SELECT * FROM people
 WHERE age = 30
   AND status = 'Lit'
```

COMPLEX PREDICATES

- The **selectivity** (**sel**) of a predicate **P** is the fraction of tuples that qualify.

- Formula depends on type of predicate:
  - Equality
  - Range
  - Negation
  - Conjunction
  - Disjunction

COMPLEX PREDICATES

- The **selectivity** (**sel**) of a predicate **P** is the fraction of tuples that qualify.

- Formula depends on type of predicate:
  - Equality
  - Range
  - Negation
  - Conjunction
  - Disjunction

SELECTIONS – COMPLEX PREDICATES

```
SELECT * FROM people
 WHERE age = 2
```

SELECTIONS – COMPLEX PREDICATES

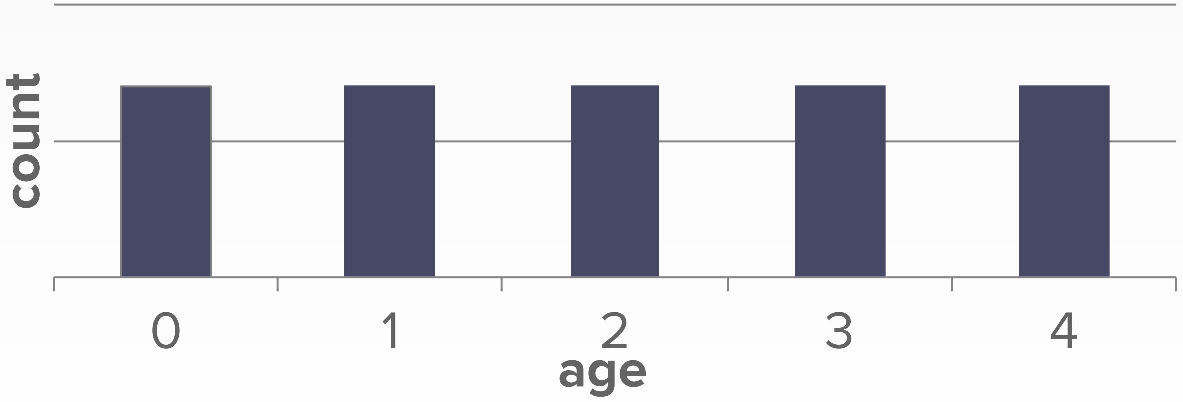
- Assume that **V(age,people)** has five distinct values (0–4) and N_R = 5
- Equality Predicate: **A=constant**
  - **sel(A=constant) = SC(P) / N_R**
  - Example: **sel(age=2) =**

```
SELECT * FROM people
 WHERE age = 2
```

SELECTIONS – COMPLEX PREDICATES

- Assume that **V(age,people)** has five distinct values (0–4) and $\mathbf{N_R} = 5$
- Equality Predicate: **A=constant**
  - **sel(A=constant) = SC(P) / $\mathbf{N_R}$**
  - Example: **sel(age=2) =**

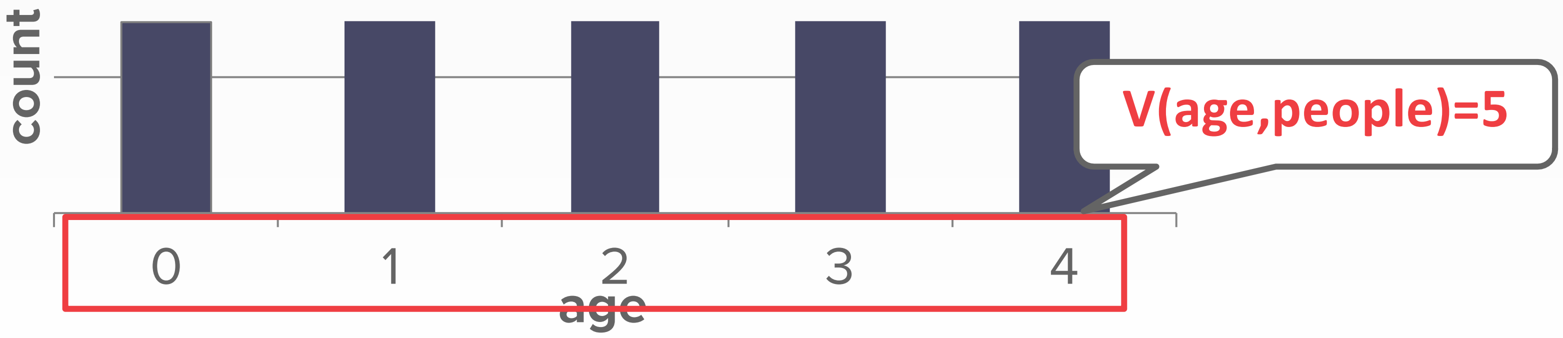
```
SELECT * FROM people
 WHERE age = 2
```



SELECTIONS – COMPLEX PREDICATES

- Assume that **V(age,people)** has five distinct values (0–4) and $N_R = 5$
- Equality Predicate: **A=constant**
  - **sel(A=constant) = SC(P) / N_R**
  - Example: **sel(age=2) =**

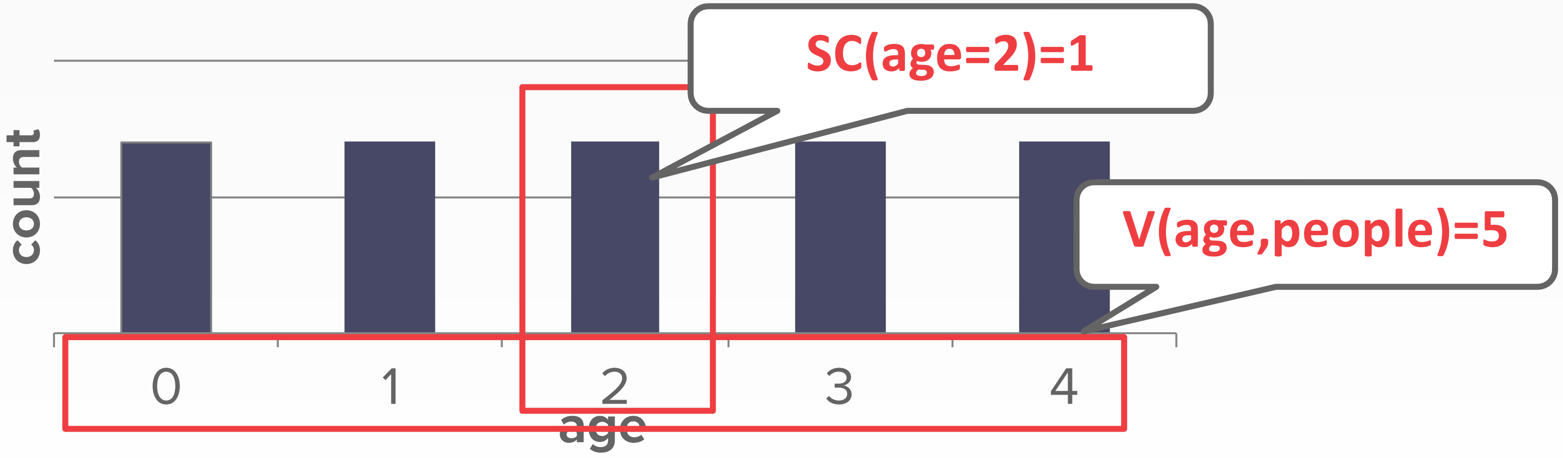
```
SELECT * FROM people
 WHERE age = 2
```



SELECTIONS – COMPLEX PREDICATES

- Assume that **V(age,people)** has five distinct values (0–4) and $\mathbf{N_R}$ = 5
- Equality Predicate: **A=constant**
  - **sel(A=constant) = SC(P) / $\mathbf{N_R}$**
  - Example: **sel(age=2) =**

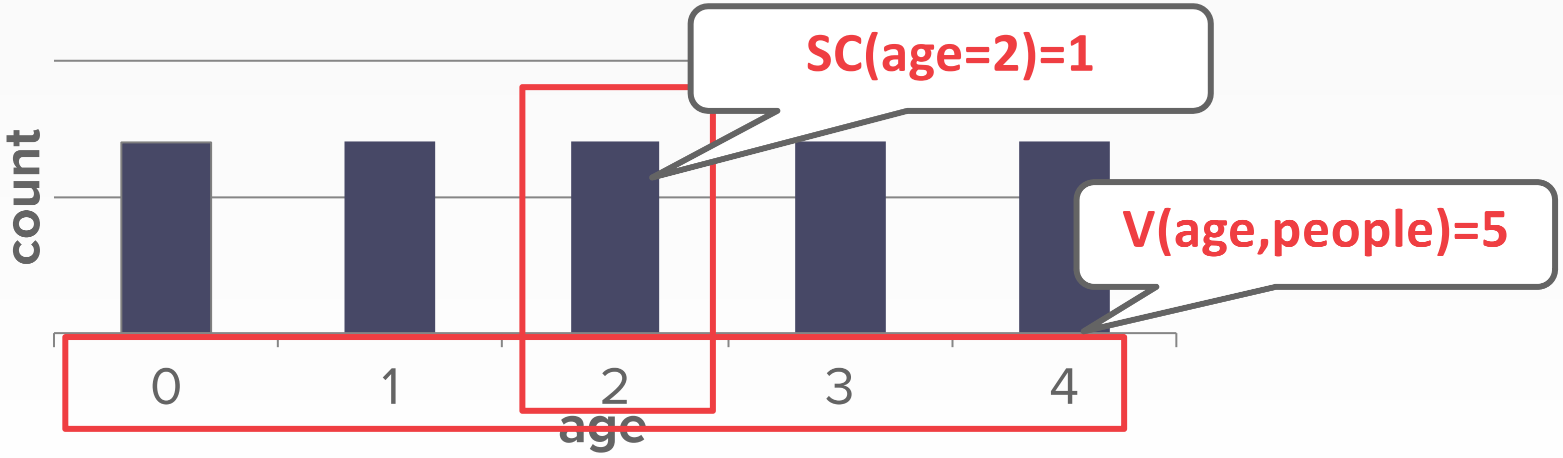
```
SELECT * FROM people
 WHERE age = 2
```



SELECTIONS – COMPLEX PREDICATES

- Assume that **V(age,people)** has five distinct values (0–4) and $\mathbf{N_R}$ = 5
- Equality Predicate: **A=constant**
  – **sel(A=constant) = SC(P) / $\mathbf{N_R}$**
  – Example: **sel(age=2) = 1/5**

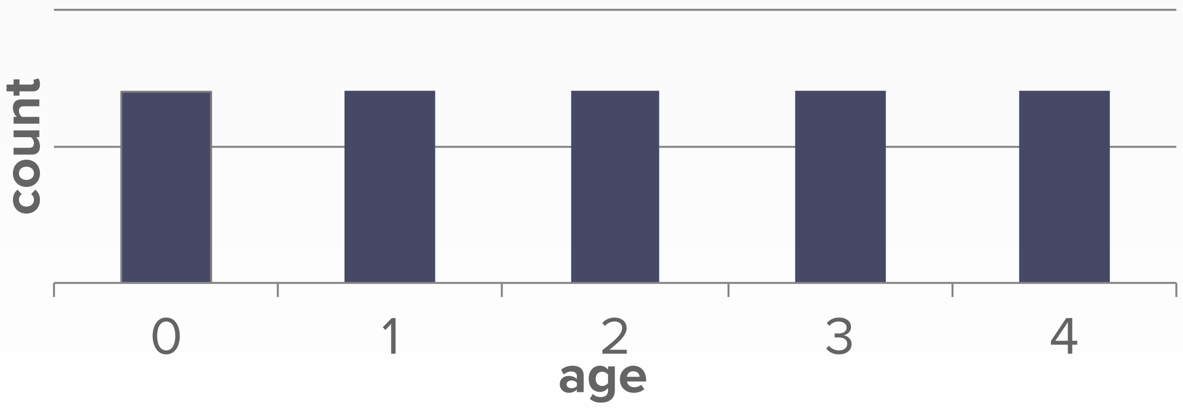
```
SELECT * FROM people
 WHERE age = 2
```



SELECTIONS – COMPLEX PREDICATES

- **Range Query:**
  - $sel(A>=a) = (A_{max}-a) / (A_{max}-A_{min})$
  - Example: **sel(age >= 2)**

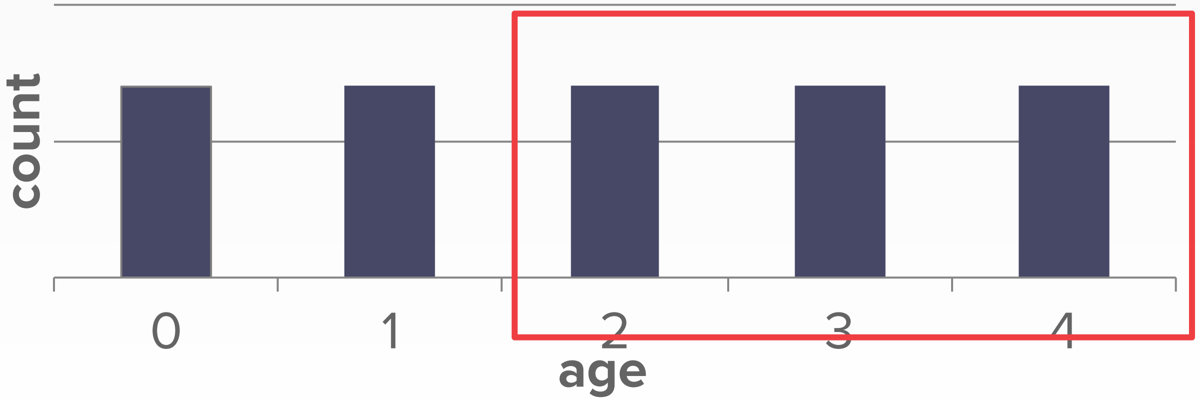
```
SELECT * FROM people
 WHERE age >= 2
```



SELECTIONS – COMPLEX PREDICATES

- **Range Query:**
  - $sel(A>=a) = (A_{max} - a) / (A_{max} - A_{min})$
  - Example: **sel(age >= 2)**

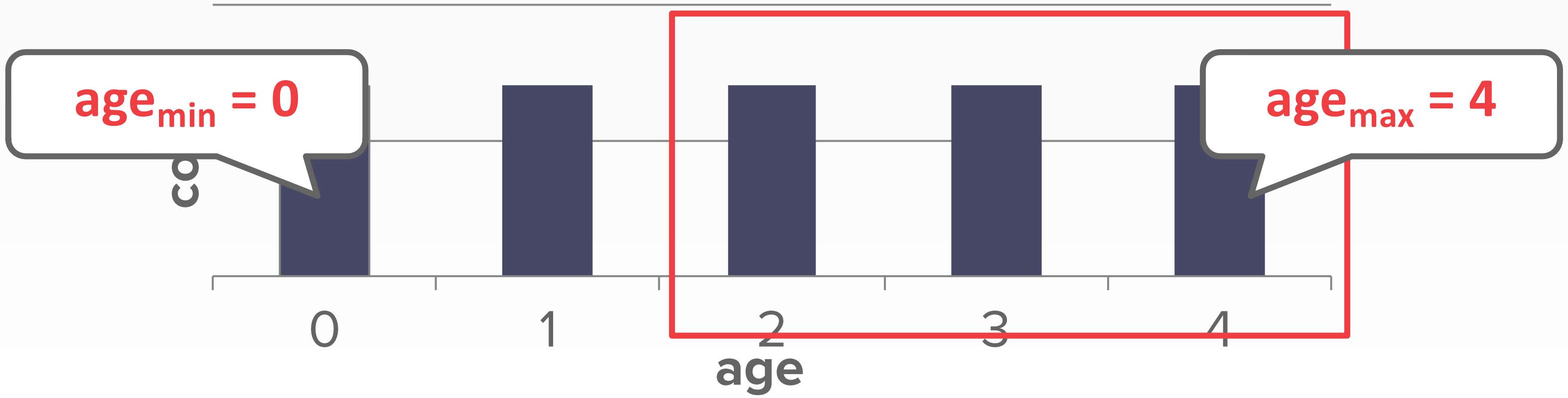
```
SELECT * FROM people
 WHERE age >= 2
```



SELECTIONS – COMPLEX PREDICATES

- **Range Query:**
  - $sel(A>=a) = (A_{max}-a) / (A_{max}-A_{min})$
  - Example: $sel(age >= 2)$

```
SELECT * FROM people
 WHERE age >= 2
```



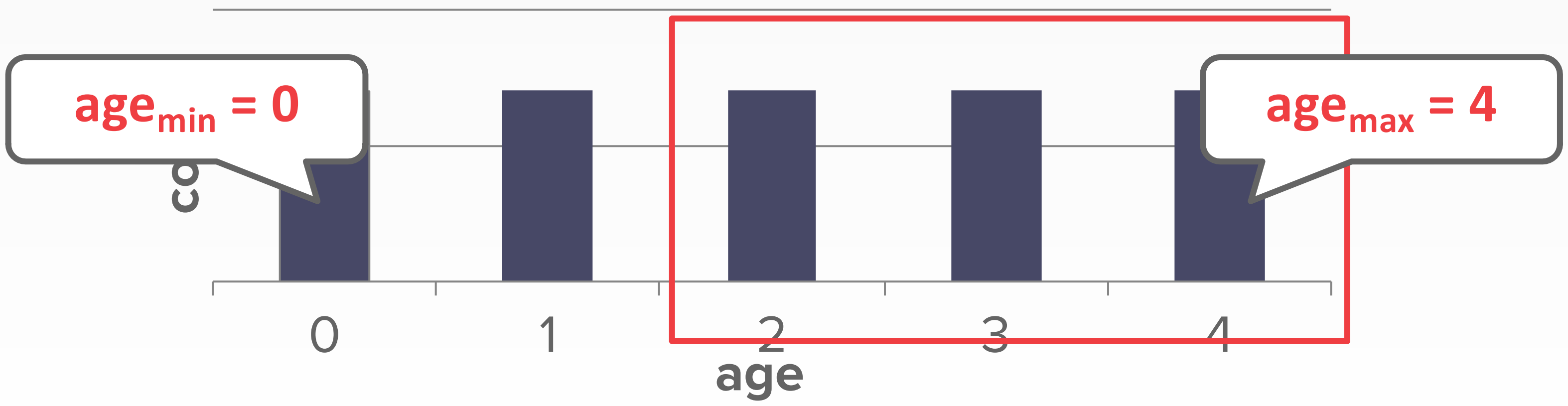
SELECTIONS – COMPLEX PREDICATES

- **Range Query:**
  - $sel(A>=a) = (A_{max}-a) / (A_{max}-A_{min})$
  - Example: $sel(age >= 2)$

$= (4-2) / (4-0)$

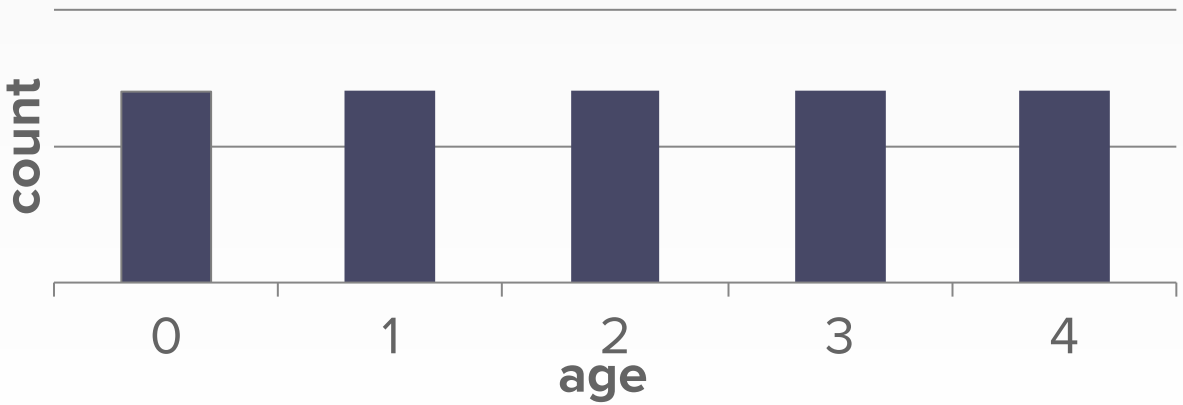
$= 1/2$

```
SELECT * FROM people
 WHERE age >= 2
```



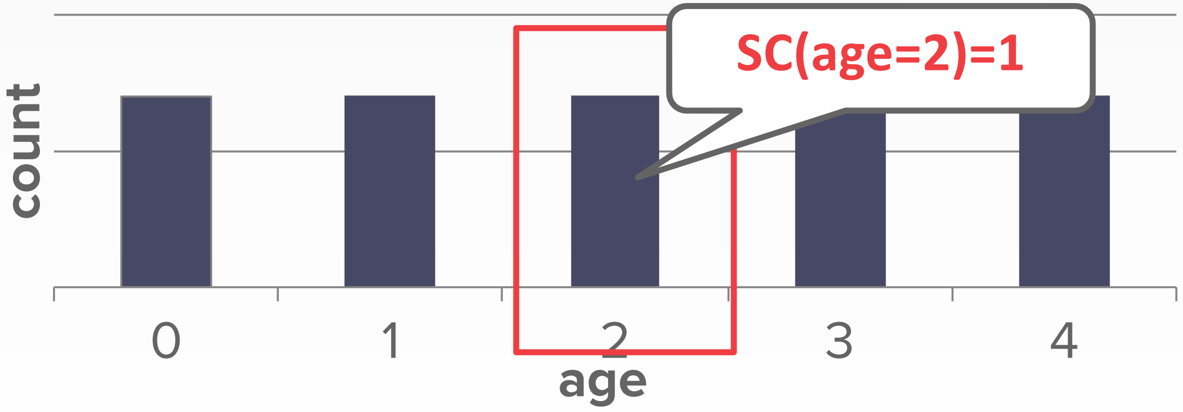
SELECTIONS – COMPLEX PREDICATES

```
SELECT * FROM people
 WHERE age != 2
```



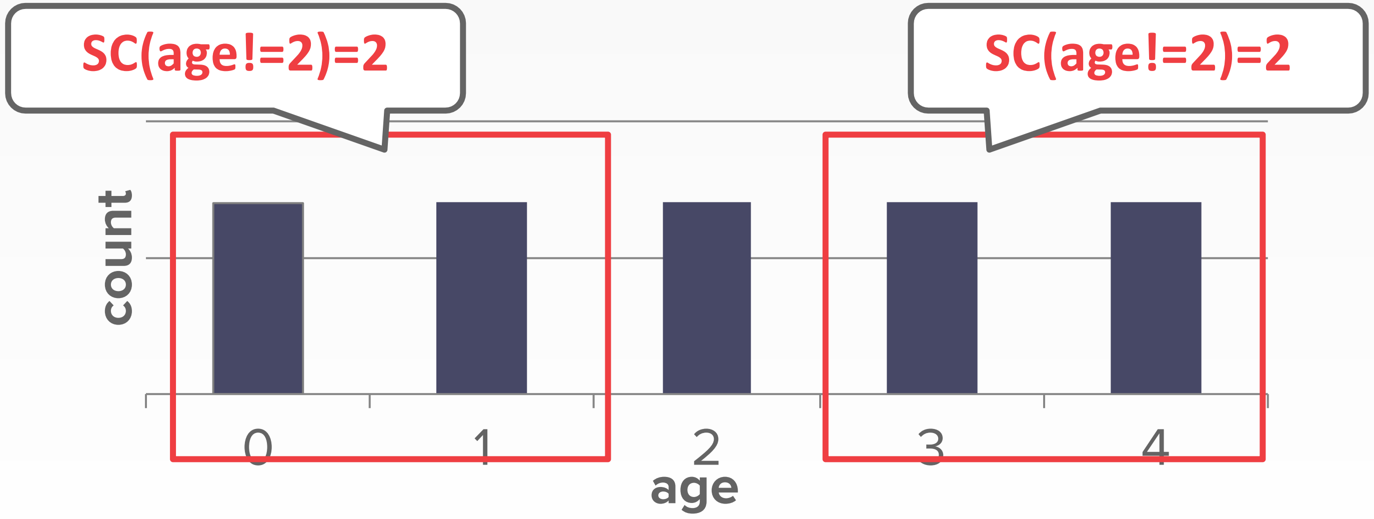
SELECTIONS – COMPLEX PREDICATES

```
SELECT * FROM people
 WHERE age != 2
```



SELECTIONS – COMPLEX PREDICATES

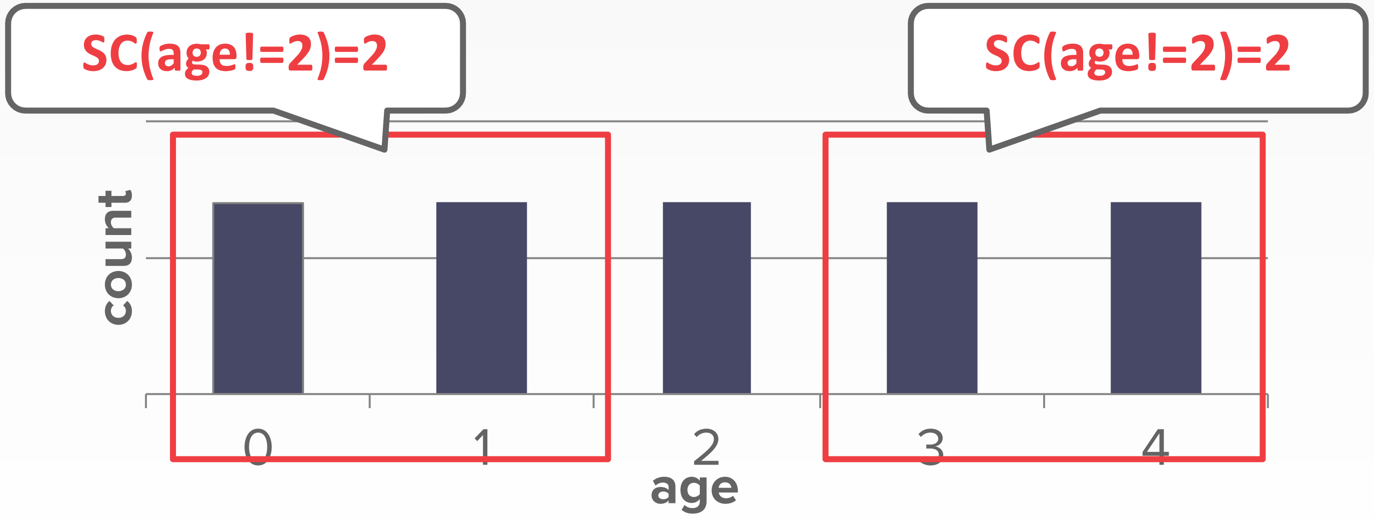
```
SELECT * FROM people
 WHERE age != 2
```



SELECTIONS – COMPLEX PREDICATES

```
SELECT * FROM people
 WHERE age != 2
```

= 1 – (1/5) = 4/5

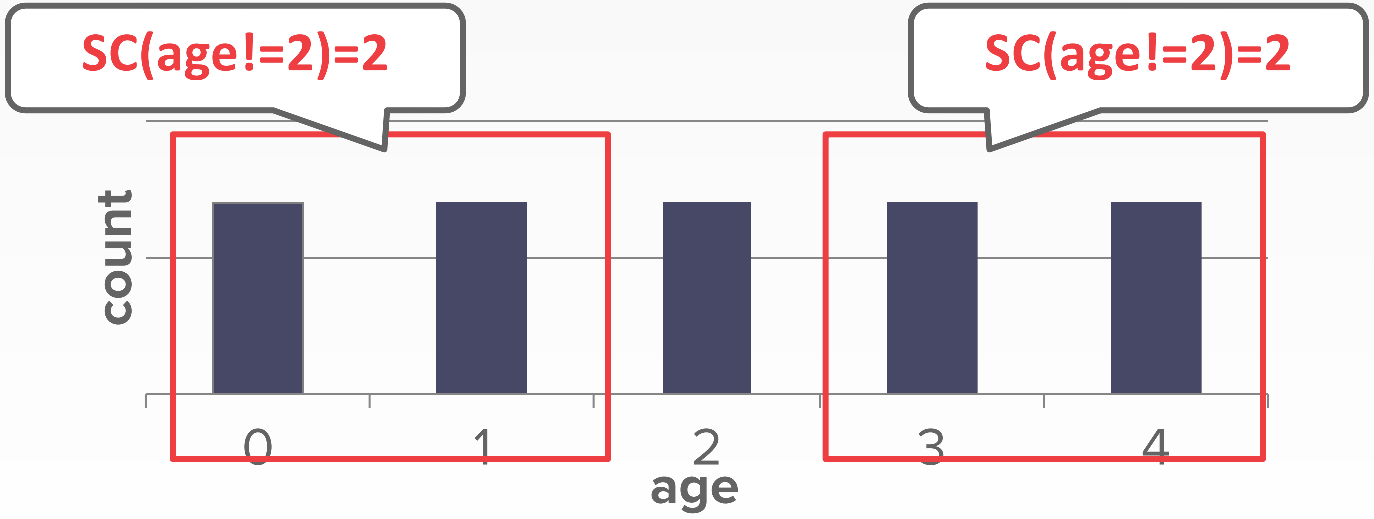


SELECTIONS – COMPLEX PREDICATES

- **Negation Query:**
  - **sel(not P) = 1 – sel(P)**
  - Example: **sel(age != 2) = 1 – (1/5) = 4/5**

```
SELECT * FROM people
 WHERE age != 2
```

- ***Observation: Selectivity ≈ Probability***

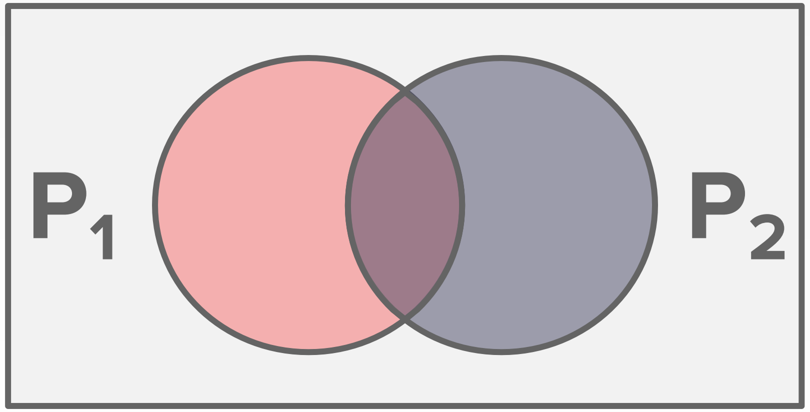


SELECTIONS – COMPLEX PREDICATES

- **Conjunction:**
  - sel(P1 ∧ P2) = sel(P1) · sel(P2)
  - sel(age=2 ∧ name LIKE 'A%')

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```

- This assumes that the predicates are independent.

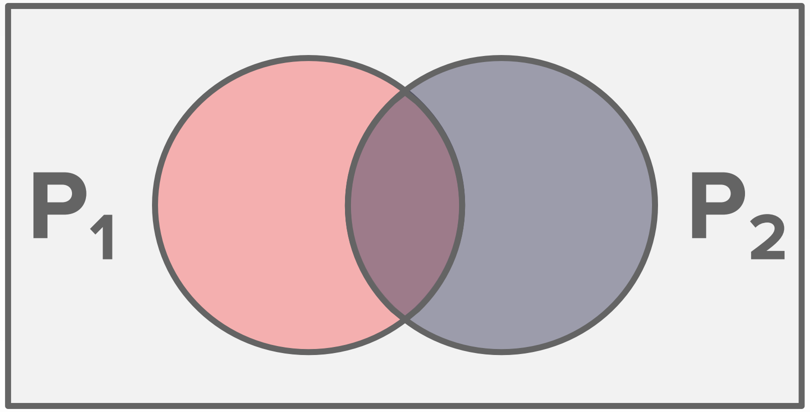


SELECTIONS – COMPLEX PREDICATES

- **Conjunction:**
  - sel(P1 ∧ P2) = sel(P1) · sel(P2)
  - sel(age=2 ∧ name LIKE 'A%')

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```

- This assumes that the predicates are independent.

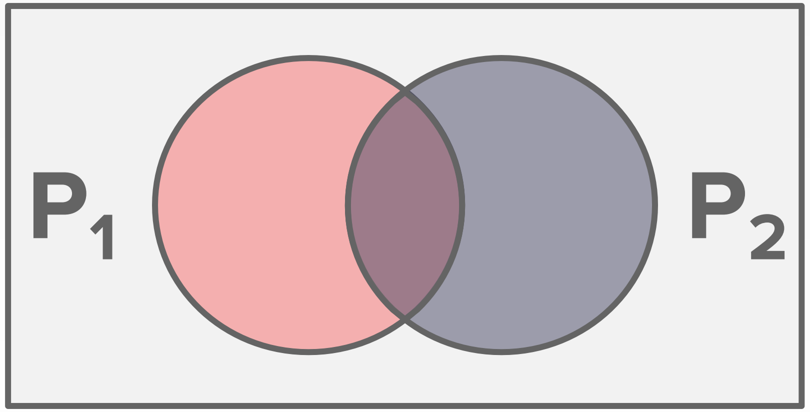


SELECTIONS – COMPLEX PREDICATES

- **Conjunction:**
  - **sel(P1 ∧ P2) = sel(P1) · sel(P2)**
  - **sel(age=2 ∧ name LIKE 'A%')**

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```

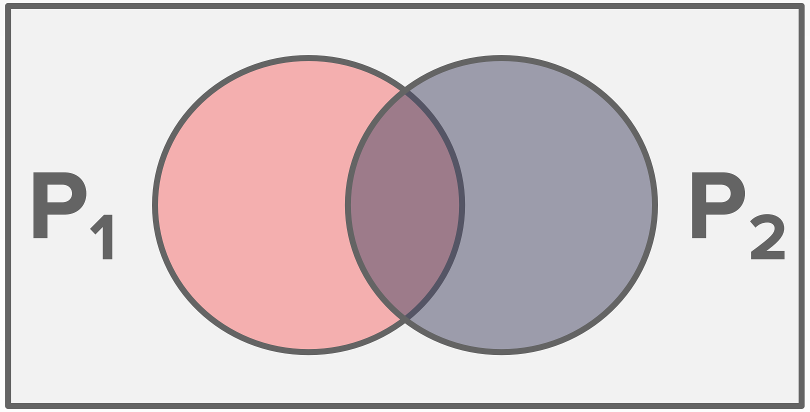
- This assumes that the predicates are independent.



SELECTIONS – COMPLEX PREDICATES

- **Disjunction:**
  - sel(P1 ⋁ P2)
    = sel(P1)+sel(P2)–sel(P1⋁P2)
    = sel(P1)+sel(P2)–sel(P1)·sel(P2)
  - sel(age=2 OR name LIKE 'A%')

```
SELECT * FROM people
 WHERE age = 2
   OR name LIKE 'A%'
```



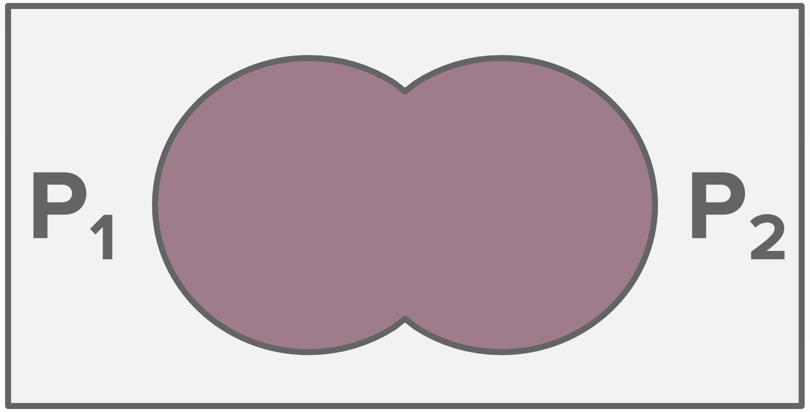
- This again assumes that the selectivities are independent.

SELECTIONS – COMPLEX PREDICATES

- **Disjunction:**
  - sel(P1 ⋁ P2)
    = sel(P1)+sel(P2)–sel(P1⋁P2)
    = sel(P1)+sel(P2)–sel(P1)·sel(P2)
  - sel(age=2 OR name LIKE 'A%')

- This again assumes that the selectivities are independent.

```
SELECT * FROM people
 WHERE age = 2
   OR name LIKE 'A%'
```



RESULT SIZE ESTIMATION FOR JOINS

- Given a join of **R** and **S**, what is the range of possible result sizes in # of tuples?

- In other words, for a given tuple of **R**, how many tuples of **S** will it match?

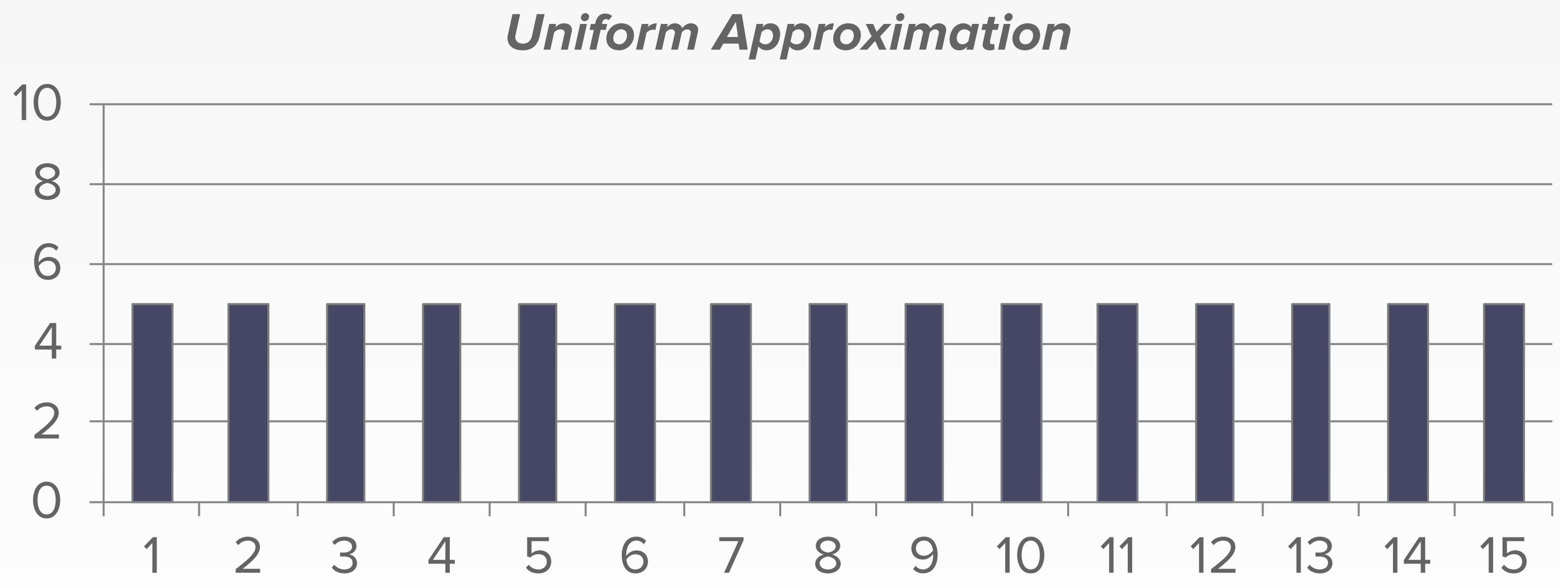
RESULT SIZE ESTIMATION FOR JOINS

- General case: $R_{cols} \cap S_{cols} = \{A\}$ where **A** is not a key for either table.
  - Match each **R**-tuple with **S**-tuples:
    $\textbf{estSize} \approx N_R \cdot N_S / V(A,S)$
  - Symmetrically, for **S**:
    $\textbf{estSize} \approx N_R \cdot N_S / V(A,R)$

- Overall:
  - $\textbf{estSize} \approx N_R \cdot N_S / \max(\{V(A,S), V(A,R)\})$

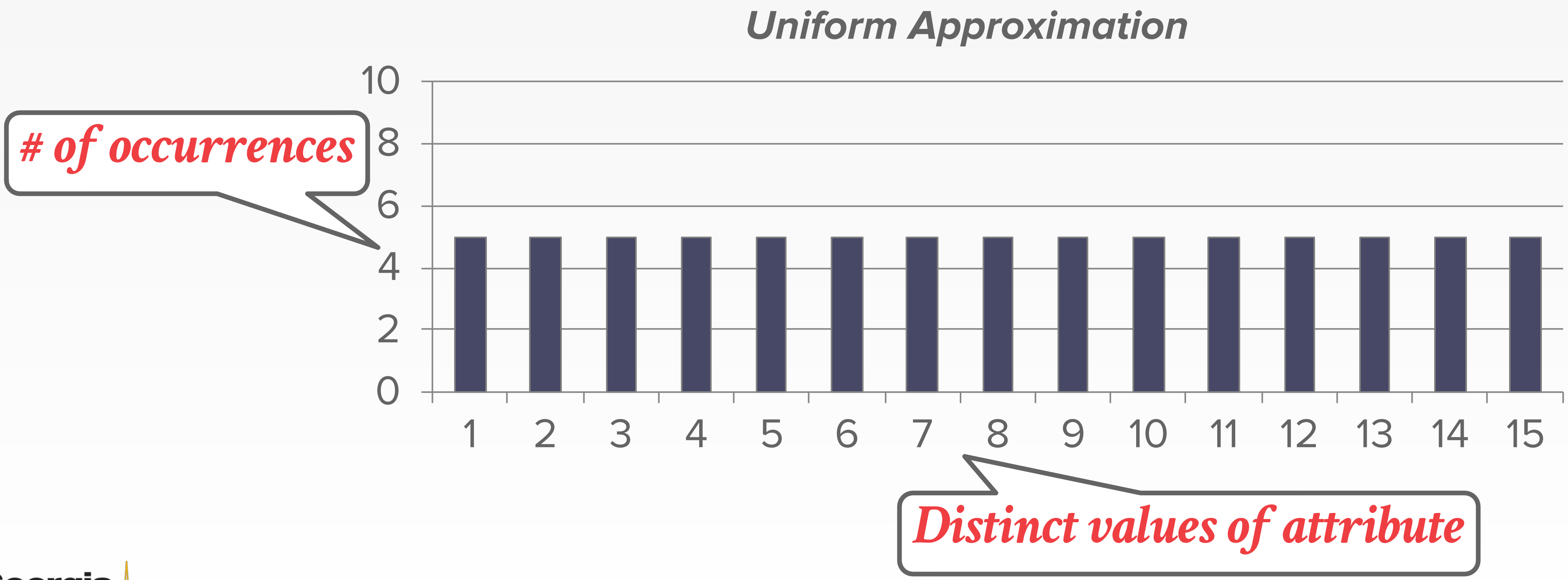
COST ESTIMATIONS

- Our formulas are nice but we assume that data values are uniformly distributed.



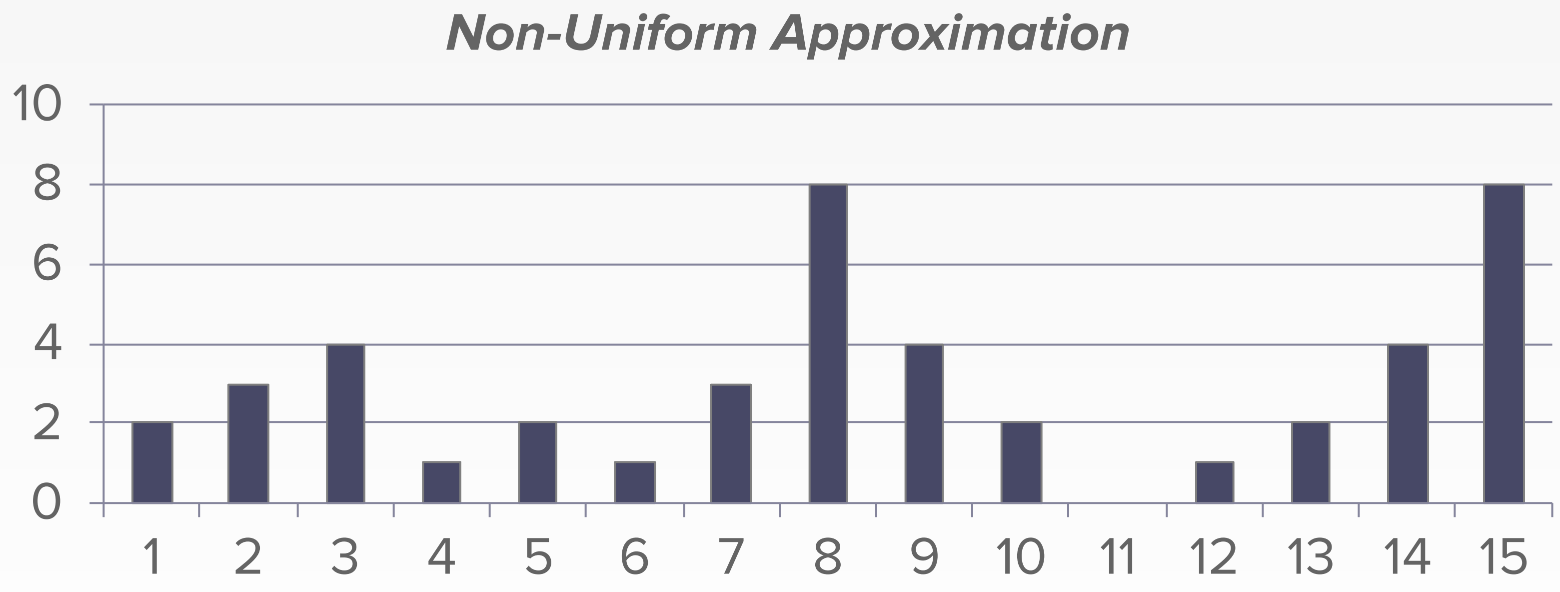
COST ESTIMATIONS

- Our formulas are nice but we assume that data values are uniformly distributed.



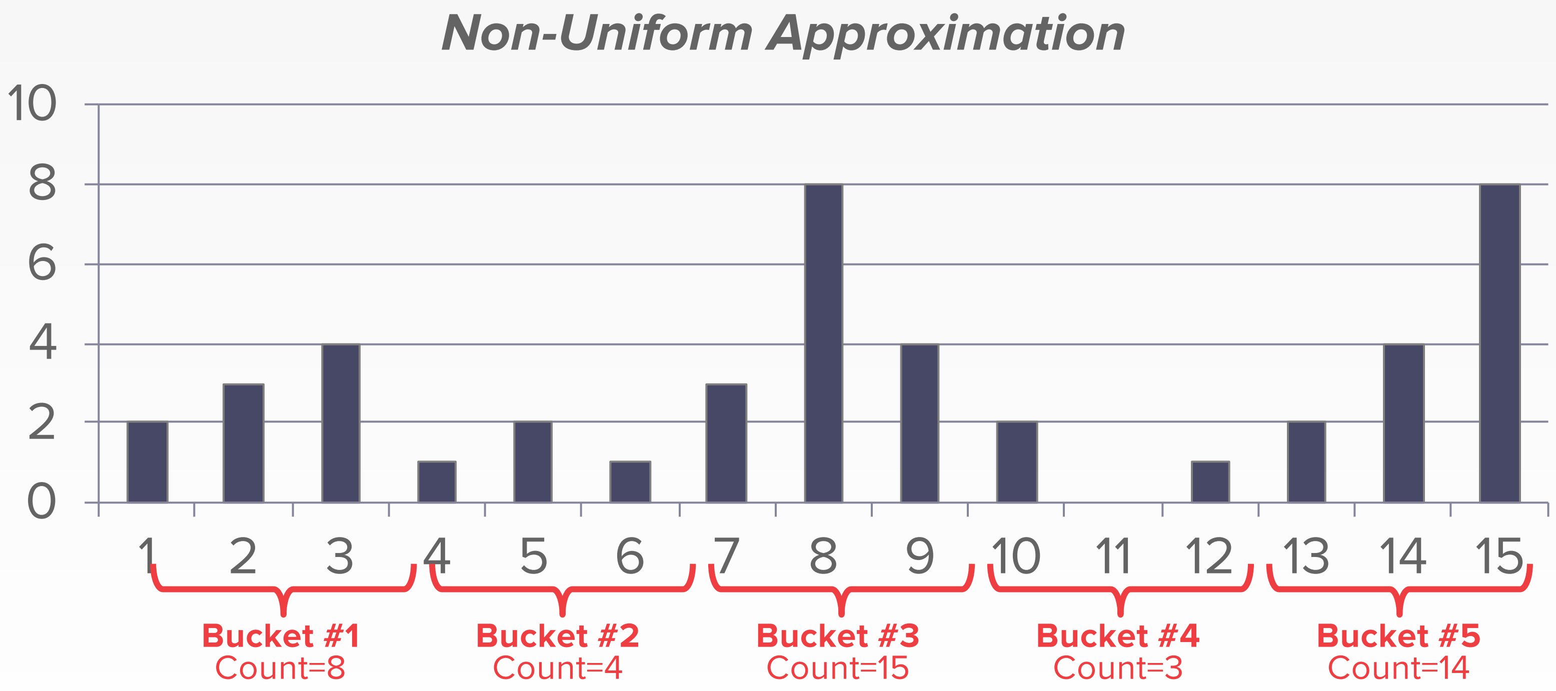
COST ESTIMATIONS

- Our formulas are nice but we assume that data values are uniformly distributed.



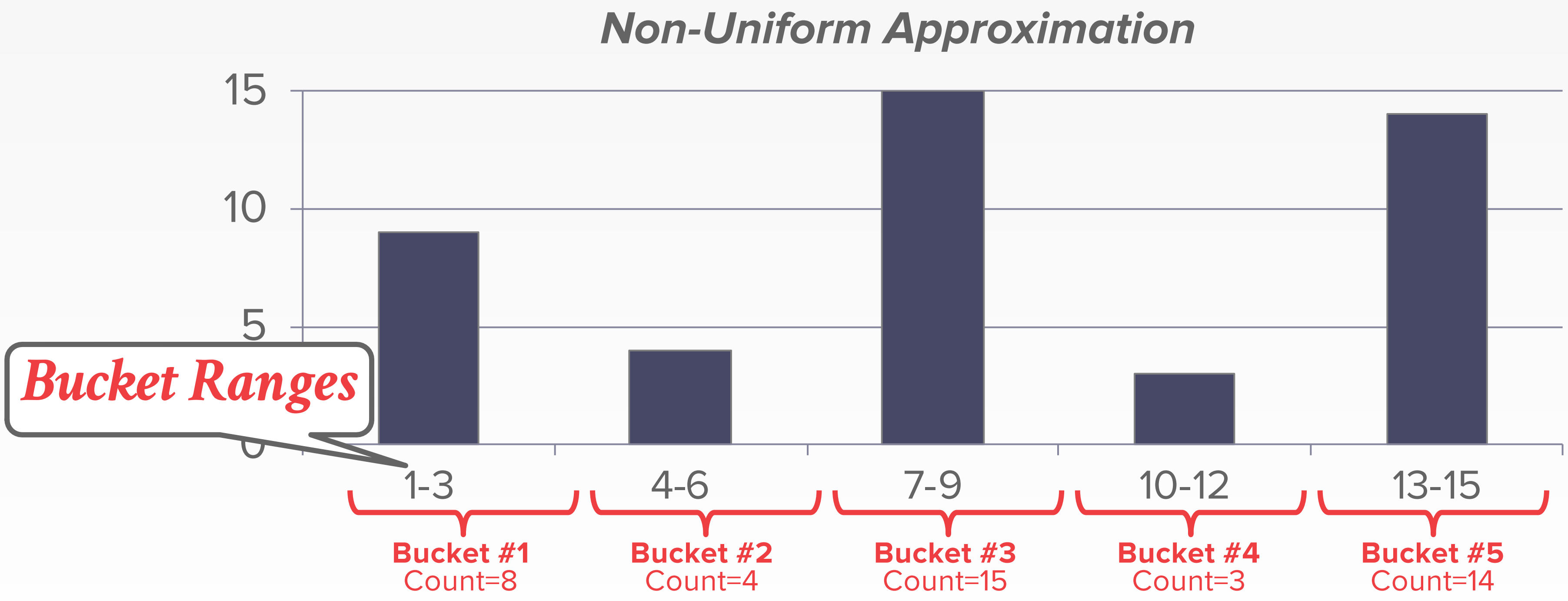
COST ESTIMATIONS

- Our formulas are nice but we assume that data values are uniformly distributed.



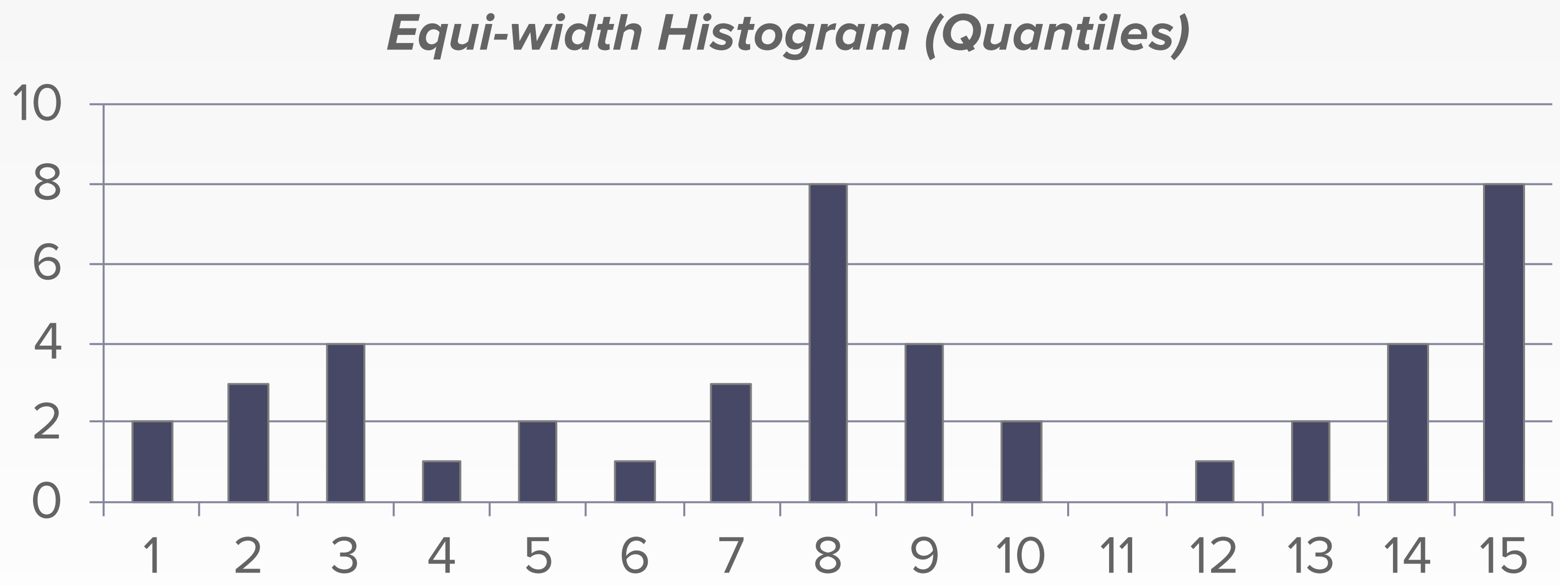
COST ESTIMATIONS

- Our formulas are nice but we assume that data values are uniformly distributed.



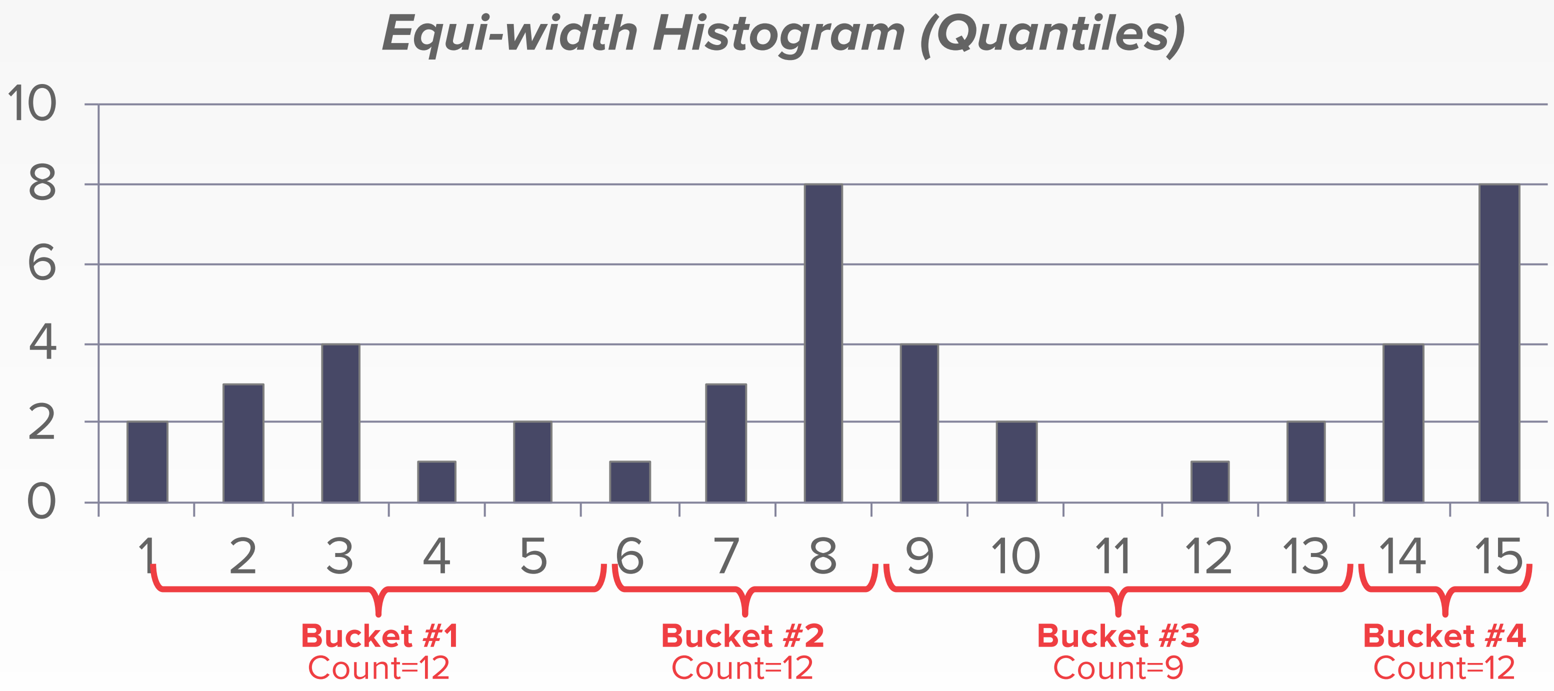
HISTOGRAMS WITH QUANTILES

- A histogram type wherein the "spread" of each bucket is same.



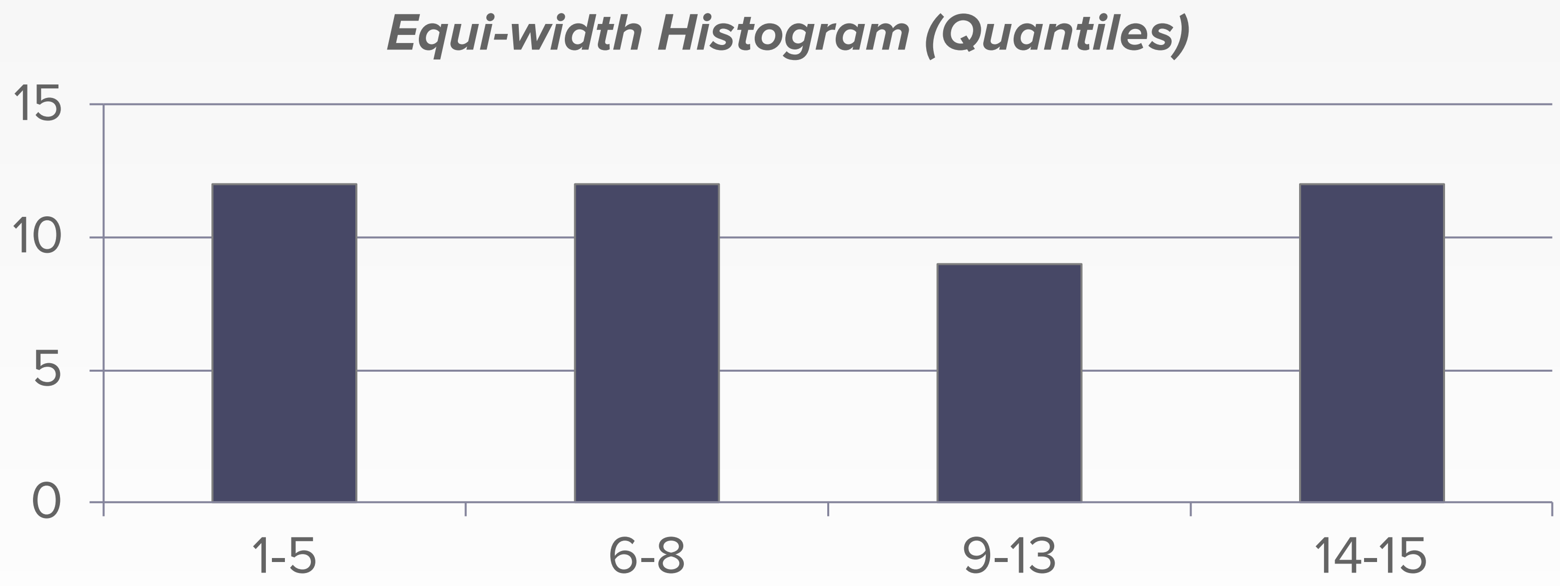
HISTOGRAMS WITH QUANTILES

- A histogram type wherein the "spread" of each bucket is same.



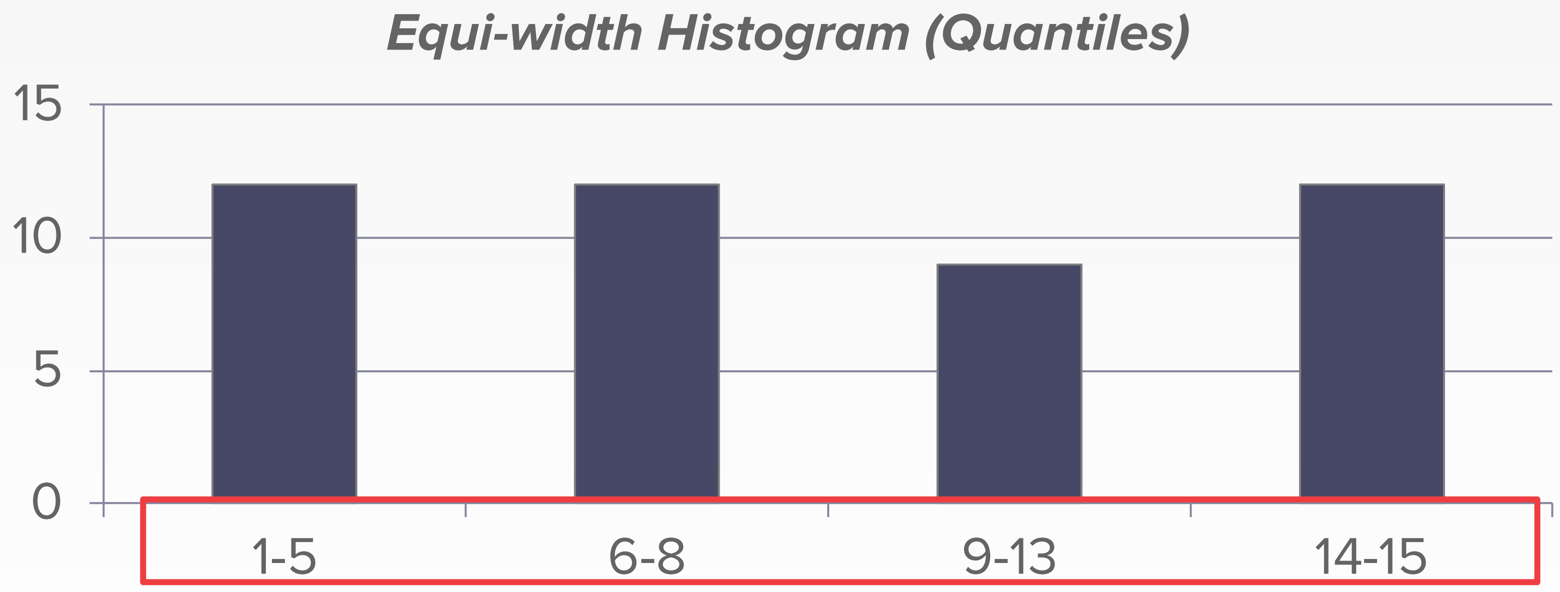
HISTOGRAMS WITH QUANTILES

- A histogram type wherein the "spread" of each bucket is same.



HISTOGRAMS WITH QUANTILES

- A histogram type wherein the "spread" of each bucket is same.



SAMPLING

- Modern DBMSs also collect sampl
tables to estimate selectivities.

- Update samples when the underl
changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	56	Rested
1002	Kanye	40	Weird
1003	Tupac	25	Dead
1004	Bieber	23	Crunk
1005	Andy	37	Lit

⋮

1 billion tuples

SAMPLING

- Modern DBMSs also collect sampl tables to estimate selectivities.
- Update samples when the underl changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	56	Rested
1002	Kanye	40	Weird
1003	Tupac	25	Dead
1004	Bieber	23	Crunk
1005	Andy	37	Lit

⋮

1 billion tuples

SAMPLING

- Modern DBMSs also collect sampl tables to estimate selectivities.
- Update samples when the underl changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	56	Rested
1002	Kanye	40	Weird
1003	Tupac	25	Dead
1004	Bieber	23	Crunk
1005	Andy	37	Lit

⋮

1 billion tuples

Table Sample

1001	Obama	56	Rested
1003	Tupac	25	Dead
1005	Andy	37	Lit

SAMPLING

- Modern DBMSs also collect sampl
  tables to estimate selectivities.

- Update samples when the underl
  changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	56	Rested
1002	Kanye	40	Weird
1003	Tupac	25	Dead
1004	Bieber	23	Crunk
1005	Andy	37	Lit

⋮

1 billion tuples

Table Sample

sel(age>50) =

1001	Obama	56	Rested
1003	Tupac	25	Dead
1005	Andy	37	Lit

SAMPLING

- Modern DBMSs also collect sampl tables to estimate selectivities.
- Update samples when the underl changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	56	Rested
1002	Kanye	40	Weird
1003	Tupac	25	Dead
1004	Bieber	23	Crunk
1005	Andy	37	Lit

⋮

1 billion tuples

Table Sample

sel(age>50) =

1001	Obama	56	Rested
1003	Tupac	25	Dead
1005	Andy	37	Lit

SAMPLING

- Modern DBMSs also collect sampl
tables to estimate selectivities.

- Update samples when the underl
changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	56	Rested
1002	Kanye	40	Weird
1003	Tupac	25	Dead
1004	Bieber	23	Crunk
1005	Andy	37	Lit

⋮

1 billion tuples

Table Sample

1001	Obama	56	Rested
1003	Tupac	25	Dead
1005	Andy	37	Lit

sel(age>50) = 1/3

OBSERVATION

- Now that we can (roughly) estimate the selectivity of predicates, what can we actually do with them?

PLAN ENUMERATION

QUERY OPTIMIZATION

- After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs.
  - Single table.
  - Multiple tables.
- It chooses the best plan it has seen for the query after exhausting all plans or some timeout.

SINGLE-TABLE QUERY PLANNING

- Pick the best access method.
  - Sequential Scan
  - Binary Search (clustered indexes)
  - Index Scan
- Simple heuristics are often good enough for this.
- OLTP queries are especially easy.

OLTP QUERY PLANNING

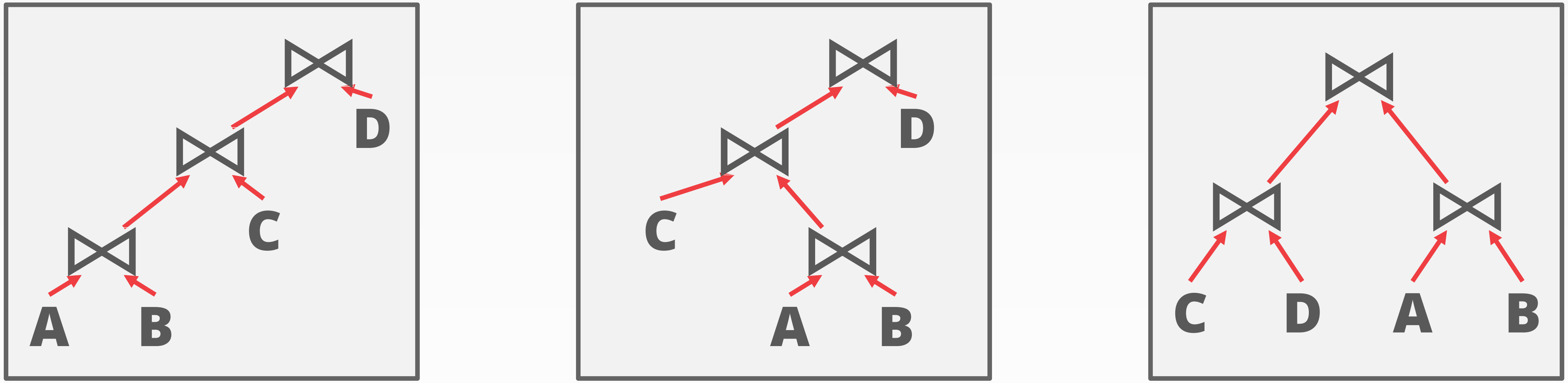
- Query planning for OLTP queries is easy because they are **sargable**.
  - **S**earch **Arg**ument **Able**
  - It is usually just picking the best index.
  - Joins are almost always on foreign key relationships with a small cardinality.
  - Can be implemented with simple heuristics.

MULTI-TABLE QUERY PLANNING

- As number of joins increases, number of alternative plans grows rapidly
  - We need to restrict search space.
- Fundamental decision in **System R**: only left-deep join trees are considered.
  - Modern DBMSs do not always make this assumption anymore.

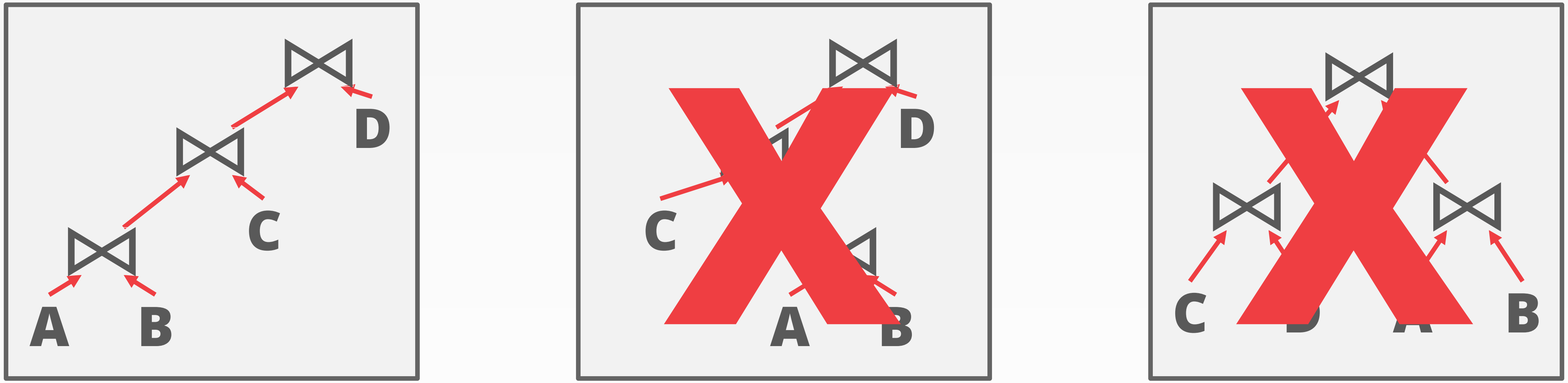
MULTI-TABLE QUERY PLANNING

- Fundamental decision in **System R**: Only consider left-deep join trees.



MULTI-TABLE QUERY PLANNING

- Fundamental decision in **System R**: Only consider left-deep join trees.



MULTI-TABLE QUERY PLANNING

- Fundamental decision in **System R**: Only consider left-deep join trees.

- Allows for fully pipelined plans where intermediate results are not written to temp files.
  - Not all left-deep trees are fully pipelined.

MULTI-TABLE QUERY PLANNING

MULTI-TABLE QUERY PLANNING

- Enumerate the orderings
  - Example: Left-deep tree #1, Left-deep tree #2…
- Enumerate the plans for each operator
  - Example: Hash, Sort-Merge, Nested Loop…
- Enumerate the access paths for each table
  - Example: Index #1, Index #2, Seq Scan…
- Use **dynamic programming** to reduce the number of cost estimations.

DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```

R
S
T

$R \bowtie S$
T

$T \bowtie S$
R

⋮

$R \bowtie S \bowtie T$

DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```

R
S
T

Hash Join
R.a=S.a

SortMerge Join
R.a=S.a

R ⋈ S
T

SortMerge Join
T.b=S.b

Hash Join
T.b=S.b

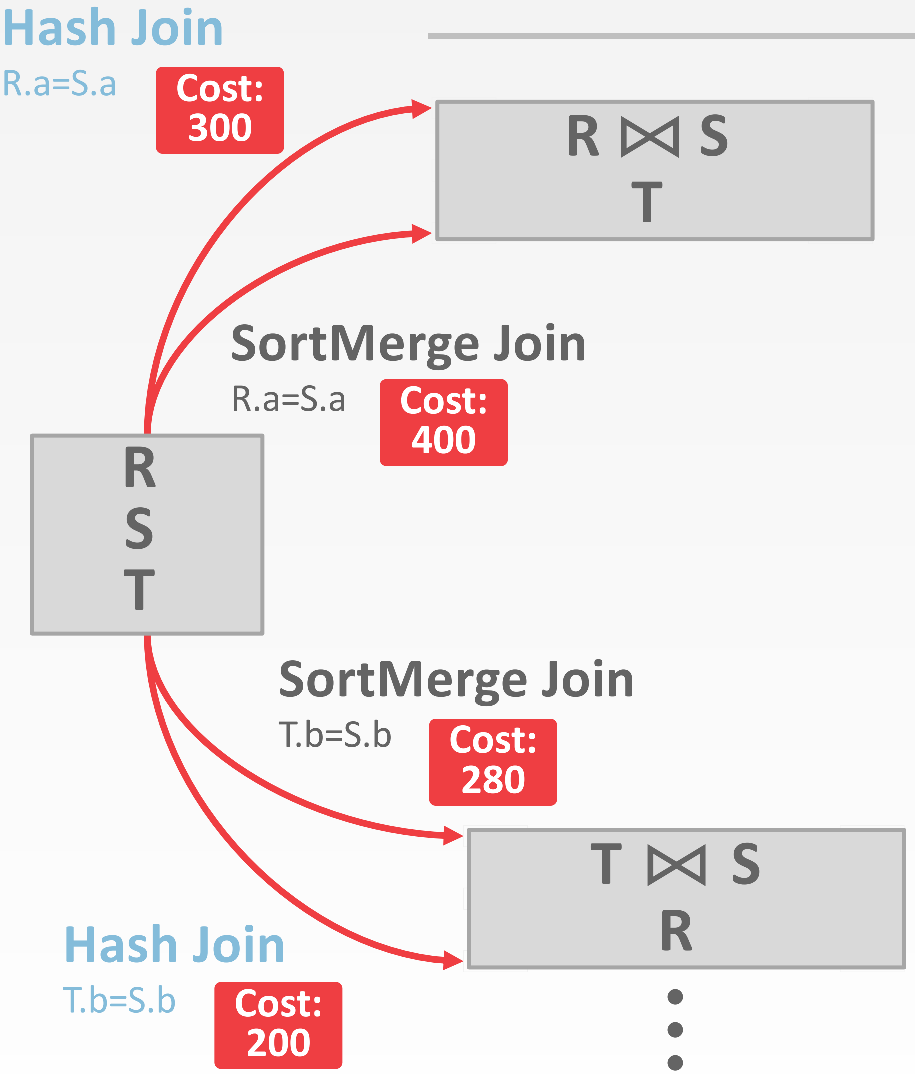
T ⋈ S
R

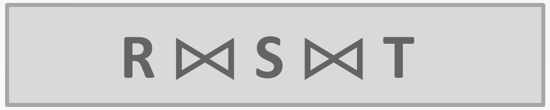
⋮

R ⋈ S ⋈ T

DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```





DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```

Hash Join

R.a=S.a

Cost: 300

R ⋈ S
T

R
S
T

T ⋈ S
R

⋮

Hash Join

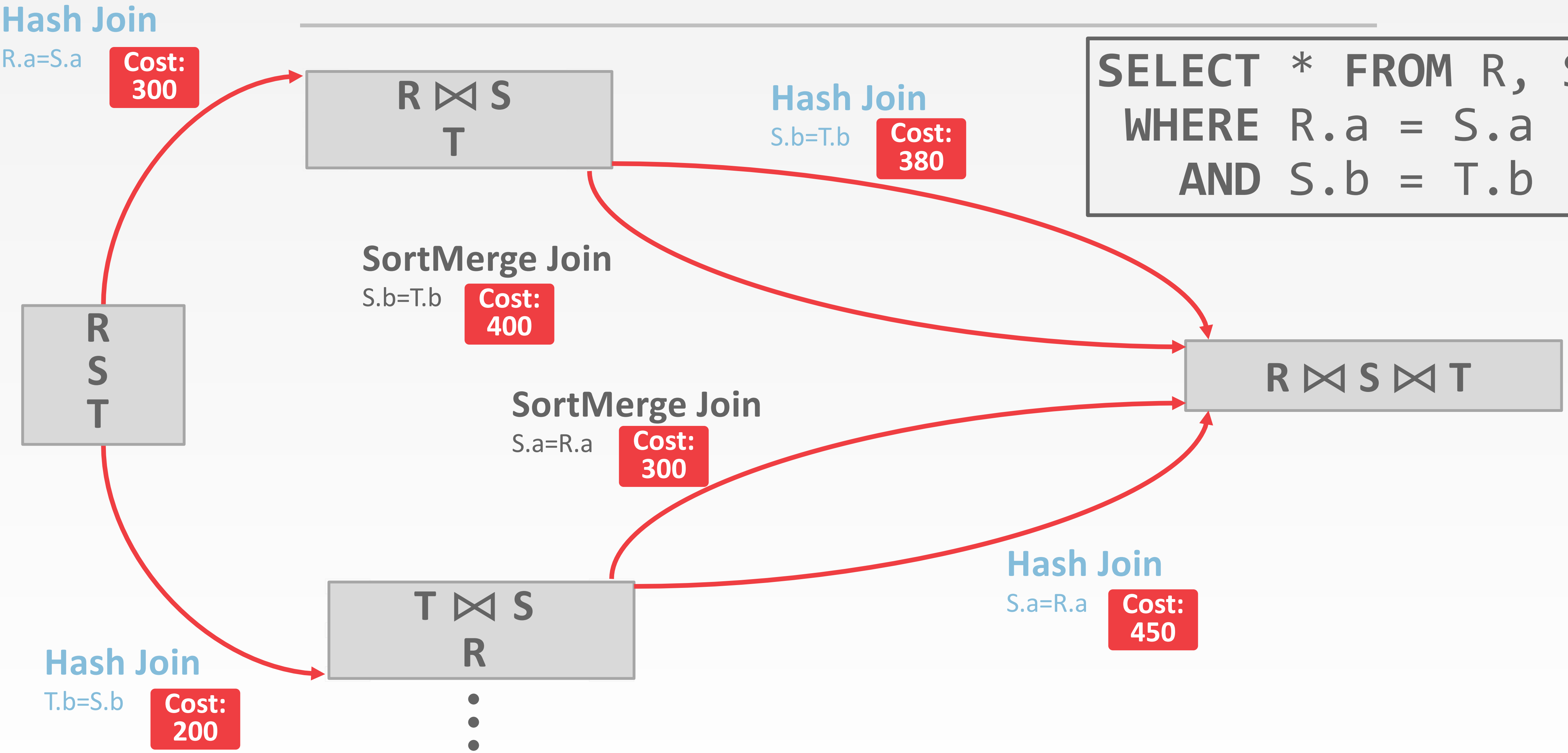
T.b=S.b

Cost: 200

R ⋈ S ⋈ T

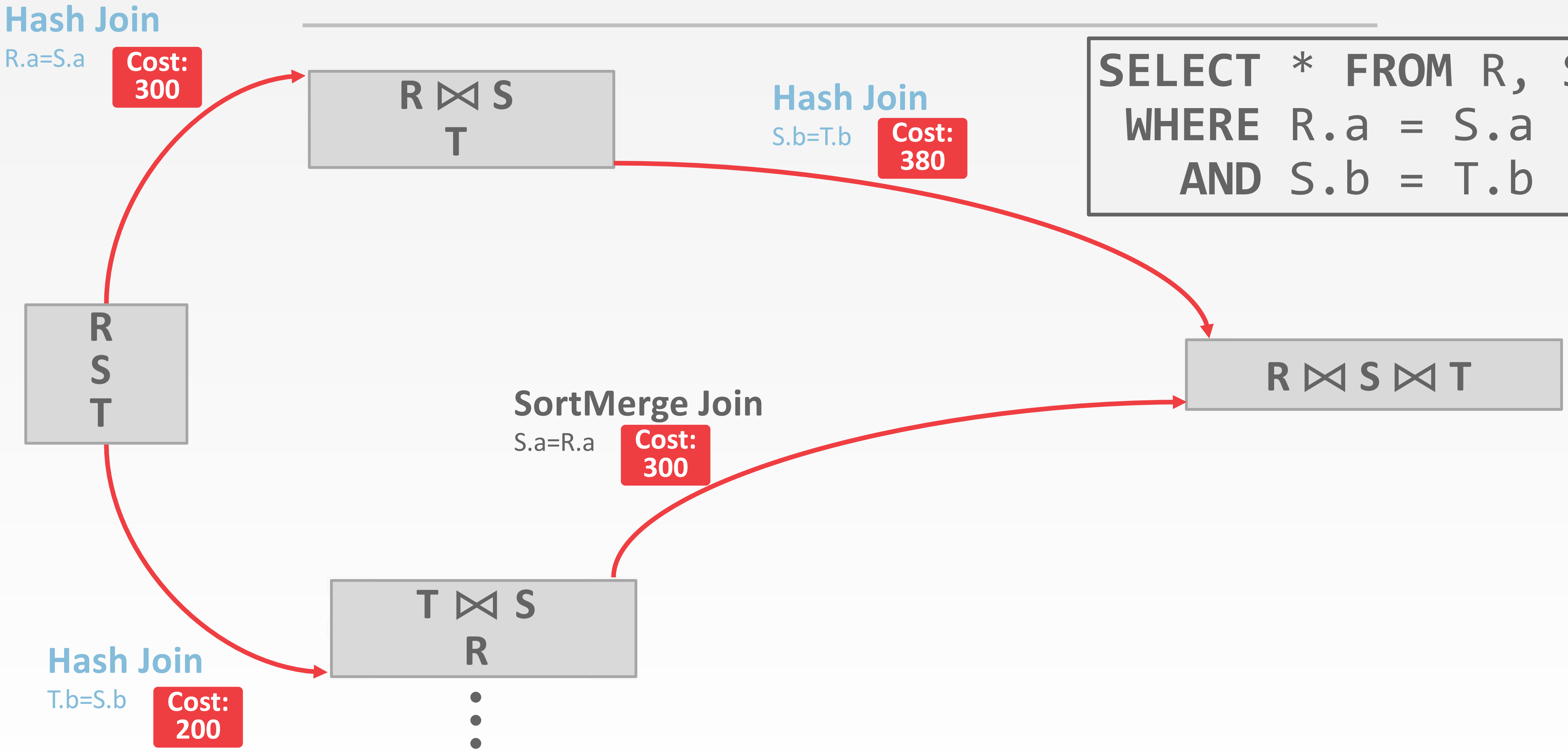
DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



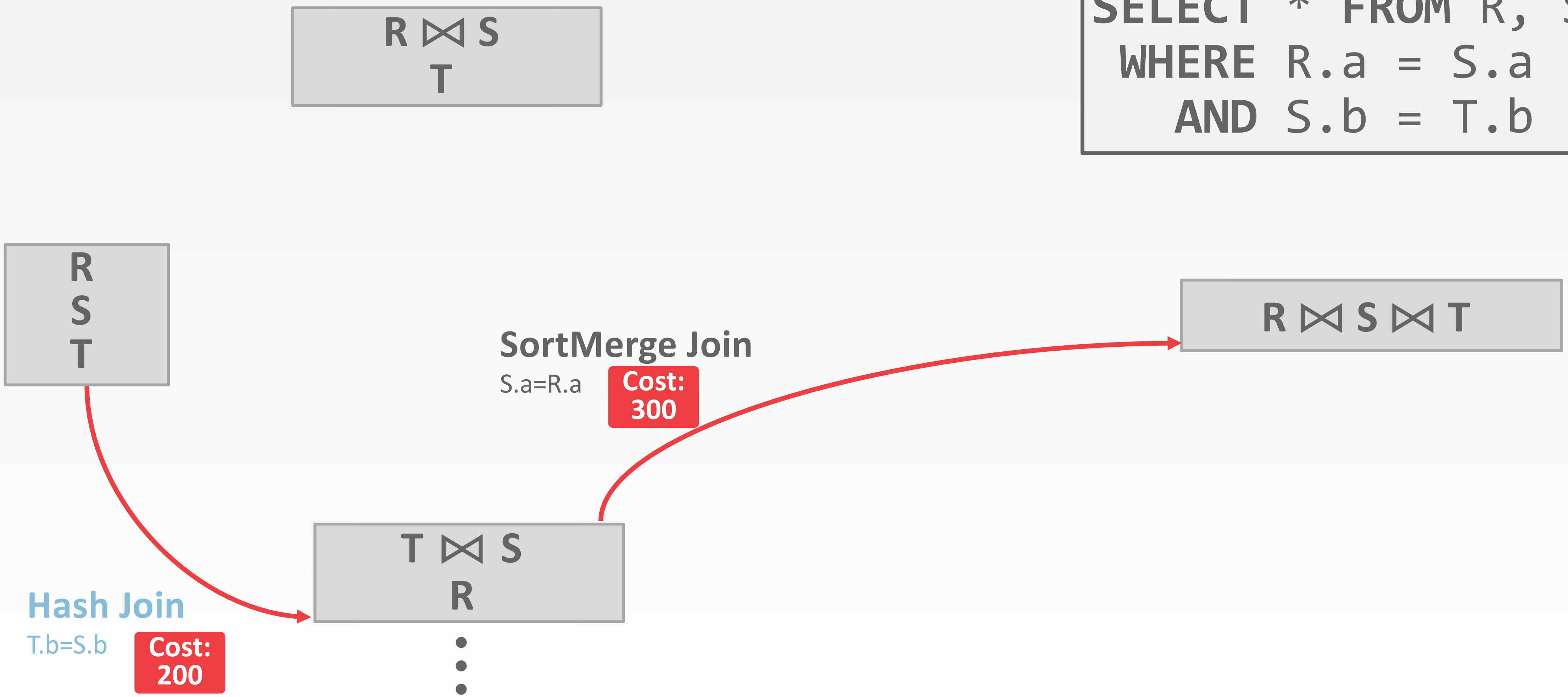
DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



CANDIDATE PLAN EXAMPLE

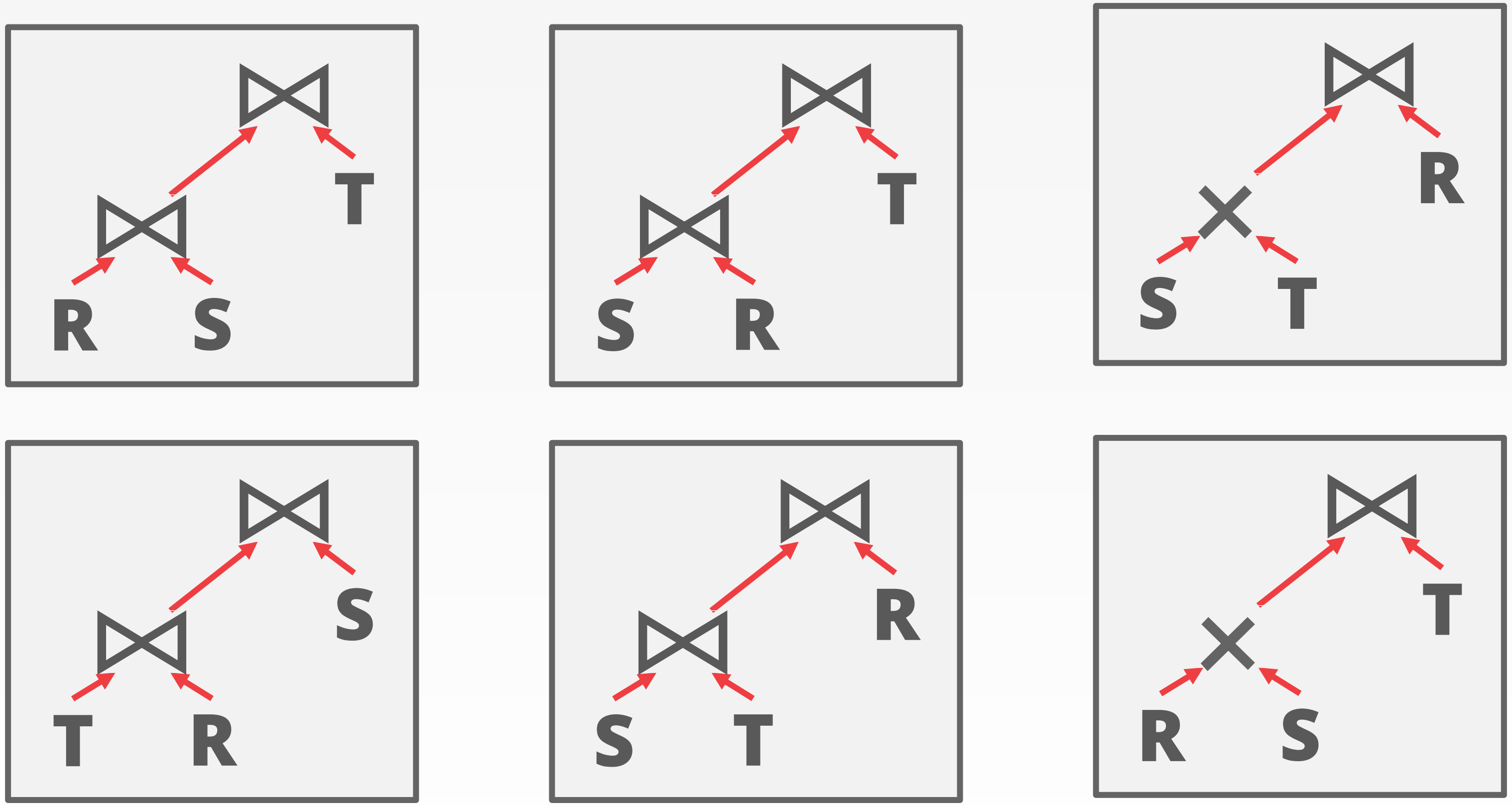
```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```

- How to generate plans for search algorithm:
  - Enumerate relation orderings
  - Enumerate join algorithm choices
  - Enumerate access method choices

- No real DBMSs does it this way. It's actually more messy…

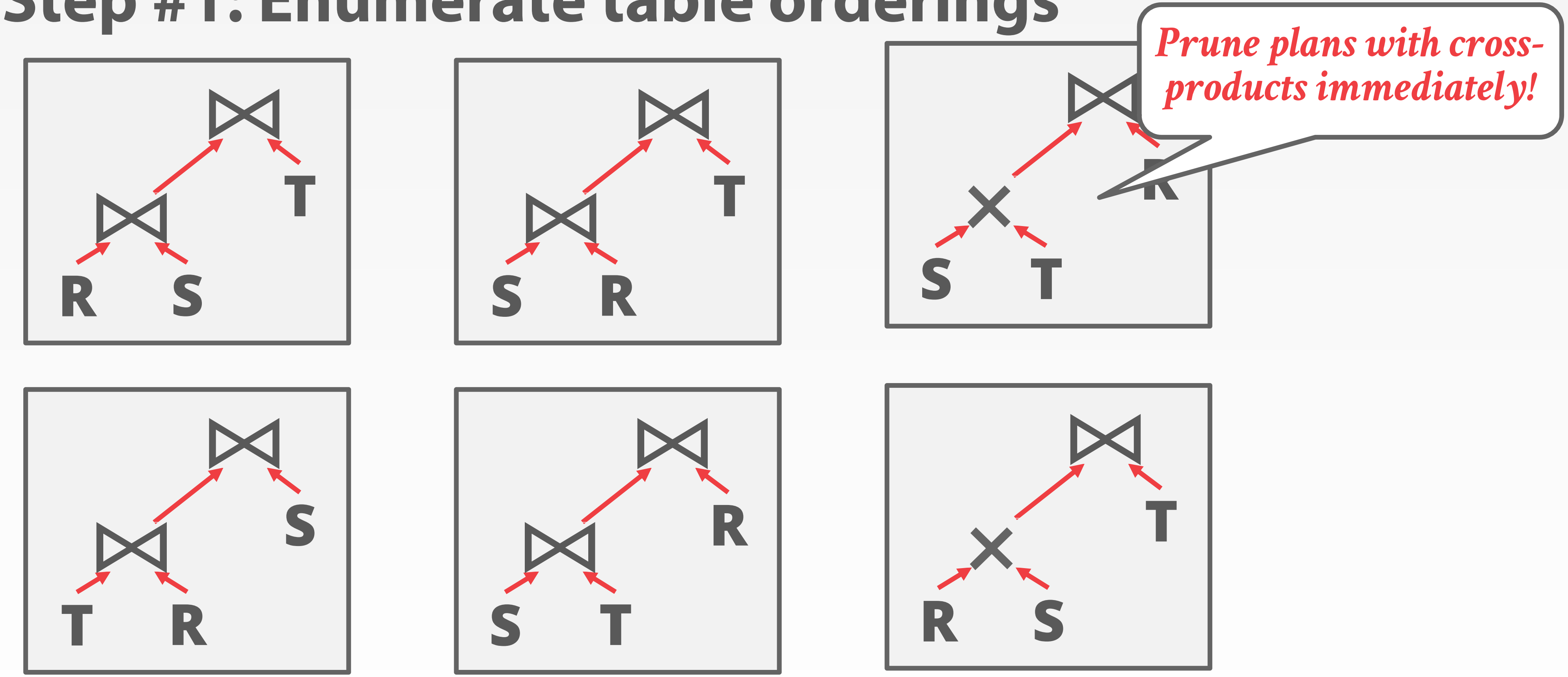
CANDIDATE PLANS

- **Step #1: Enumerate table orderings**



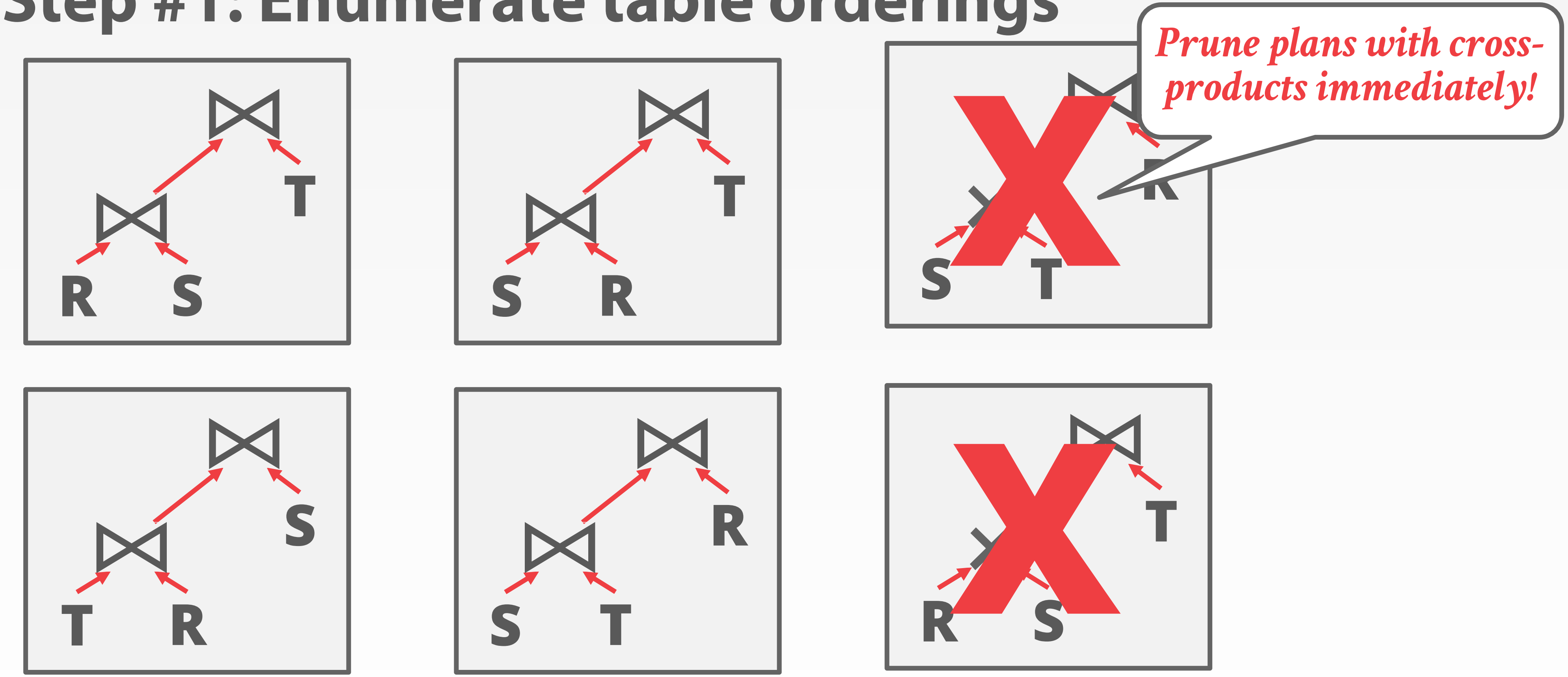
CANDIDATE PLANS

- **Step #1: Enumerate table orderings**



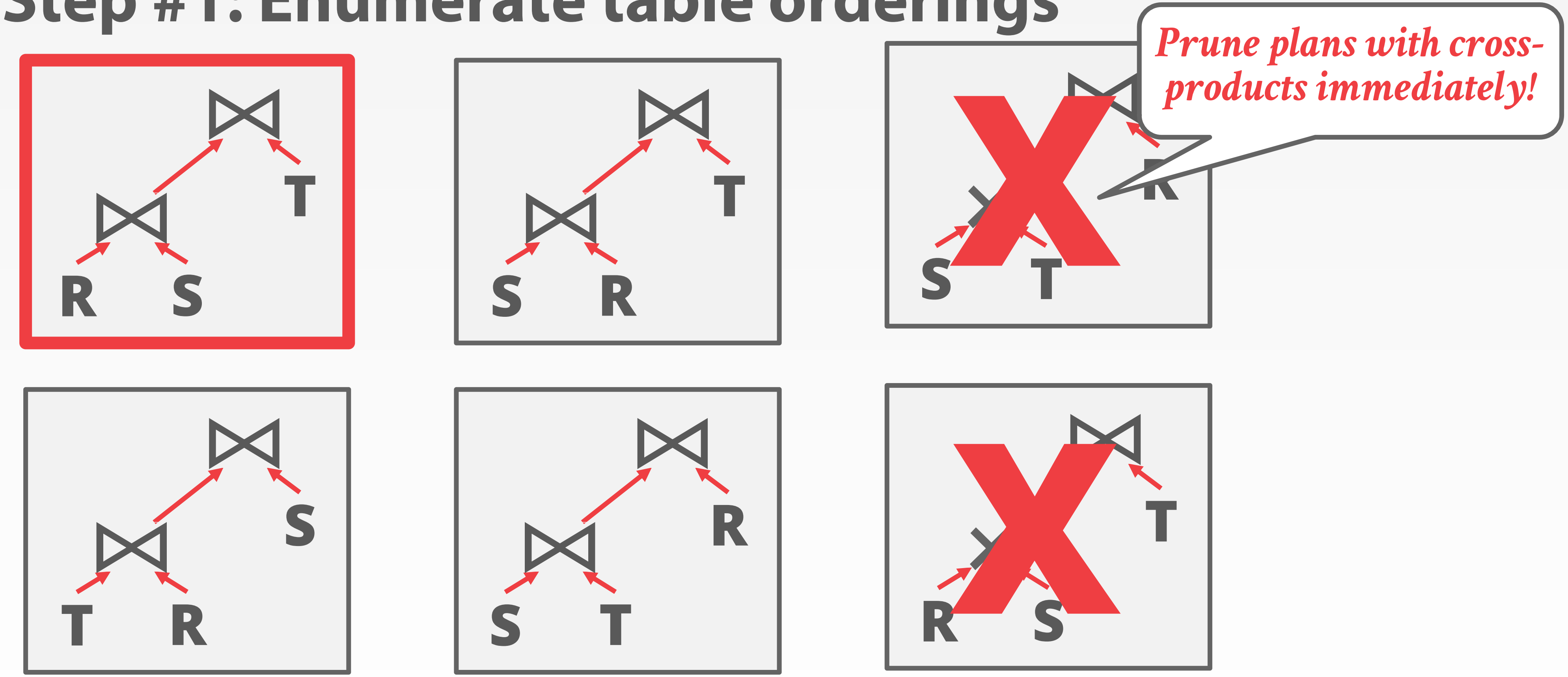
CANDIDATE PLANS

- **Step #1: Enumerate table orderings**



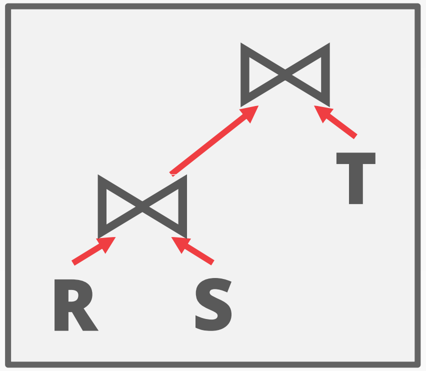
CANDIDATE PLANS

- **Step #1: Enumerate table orderings**



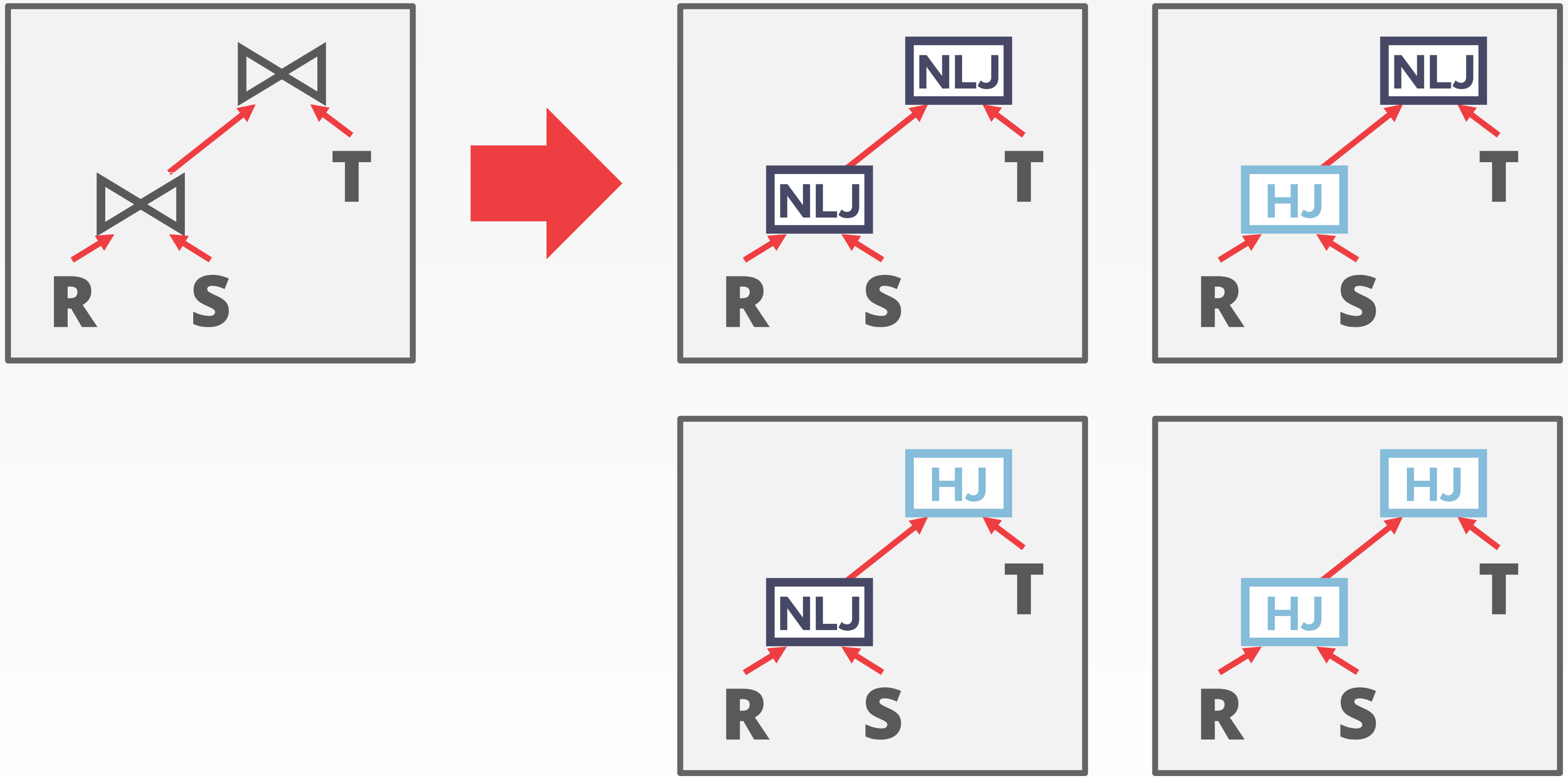
CANDIDATE PLANS

- **Step #2: Enumerate join algorithm choices**



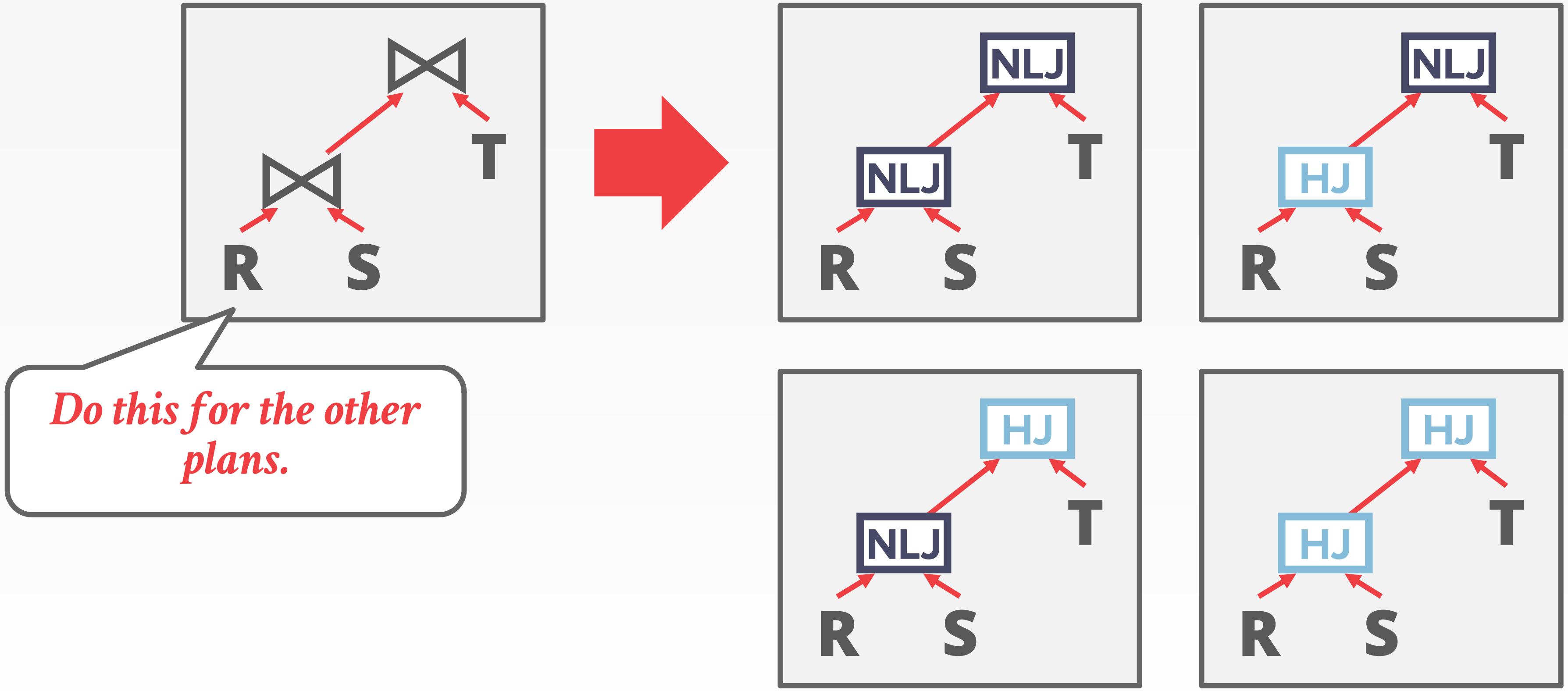
CANDIDATE PLANS

- **Step #2: Enumerate join algorithm choices**



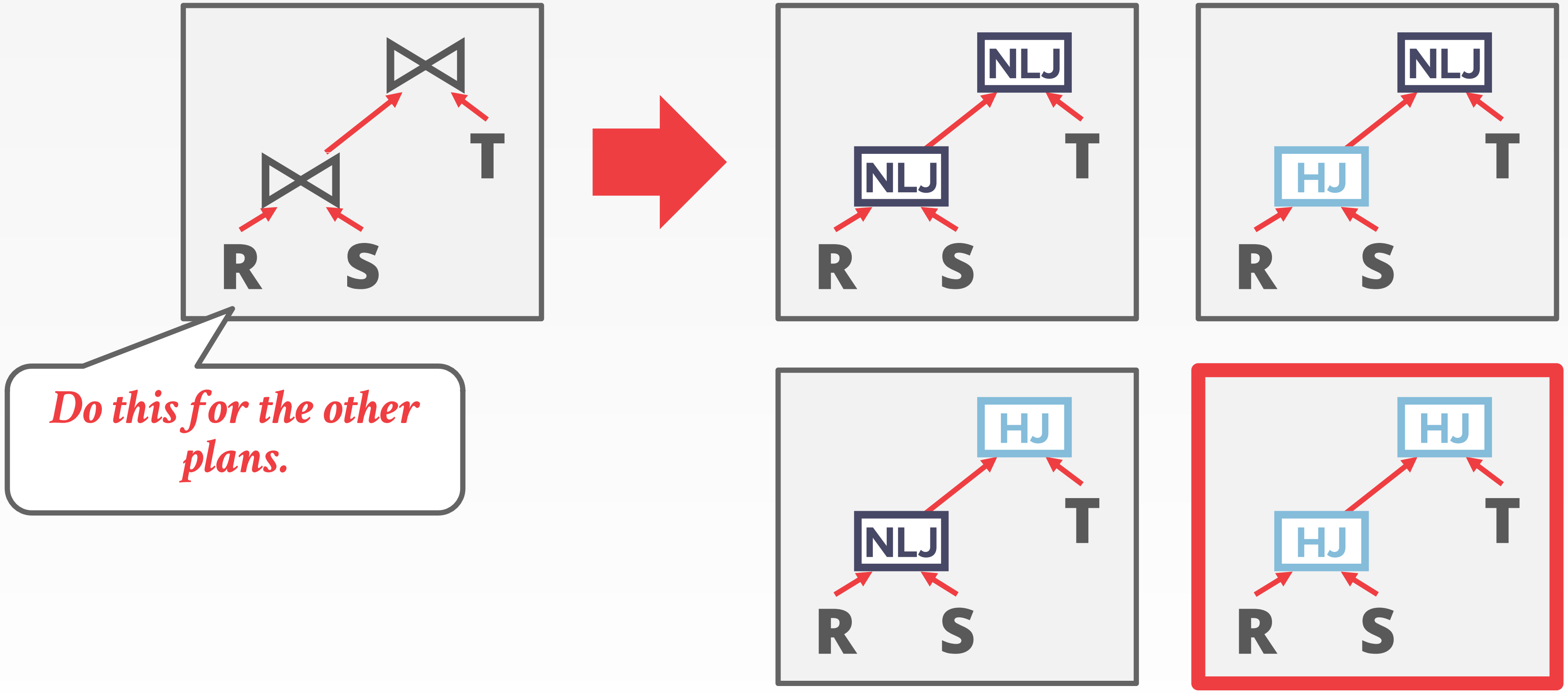
CANDIDATE PLANS

- Step #2: Enumerate join algorithm choices



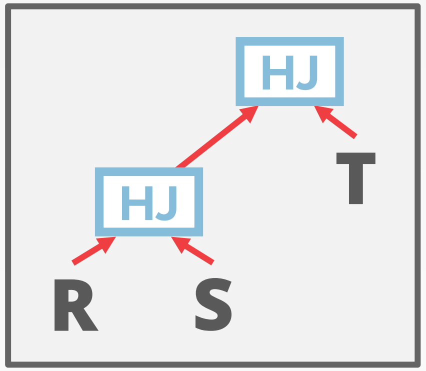
CANDIDATE PLANS

- **Step #2: Enumerate join algorithm choices**



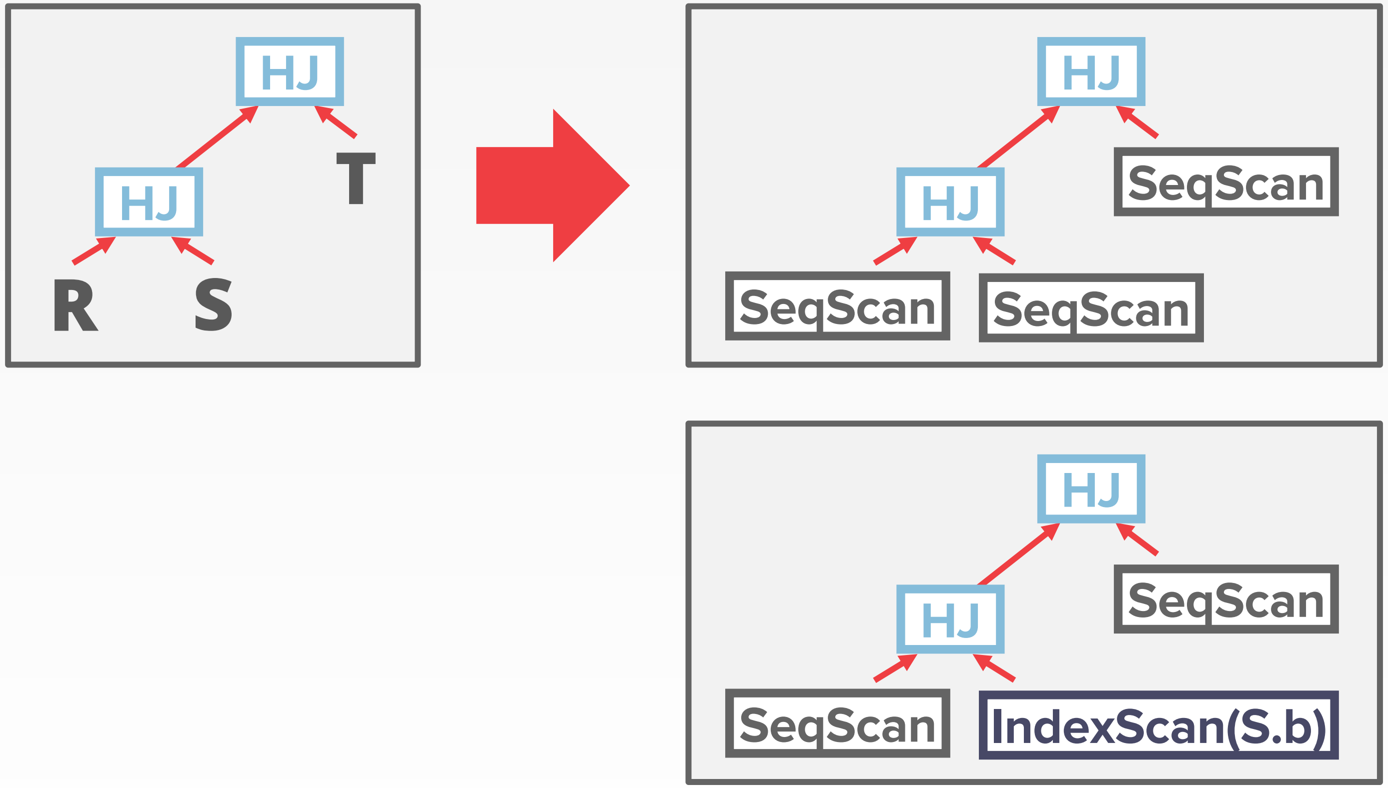
CANDIDATE PLANS

- **Step #3: Enumerate access method choices**



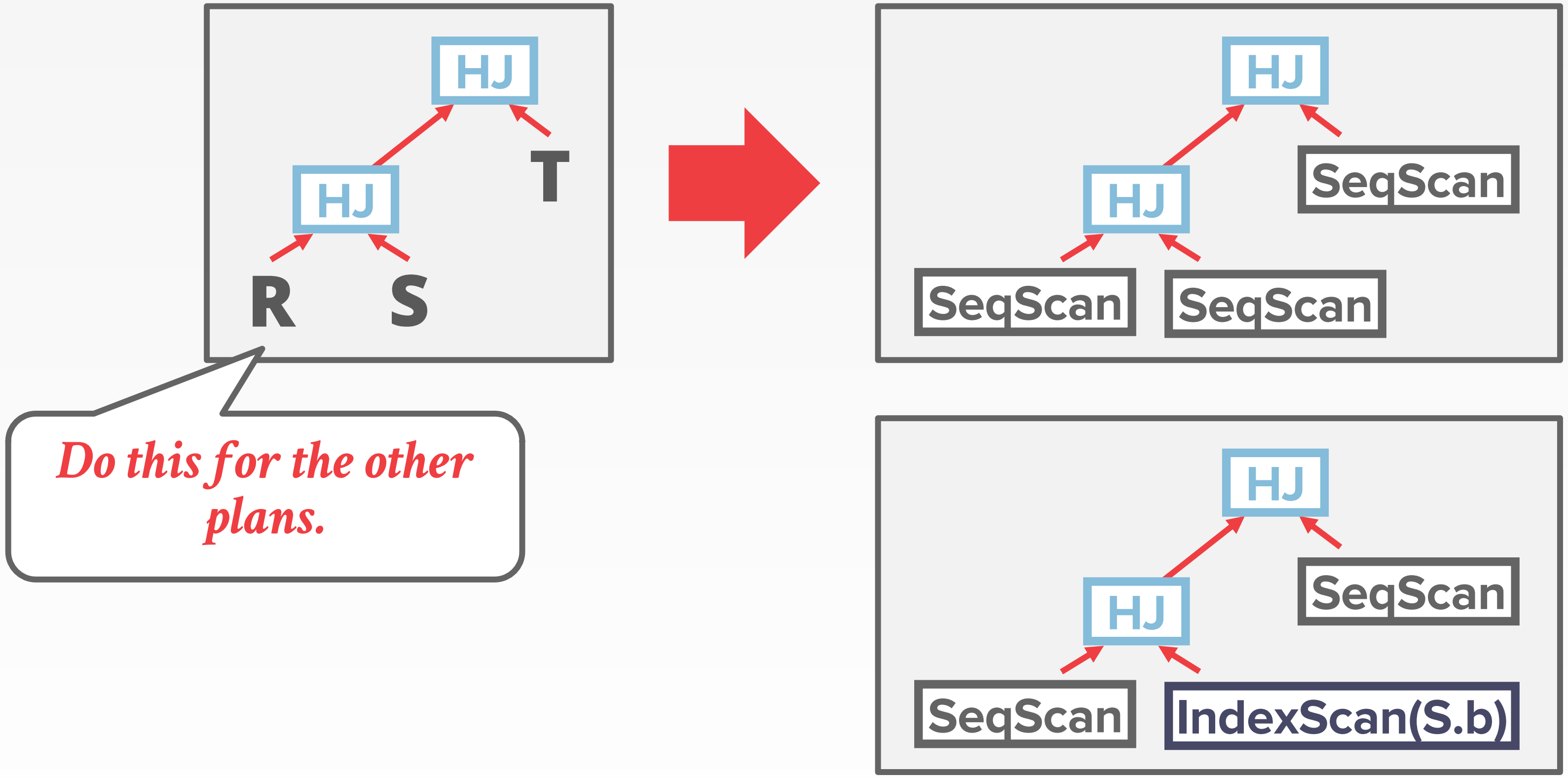
CANDIDATE PLANS

- **Step #3: Enumerate access method choices**



CANDIDATE PLANS

- **Step #3: Enumerate access method choices**

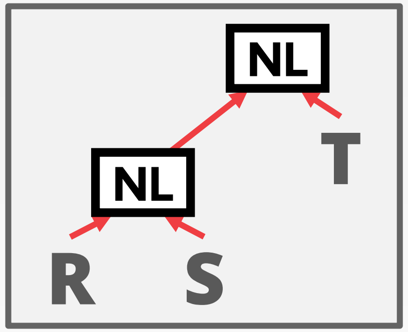


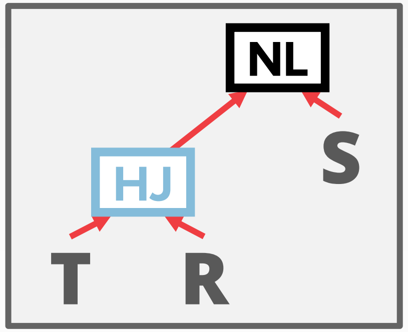
POSTGRES QUERY OPTIMIZER

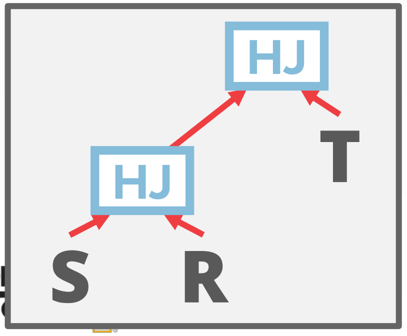
- Examines all types of join trees
  - Left-deep, Right-deep, bushy
- Two optimizer implementations:
  - Traditional Dynamic Programming Approach
  - Genetic Query Optimizer (GEQO)
- Postgres uses the traditional algorithm when # of tables in query is <u>**less**</u> than 12 and switches to GEQO when there are 12 or more.

POSTGRES QUERY OPTIMIZER

1st Generation

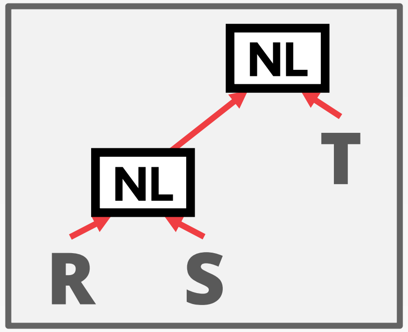




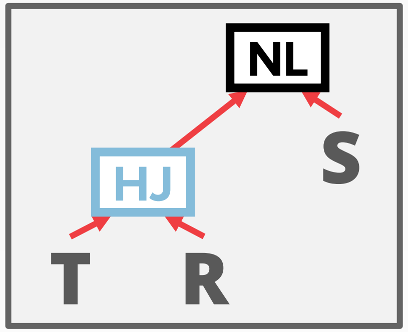


POSTGRES QUERY OPTIMIZER

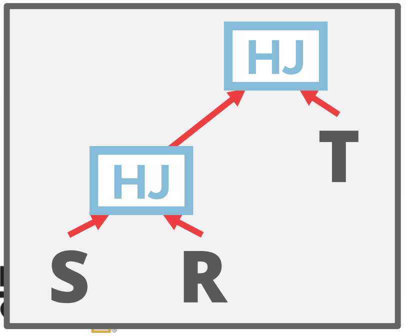
1st Generation



Cost:300

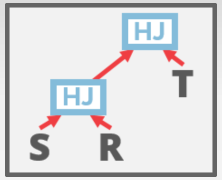


Cost:200



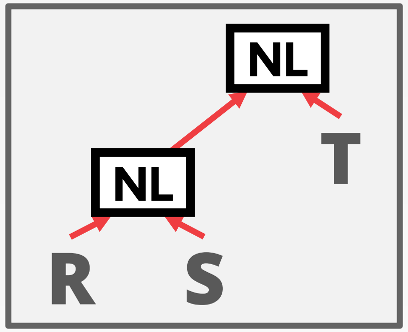
Cost: 100

POSTGRES QUERY OPTIMIZER

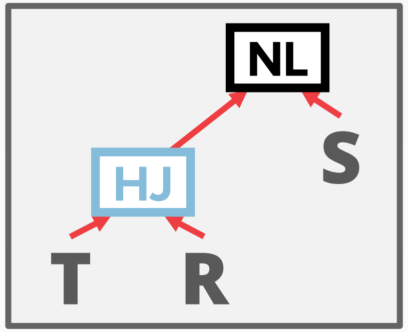


Best:100

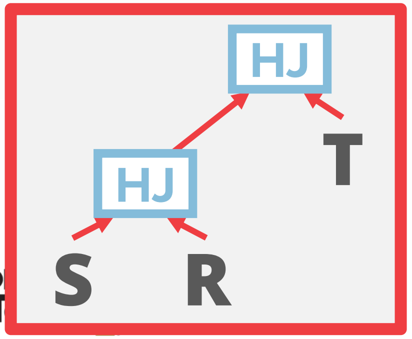
1st Generation



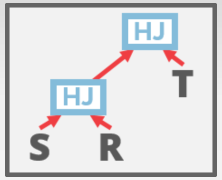
Cost:300



Cost:200



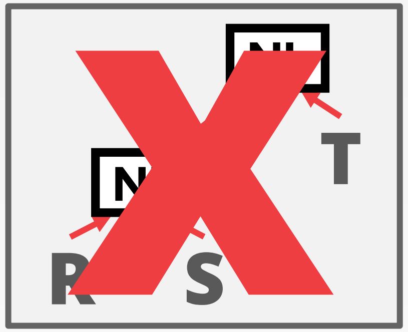
Cost: 100



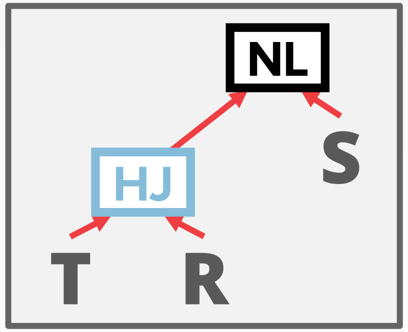
Best:100

POSTGRES QUERY OPTIMIZER

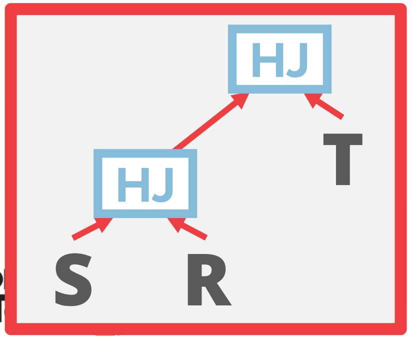
1st Generation



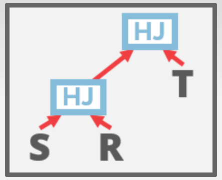
Cost:3
00



Cost:2
00



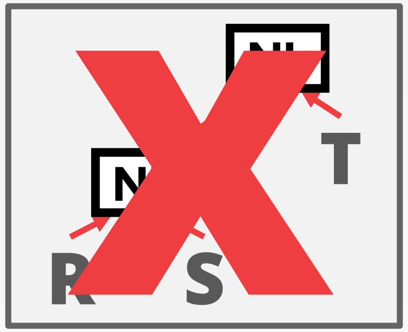
Cost:
100



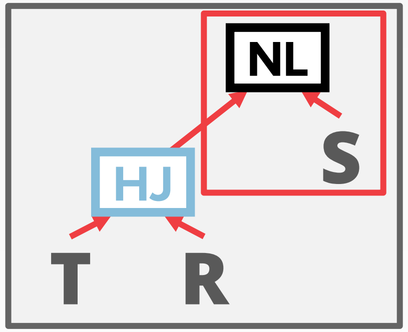
Best:100

POSTGRES QUERY OPTIMIZER

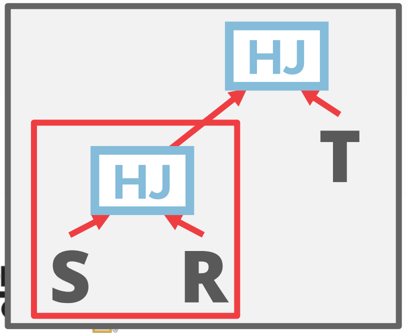
1st Generation



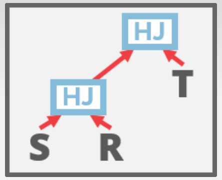
Cost:300



Cost:200



Cost:100

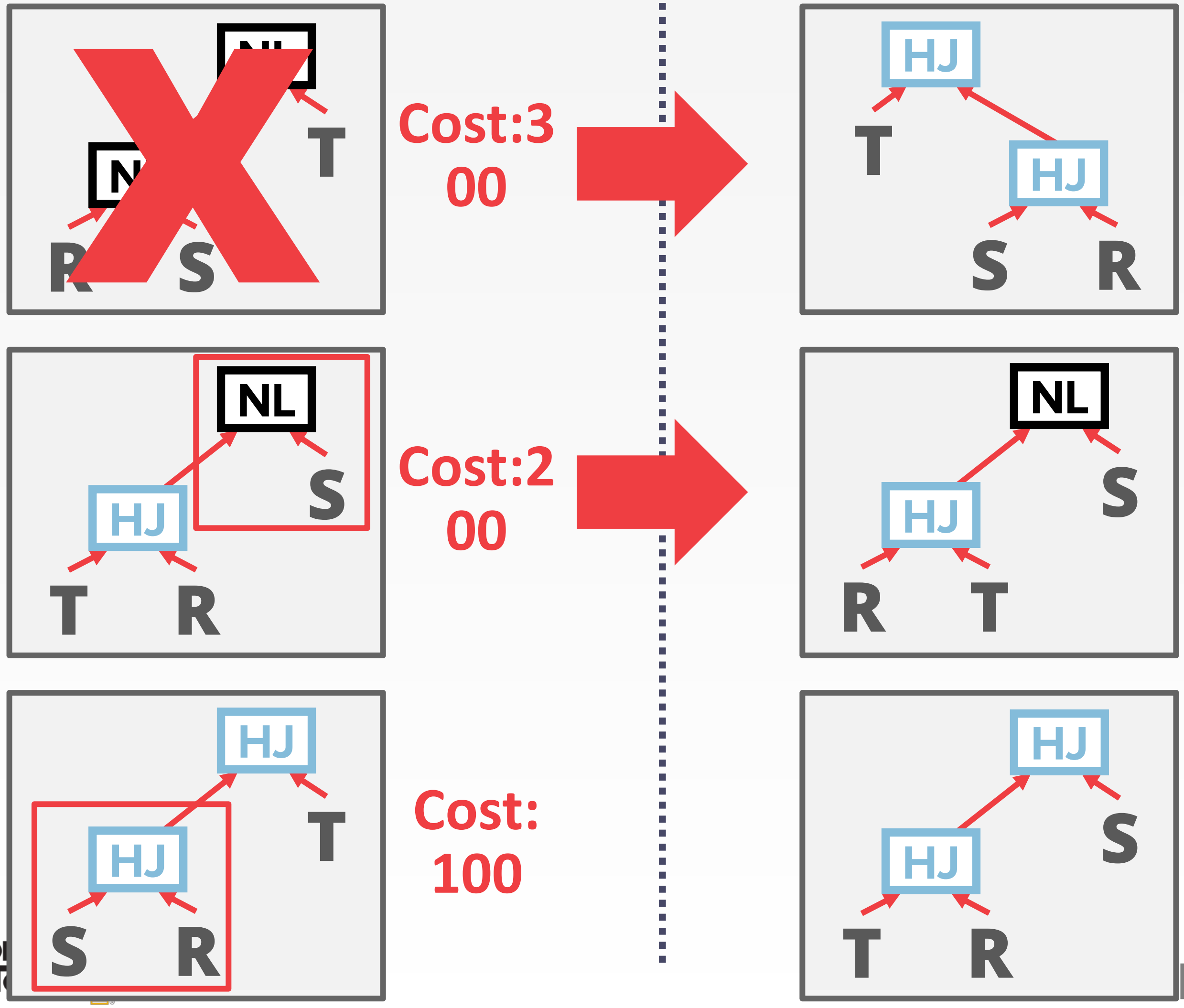


Best:100

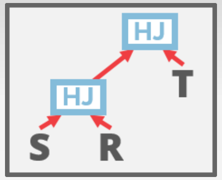
POSTGRES QUERY OPTIMIZER

1st Generation

2nd Generation

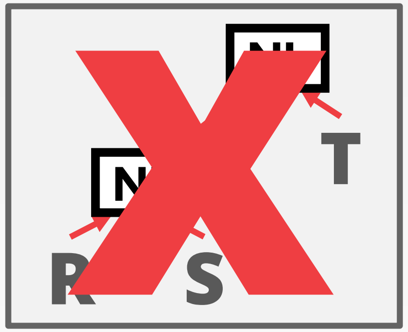


POSTGRES QUERY OPTIMIZER

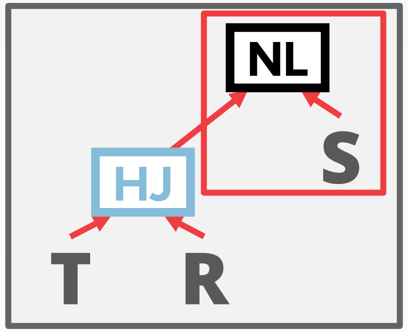


Best:100

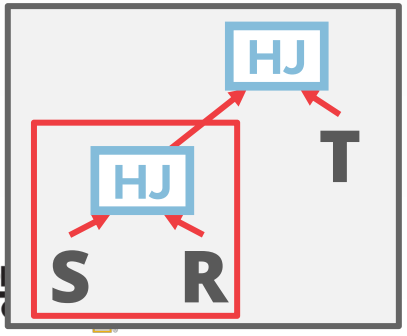
1st Generation



Cost:300

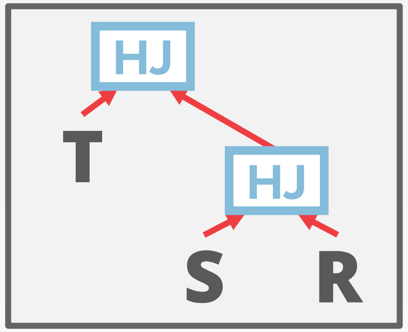


Cost:200

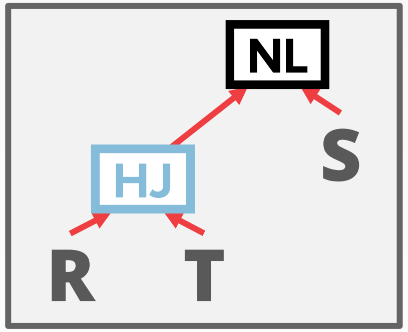


Cost: 100

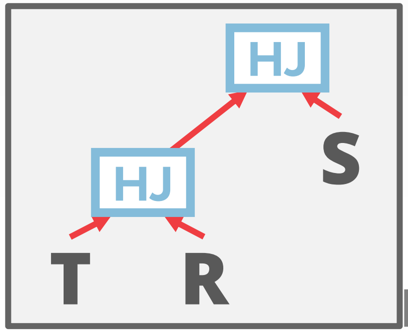
2nd Generation



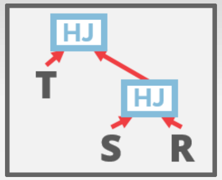
Cost: 80



Cost: 200

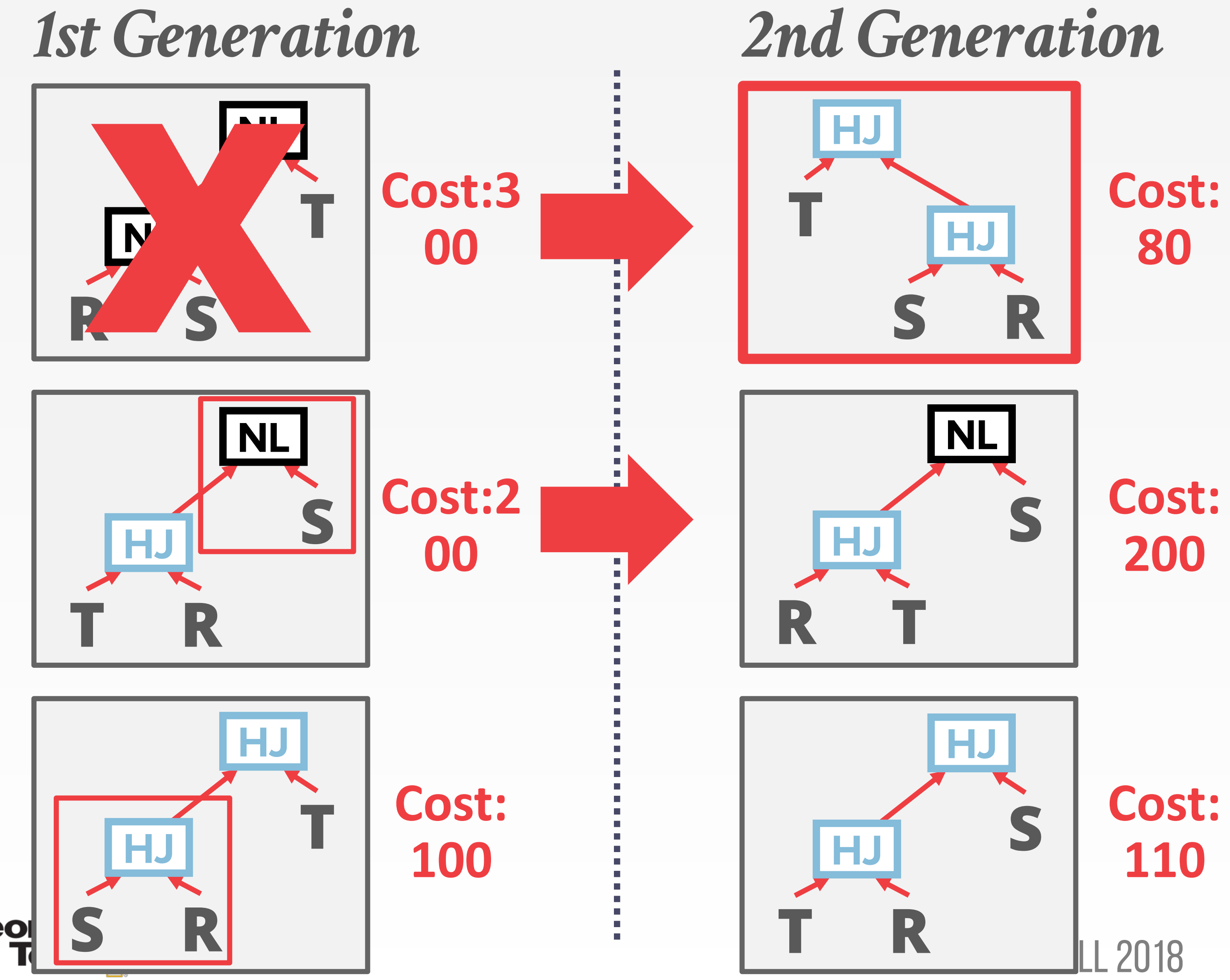


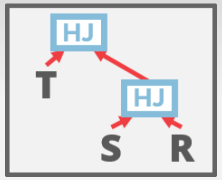
Cost: 110



Best:80

POSTGRES QUERY OPTIMIZER

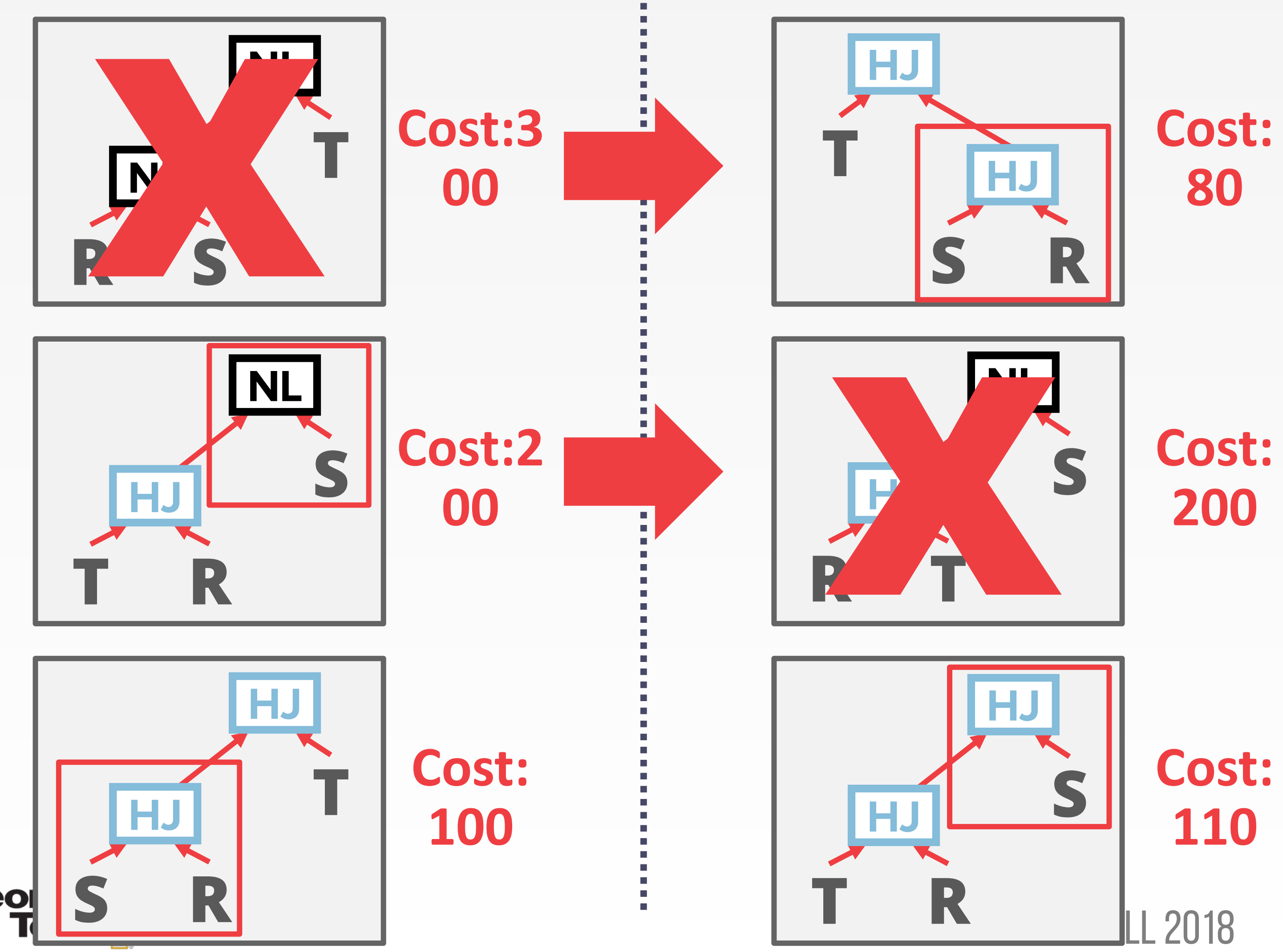




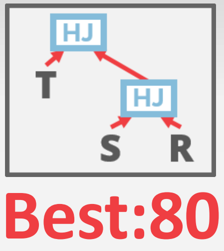
Best:80

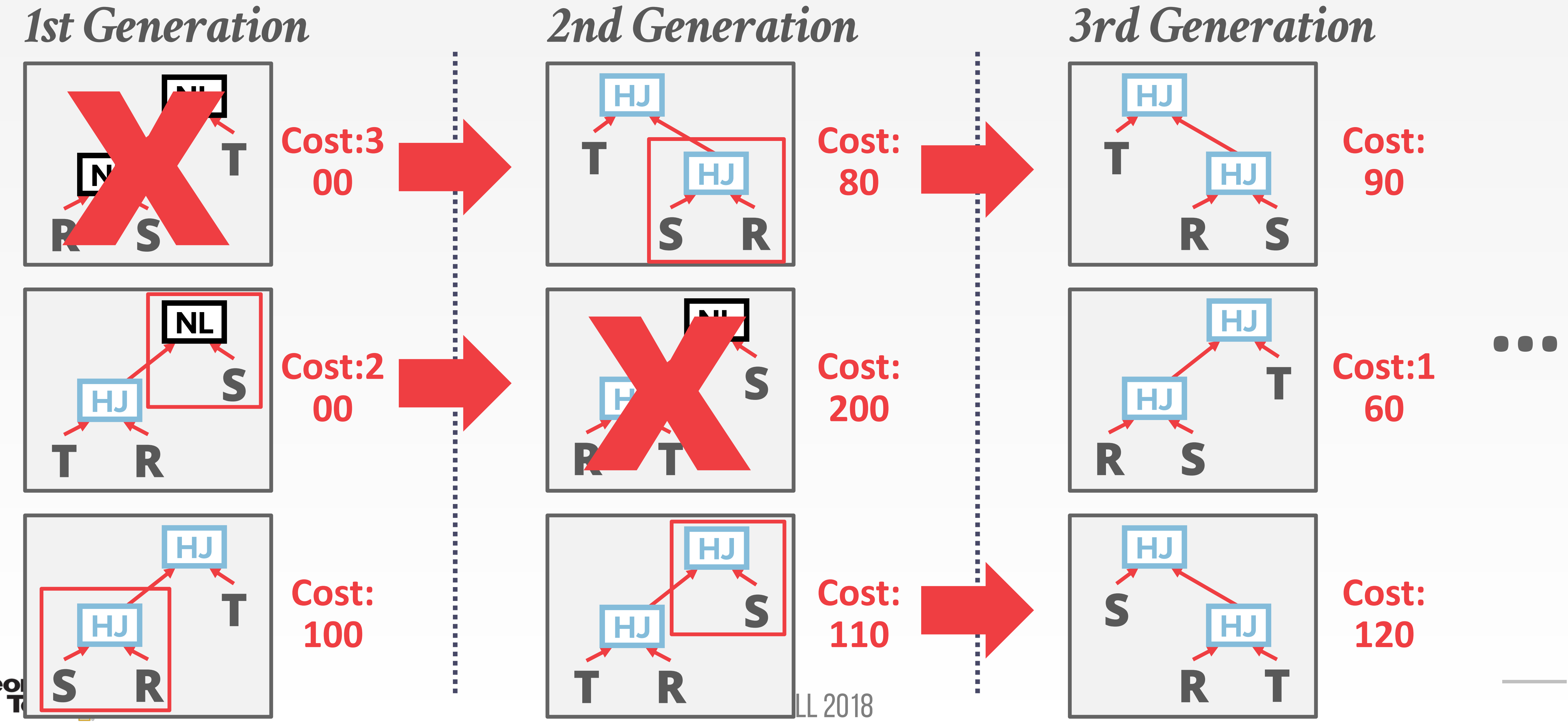
POSTGRES QUERY OPTIMIZER

1st Generation *2nd Generation*



POSTGRES QUERY OPTIMIZER





VISUAL QUERY OPTIMIZER

VISUAL QUERY OPTIMIZATION

- Queries only contain a complex predicate

```
SELECT frameID, vehType, vehColor
FROM PROCESS(inputVideo)
WHERE vehType=SUV ∧ vehColor=red;
```

- Optimization techniques
  - BlazeIt (Stanford): Rule-based optimization
  - PP (Microsoft Research): Cost-based optimization

VISUAL QUERY OPTIMIZATION

- Queries only contain a complex predicate

```
SELECT frameID, vehType, vehColor
FROM PROCESS(inputVideo)
WHERE vehType=SUV ∧ vehColor=red;
```

- Optimization techniques
  - BlazeIt (Stanford): Rule-based optimization
  - PP (Microsoft Research): Cost-based optimization

BLAZEIT: RULE-BASED OPTIMIZER

- **Example:** Content-based selection for red buses.
  - Train a specialized NN to filter frames with buses
  - But the NN may not be accurate on every frame
  - Call the object detection model on uncertain frames
  - To account for this error rate, it uses held-out set of frames to estimate the selectivity and error rate.
- Given an error budget, the optimizer selects between the filters and uses rule-based optimization to select the fastest query plan

BLAZEIT: RULE-BASED OPTIMIZER

- **Example:** Choosing a filter
- Consider two possible filters for redness:
  - F_1 : A filter which returns true if the over 80% of the pixels have a red-channel value of at least 200
  - F_2: A filter that returns the average of the red-channel values

BLAZEIT: RULE-BASED OPTIMIZER

- In estimating thresholds at the frame-level based on frames from the held-out set, it learns that:
  - **sel(F_1)** = 0.9 and **sel(F_2)** = 0.3
- Which filter should it pick?
  - More selective filter (**F_2**)

PP: COST-BASED OPTIMIZER

- Decompose a complex predicate to expressions over simple predicates
  - **Old:** `<vehType=SUV AND vehColor=red>`
  - **New:** `<`**`vehType=SUV`**`> ∧ <`**`vehColor=red`**`>`
- Rewrite rules (logical equivalences):
  - $p \wedge (P_{rest}) \Rightarrow Filter_{p}$
  - $Filter_{p \wedge q} \Rightarrow Filter_{p} \wedge Filter_{q}$
  - $Filter_{p \vee q} \Rightarrow Filter_{p} \vee Filter_{q}$

PP: COST-BASED OPTIMIZER

- Sort the list of available filters based on:
  - Filter evaluation cost (**C**)
  - Data reduction ratio (**R[Accuracy]**)
- Efficacy of filter = **C / R[1]**
  - A smaller ratio of cost to data reduction indicates better performance

PP: COST-BASED OPTIMIZER

- **Example:** $(p \lor q) \land \neg r \land P_{rest}$
- $\Rightarrow p \lor q \Rightarrow F_{p \lor q} \Rightarrow F_p \lor F_q$
- $\Rightarrow \neg r \Rightarrow F_{\neg r}$
- $\Rightarrow F_{(p \lor q) \land \neg r} \Rightarrow (F_p \lor F_q) \land F_{\neg r}$
- $\Rightarrow F_{(p \land \neg r) \lor (q \land \neg r)} \Rightarrow F_{p \land \neg r} \lor F_{q \land \neg r}$
  $\Rightarrow (F_p \land F_{\neg r}) \lor (F_q \land F_{\neg r})$

PP: COST-BASED OPTIMIZER

- Pruning search space
  – Limit the number of different filters to be a small constant (**k**)
- **Example:**
  – Available filters: $\{F_{p \vee q}, F_p, F_{p \wedge \neg r}, F_{q \wedge \neg r}, F_q, F_{\neg r}\}$
  – Query requirements: $\{F_{p \vee q}, F_p \vee F_q, F_{\neg r}, (F_p \vee F_q) \wedge F_{\neg r}, F_{p \wedge \neg r} \vee F_{q \wedge \neg r}\}$
  – **k = 2**
  – Candidate plans: $\{F_{p \vee q}, F_{\neg r}, F_{p \wedge \neg r} \vee F_{q \wedge \neg r}\}$

PP: COST-BASED OPTIMIZER

- Plan Enumeration
  - First, explore different allocations of the query's accuracy budget to individual filters.
  - Next, explore different orderings of filters within a conjunction or disjunction.
- Cost Estimation
  - Finally, after fixing both the accuracy thresholds and the order of filters, compute the cost and reduction rate of the resulting plan.

PARTING THOUGHTS

- Filter as early as possible.
- Filter selectivity estimations
  - Uniformity, Independence, Histograms
- Dynamic programming for join orderings
- Again, query optimization is super important.

NEXT LECTURE

- Convolutional neural networks
  - Popular neural network architecture