Optimization

BBM471 Database Management Systems
Dr. Fuat Akal
akal@hacettepe.edu.tr

Today's Lecture

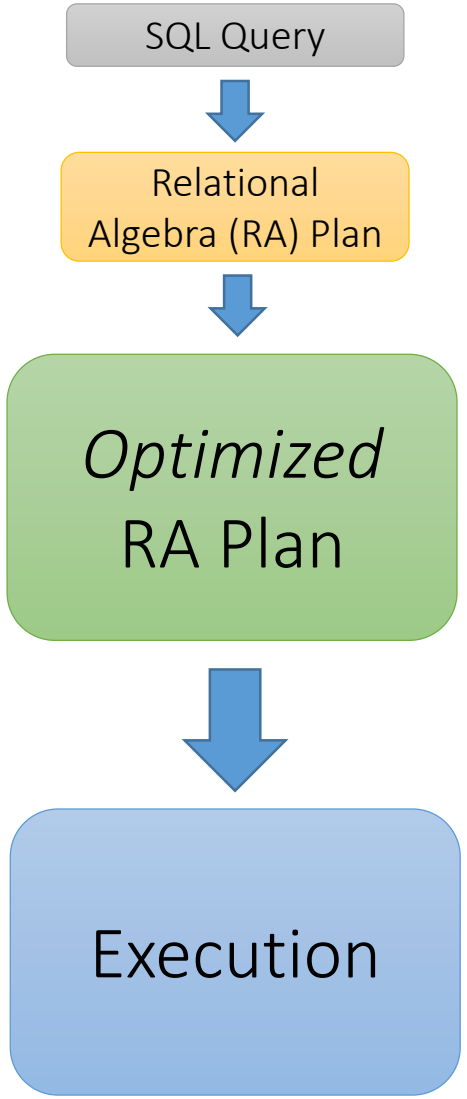
1. Logical Optimization

2. Physical Optimization

Logical vs. Physical Optimization

- **Logical optimization:**
  - Find equivalent plans that are more efficient
  - *Intuition: Minimize # of tuples at each step by changing the order of RA operators*

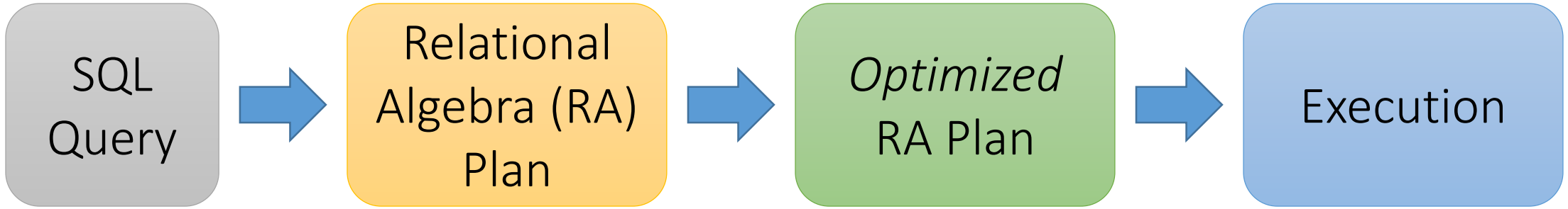
- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



1. Logical Optimization

RDBMS Architecture

How does a SQL engine work ?



Declarative query (from user)

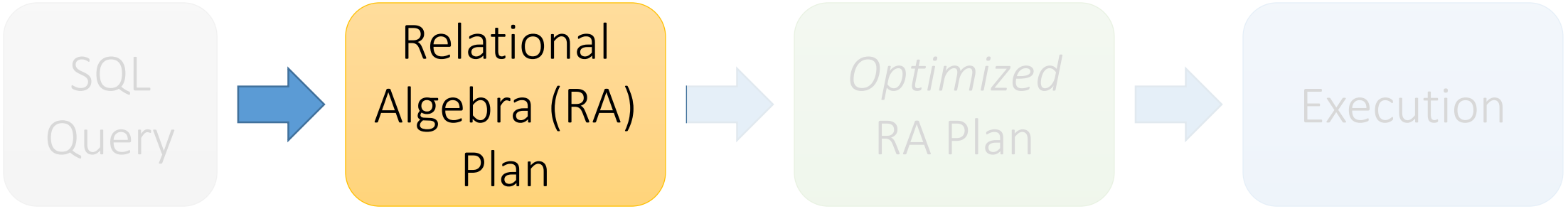
Translate to relational algebra expresson

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

RDBMS Architecture

How does a SQL engine work ?



Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!

What does DBMS Do When You submit a Query?

- Translates SQL into get/put req. to backend storage
- Extracts, processes, transforms tuples from blocks
- Performs tons of optimizations
  - Choosing algorithms for SQL operators (hashing, sorting)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Parallel execution and concurrency
  - Load and admission control
  - Layout of data on backend storage
  - Buffer management and caching
  - …

Input: SQL statement
Output: {tuples}

DBMS vs. OS Optimizations

- Many DBMS tasks are also carried out by OS
  - Load control
  - Buffer management
  - Access to external storage
  - Scheduling of processes

- What is the difference?
  - DBMS has intimate knowledge of workload
  - DBMS can predict and shape access pattern of a query
  - DBMS knows the contention between queries
  - OS does generic optimizations

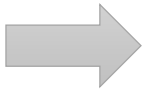
Recall: Relational Algebra (RA)

- Five **basic** operators:
  1. Selection: σ
  2. Projection: Π
  3. Cartesian Product: $\times$
  4. Union: $\cup$
  5. Difference: -
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ
  - Division

Recall: Converting SFW Query -> RA

Students(sid,name,gpa)
People(ssn,name,address)

```
SELECT DISTINCT
  gpa,
  address
FROM Students S,
     People P
WHERE gpa > 3.5 AND
   s.name = p.name;
```



$$\Pi_{gpa,address}(\sigma_{gpa>3.5}(S \bowtie P))$$

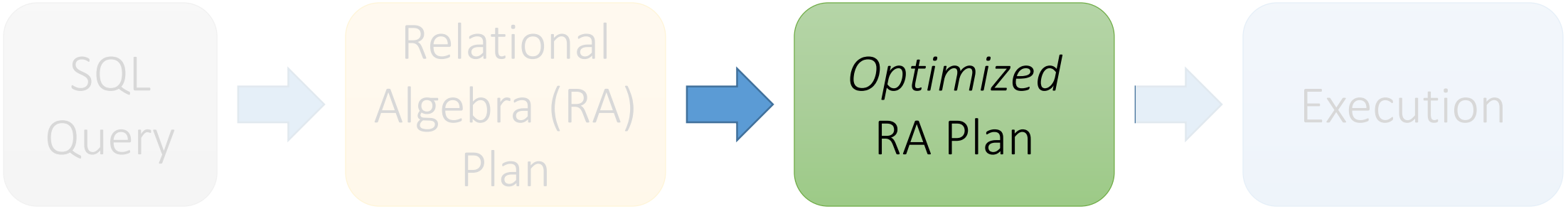
How do we represent this query in RA?

Recall: Logical Equivalence of RA Plans

- Given relations R(A,B) and S(B,C):
  - Here, projection & selection commute:
    - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
  - What about here?
    - $\sigma_{A=5}(\Pi_B(R)) \,?= \Pi_B(\sigma_{A=5}(R))$

RDBMS Architecture

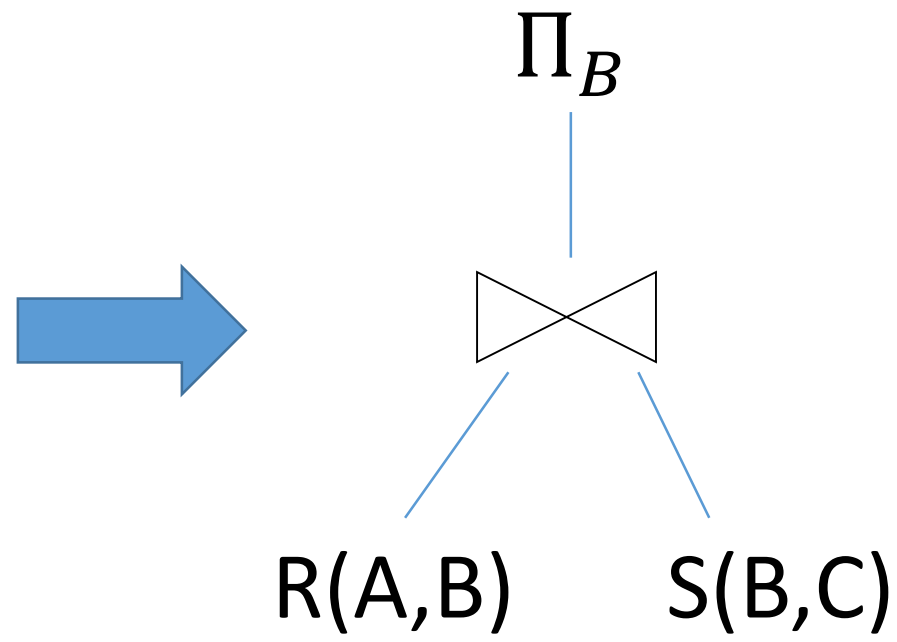
How does a SQL engine work ?



We'll look at how to then optimize these plans now

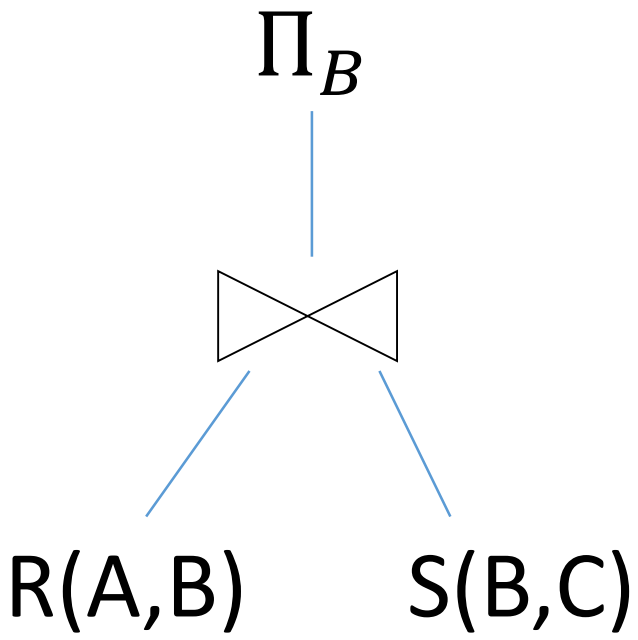
Note: We can visualize the plan as a tree

$$\Pi_B(R(A,B) \bowtie S(B,C))$$



Bottom-up tree traversal = order of operation execution!

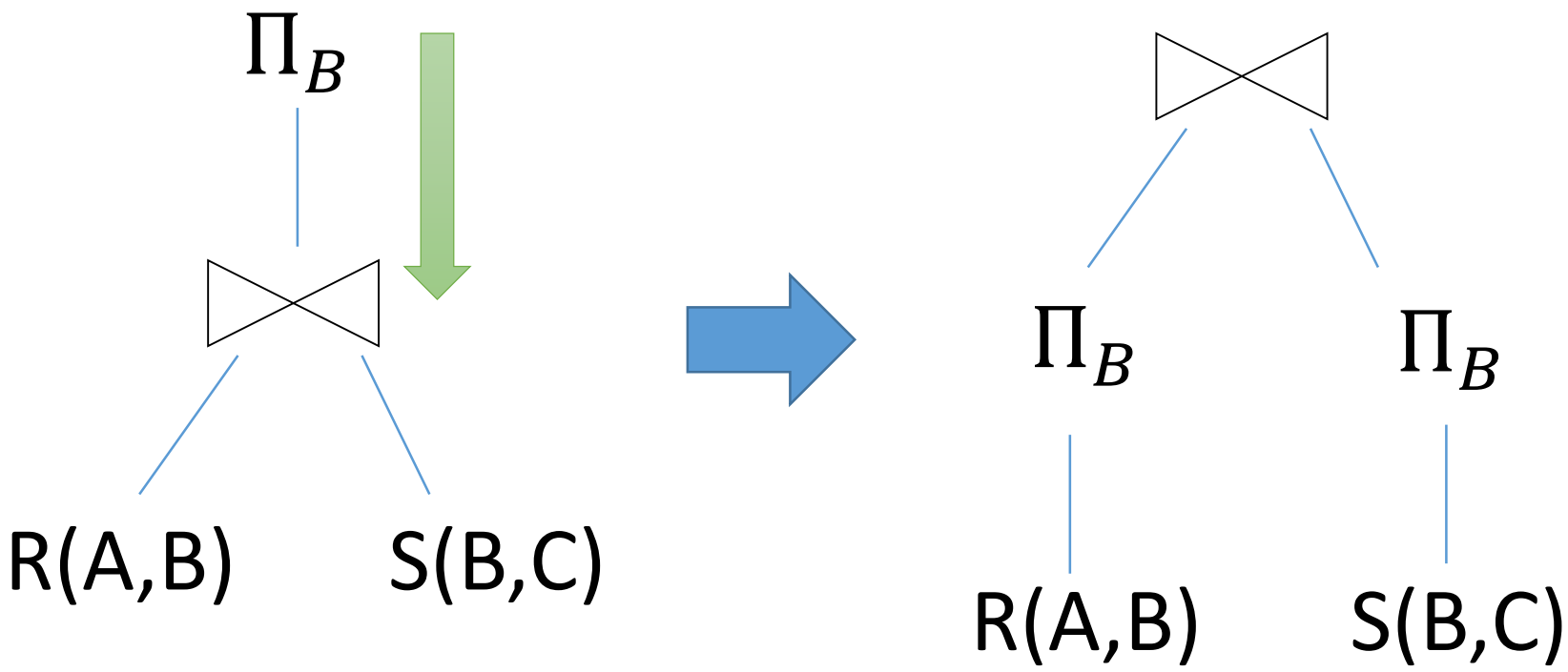
A simple plan



What SQL query does this correspond to?

Are there any logically equivalent RA expressions?

“Pushing down” projection



Why might we prefer this plan?

Takeaways

- This process is called logical optimization
- Many equivalent plans used to search for “good plans”
- Relational algebra is an important abstraction

RA commutators

- The basic commutators:
  - Push **projection** through **(1) selection**, **(2) join**
  - Push **selection** through **(3) selection, (4) projection, (5) join**
  - *Also:* Joins can be re-ordered!

- Note that this is not an exhaustive set of operations
  - This covers *local re-writes; global re-writes possible but much harder*

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!

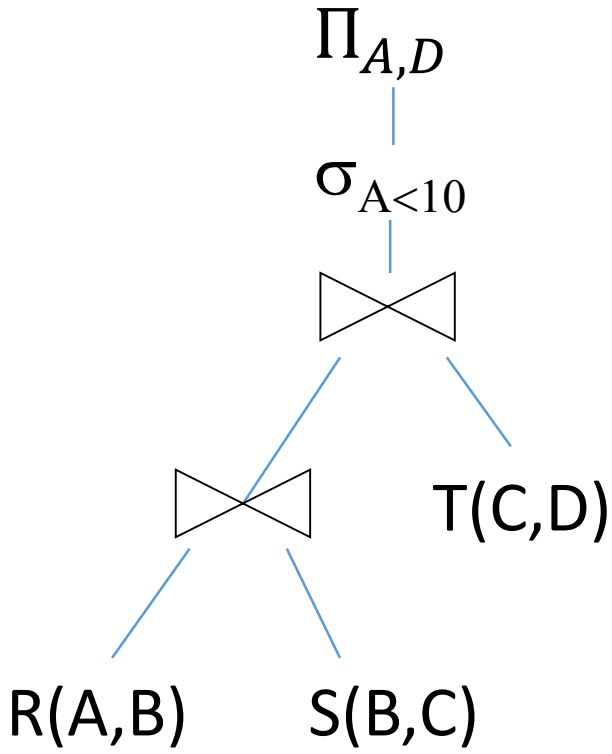
Optimizing the SFW RA Plan

Translating to RA

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$



Logical Optimization

- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology: “push down **selections**” and “pushing down **projections.**”

- **Intuition:** We will have fewer tuples in a plan.
  - Could fail if the selection condition is very expensive (say runs some image processing algorithm).
  - Projection could be a waste of effort, but more rarely.

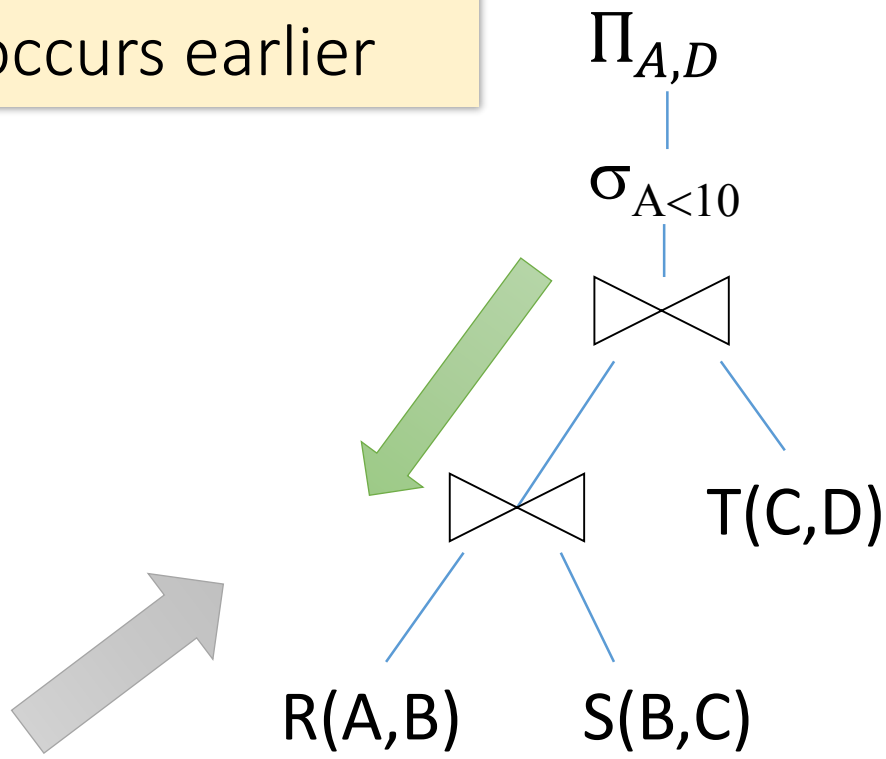
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down selection on A so it occurs earlier

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$



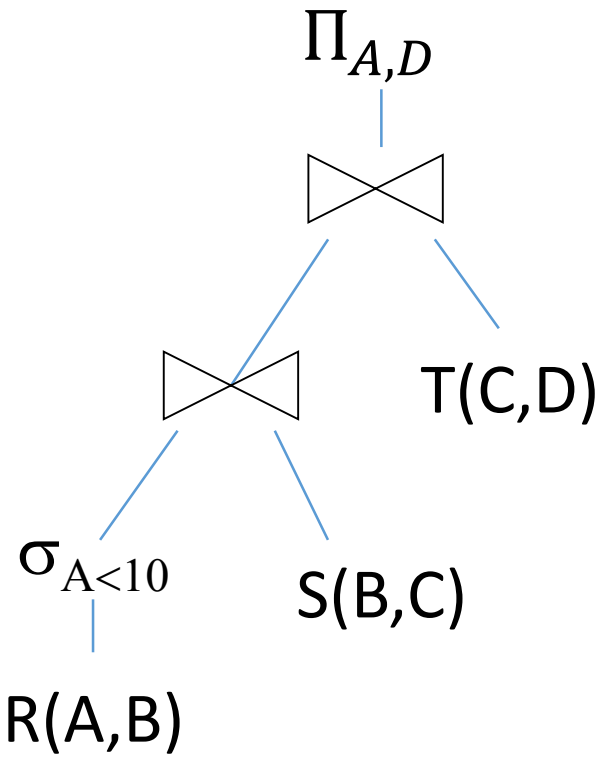
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down selection on A so it occurs earlier

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$



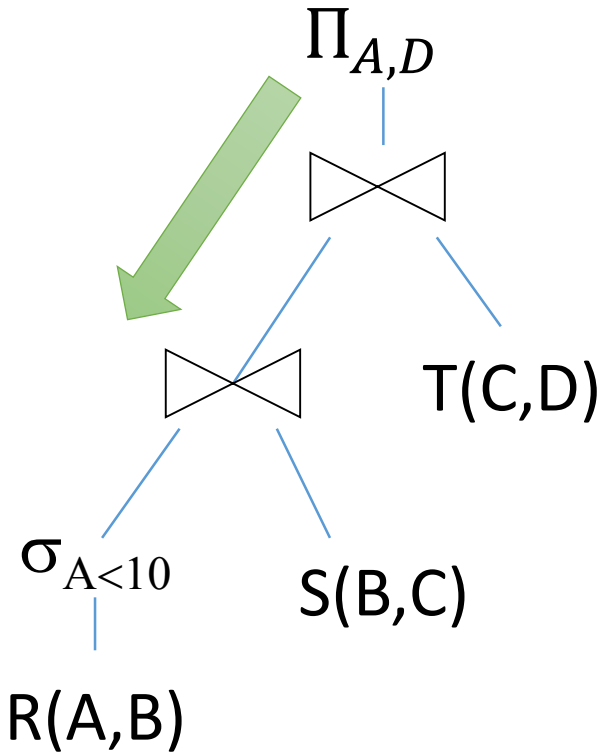
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down projection so it occurs earlier

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$



Optimizing RA Plan

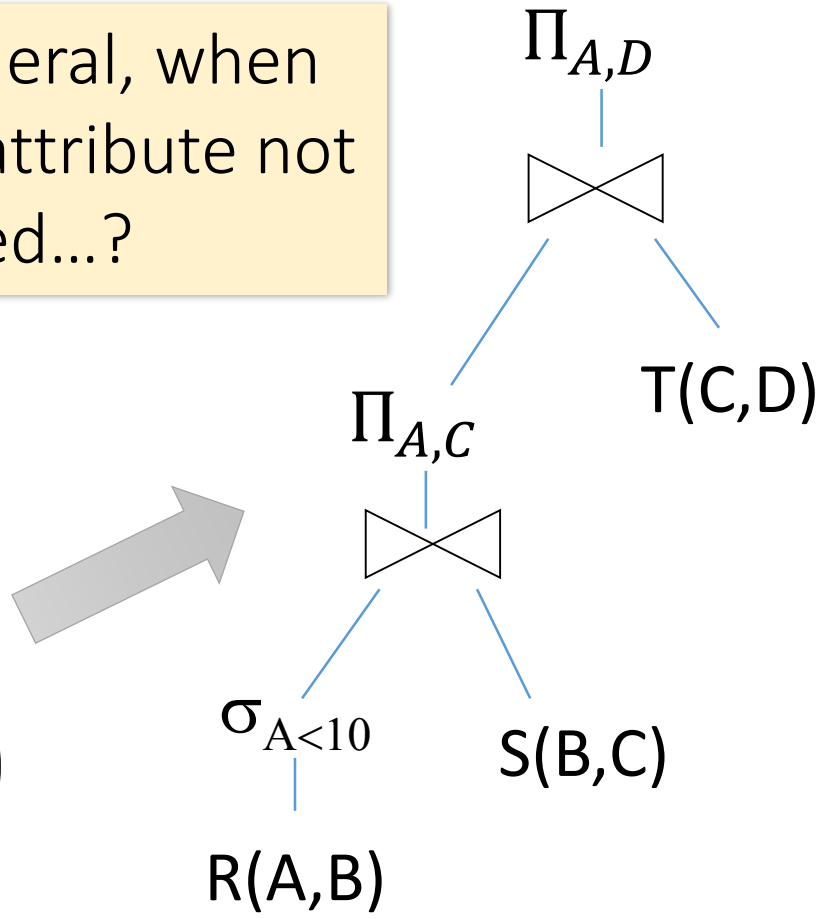
R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

We eliminate B earlier!

In general, when is an attribute not needed…?

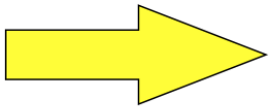
$$\Pi_{A,D}\left(T \bowtie \Pi_{A,C}(\sigma_{A<10}(R) \bowtie S)\right)$$



Query Rewrite: Unnesting of Views

- Example: Unnesting of Views

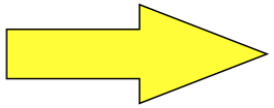
```
select A.x
from   A
where y in
   (select y from B)
```



```
select A.x
from   A, B
where A.y = B.y
```

- Example: Unnesting of Views

```
select A.x
from   A
where exists
   (select * from B where A.y = B-y)
```



```
select A.x
from   A, B
where A.y = B.y
```

Query Rewrite: Predicate Augmentation

- Example: Predicate Augmentation

```
select *
from   A, B, C
where  A.x = B.x
 and B.x = C.x
```

```
select *
from   A, B, C
where  A.x = B.x
 and B.x = C.x
 and A.x = C.x
```

Why is that useful?

Why Predicate Augmentation?

A (odd numbers)

...	x
...	1
...	3
...	5
...	...

B (all numbers)

...	x
...	1
...	2
...	3
...	...

C (even numbers)

...	x
...	2
...	4
...	6
...	...

- Cost((A x C) x B) < Cost((A x B) x C)
  - get second join for free
- Query Rewrite does not know that, ...
  - but it knows that it might happen and hopes for optimizer...
- Codegen gets rid of unnecessary predicates (e.g., A.x = B.x)

Query Optimization

- Two tasks
  - Determine order of operators
  - Determine algorithm for each operator (hashing, sorting, ...)
- Components of a query optimizer
  - Search space
  - Cost model
  - Enumeration algorithm (NP hard)
- Working principle
  - Enumerate alternative plans
  - Apply cost model to alternative plans
  - Select plan with lowest expected cost

Optimization: Does It Really Matter? - 1

- A ⋈ B ⋈ C
  - size(A) = 10,000
  - size(B) = 100
  - size(C) = 1
  - cost(X ⋈ Y) = size(X) + size(Y)

- cost((A ⋈ B) ⋈ C) = 1,010,101
  - cost(A ⋈ B) = 10,100
  - cost(**X** ⋈ C) = 1,000,001 with **X = A ⋈ B**

- cost (A ⋈ (B ⋈ C)) = 10,201
  - cost(B ⋈ C) = 101
  - cost(A ⋈ **X**) = 10,100 with **X = B ⋈ C**

Optimization: Does It Really Matter? - 2

- A ⋈ B ⋈ C
  - size(A) = 1000
  - size(B) = 1
  - size(C) = 1
  - cost(X ⋈ Y) = size(X) * size(Y)

- cost((A ⋈ B) ⋈ C) = 2000
  - cost(A ⋈ B) = 1000
  - cost(**X** ⋈ C) = 1000 with **X = A ⋈ B**

- cost (A ⋈ (B ⋈ C)) = 1001
  - cost(B ⋈ C) = 1
  - cost(A ⋈ **X**) = 1000 with **X = B ⋈ C**

2. Physical Optimization

What you will learn about in this section

1. Index Selection
2. Histograms
3. Materialized Views

Index Selection

Input:

- Schema of the database
- **Workload description:** set of (query template, frequency) pairs

Goal: Select a set of indexes that minimize execution time of the workload.

- Cost / benefit balance: Each additional index may help with some queries, but requires updating

This is an optimization problem!

Example

Workload description:

```
SELECT pname
FROM Product
WHERE year = ? AND category = ?
```

Frequency
10,000,000

```
SELECT pname,
FROM Product
WHERE year = ? AND Category = ?
AND manufacturer = ?
```

Frequency
10,000,000

Which indexes might we choose?

Example

Workload description:

```
SELECT pname
FROM Product
WHERE year = ? AND category =?
```

Frequency 10,000,000

```
SELECT pname
FROM Product
WHERE year = ? AND Category =?
AND manufacturer = ?
```

Frequency 100

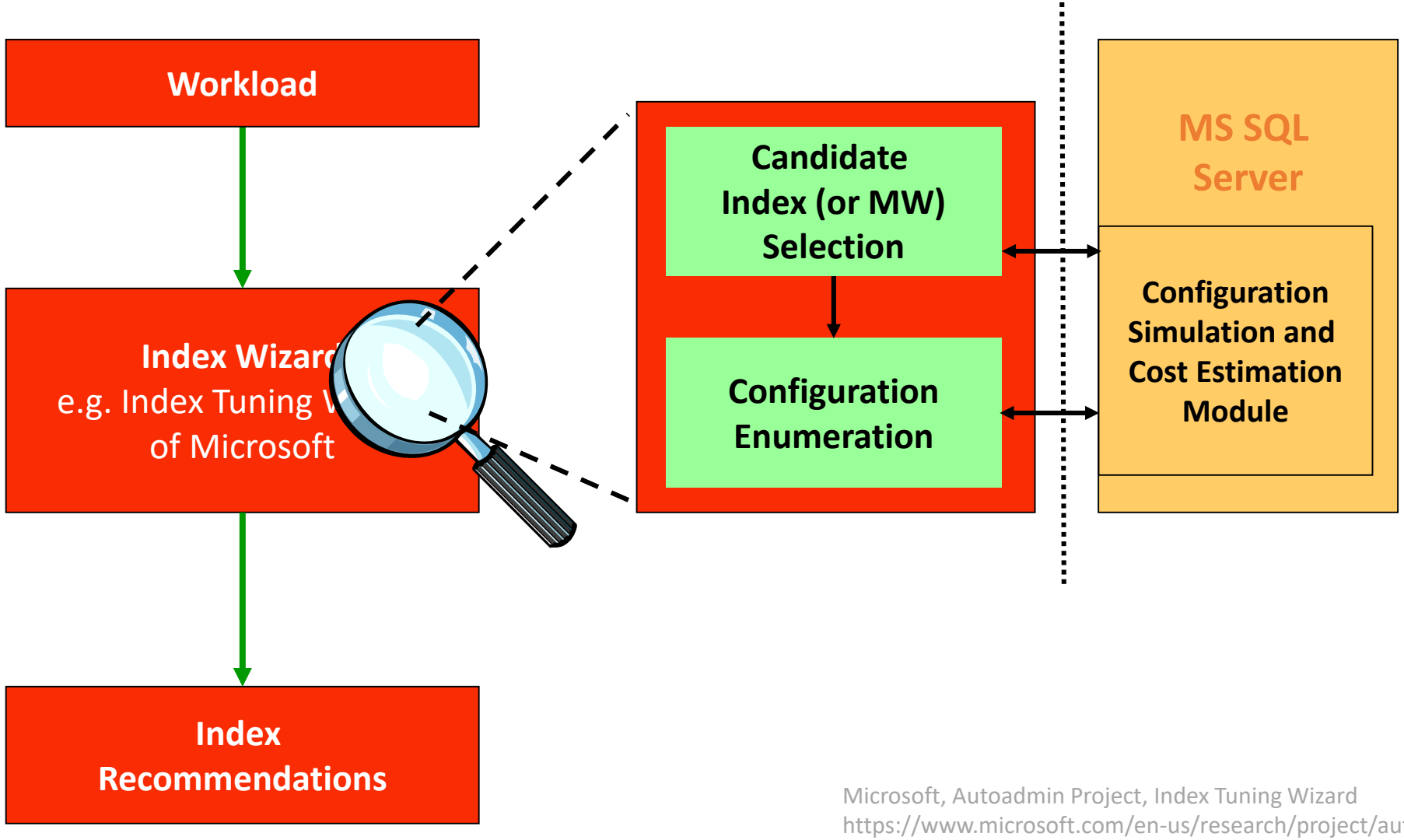
Now which indexes might we choose? Worth keeping an index with manufacturer in its search key around?

Simple Heuristic

- Can be framed as standard optimization problem: Estimate how cost changes when we add index.
    - We can ask the optimizer!
- Search over all possible space is too expensive, optimization surface is really nasty.
    - Real DBs may have 1000s of tables!
- Techniques to exploit *structure* of the space.
    - In SQL Server Autoadmin.

NP-hard problem, but can be solved!

Automatic Database Design (Physical Layout)



Microsoft, Autoadmin Project, Index Tuning Wizard
https://www.microsoft.com/en-us/research/project/autoadmin/

Estimating index cost?

- Note that to frame as optimization problem, we first need an estimate of the ***cost*** of an index lookup

- Need to be able to estimate the costs of different indexes / index types…

We will see this mainly depends on getting estimates of result set size!

Histograms & IO Cost Estimation

IO Cost Estimation via Histograms

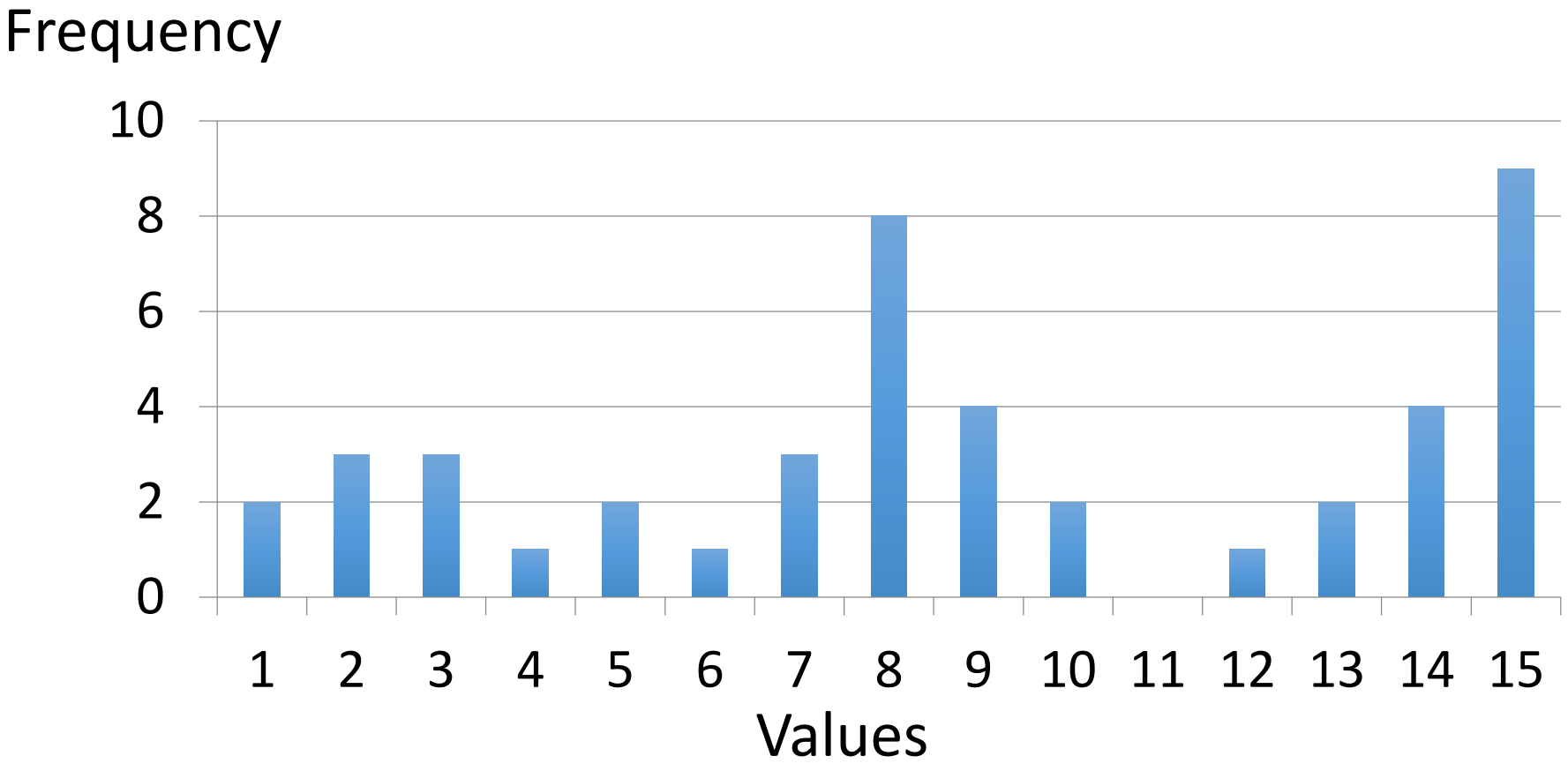
- For **index selection**:
  - What is the cost of an index lookup?
- Also for **deciding which algorithm to use**:
  - Ex: To execute $\mathrm{R} \bowtie S$, which join algorithm should DBMS use?
  - **What if we want to compute $\boldsymbol{\sigma_{A>10}(\mathrm{R}) \bowtie \sigma_{B=1}(S)}$?**
- In general, we will need some way to ***estimate intermediate result set sizes***

Histograms provide a way to efficiently store estimates of these quantities

Histograms

- A histogram is a set of value ranges (“buckets”) and the frequencies of values in those buckets occurring
- Can be used to estimate cardinality of result sets
- How to choose the buckets?
  - Equiwidth & Equidepth
- Turns out high-frequency values are **very** important

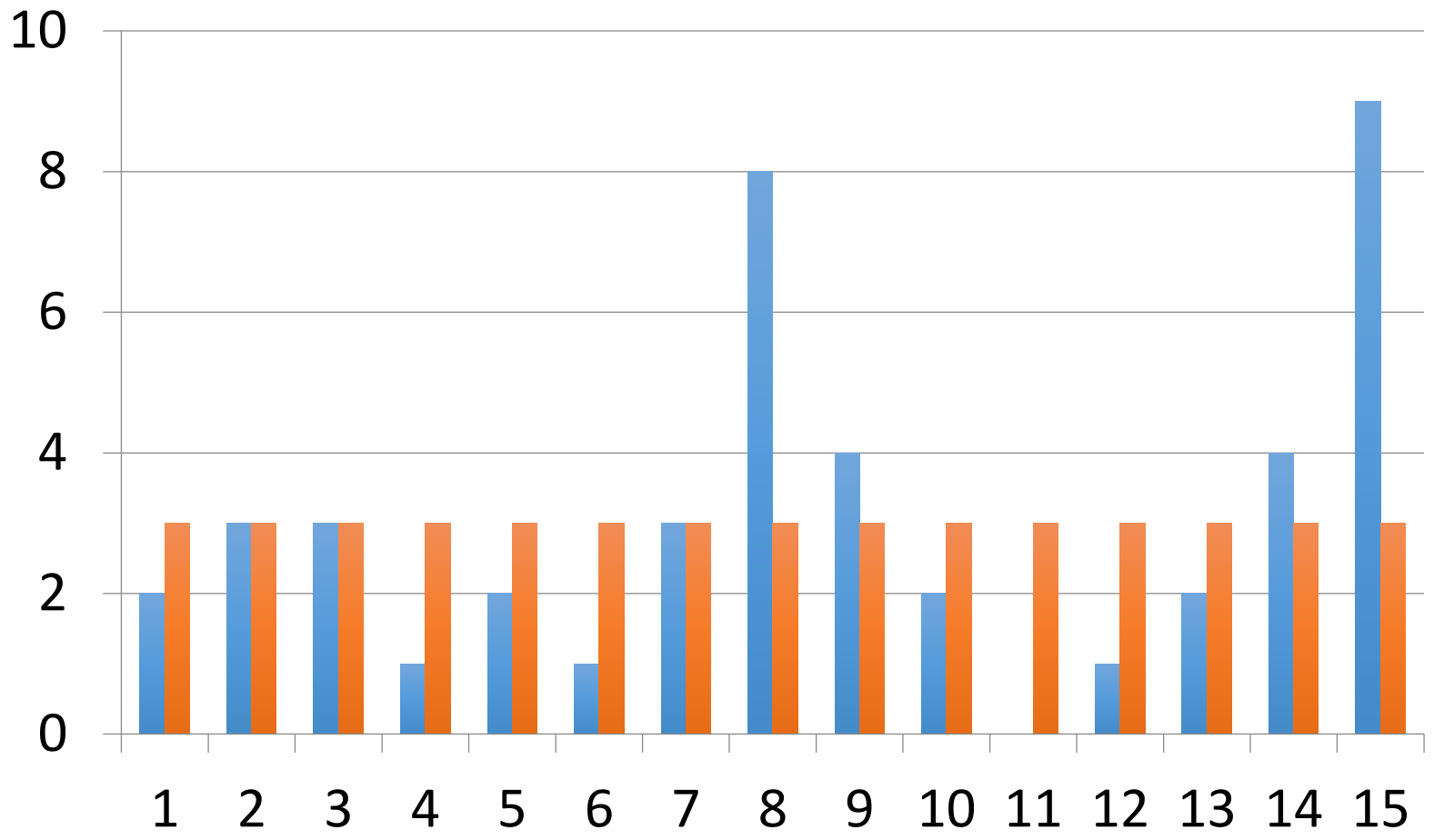
Example Histogram



How do we compute how many values between 8 and 10?
(Yes, it's obvious)

Problem: counts take up too much space!

Full vs. Uniform Counts



How much space do the full counts (bucket_size=1) take?

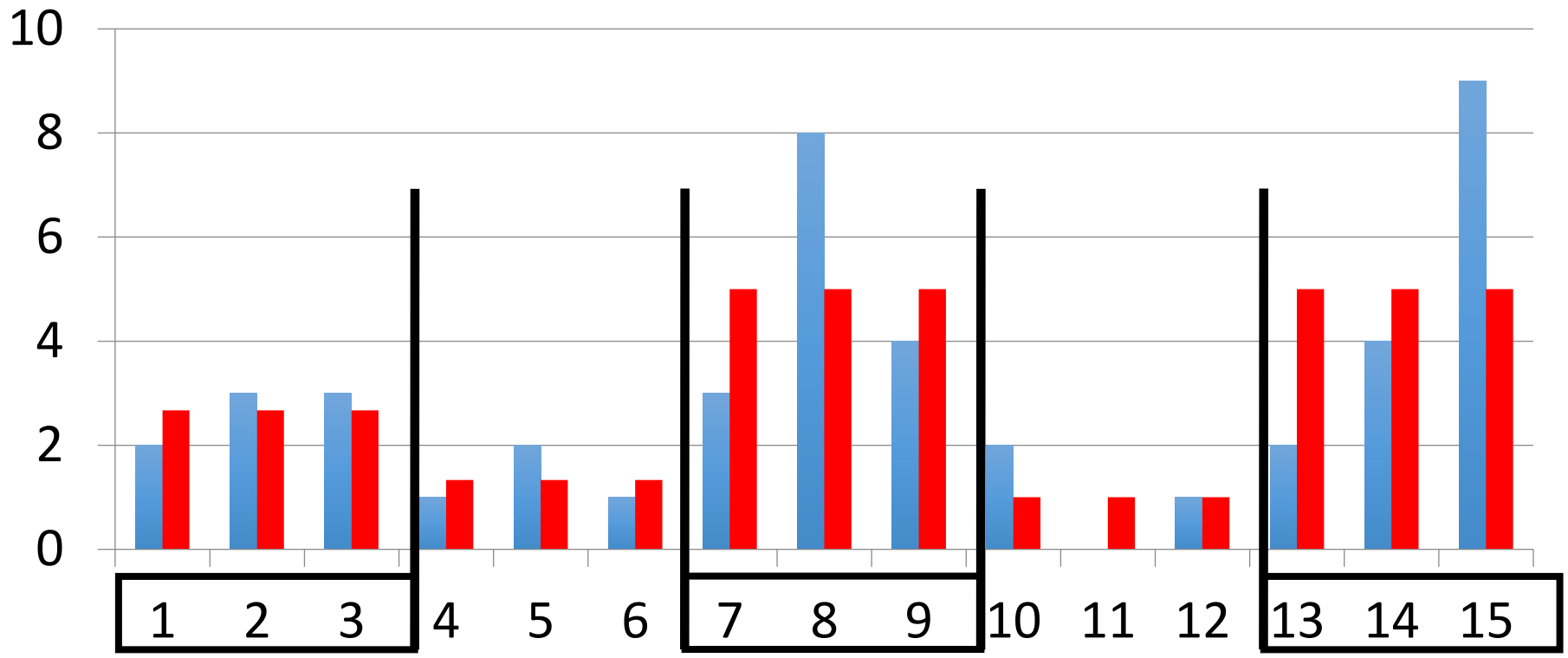
How much space do the uniform counts (bucket_size=ALL) take?

Fundamental Tradeoffs

- Want high resolution (like the full counts)
- Want low space (like uniform)
- Histograms are a compromise!

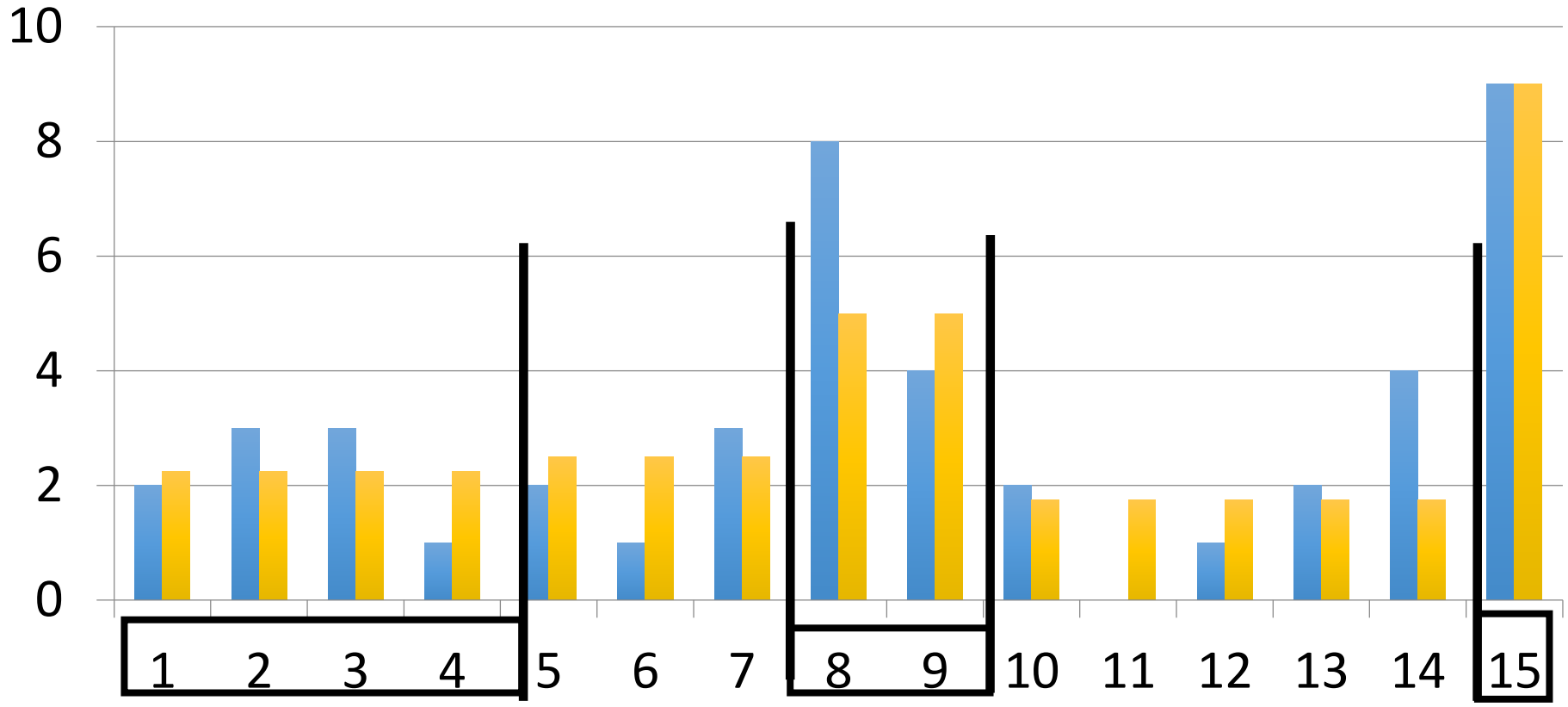
So how do we compute the “bucket” sizes?

Equi-width



All buckets roughly the same width

Equidepth



All buckets contain roughly the same number of items (total frequency). Able to adapt to skew.

Histograms

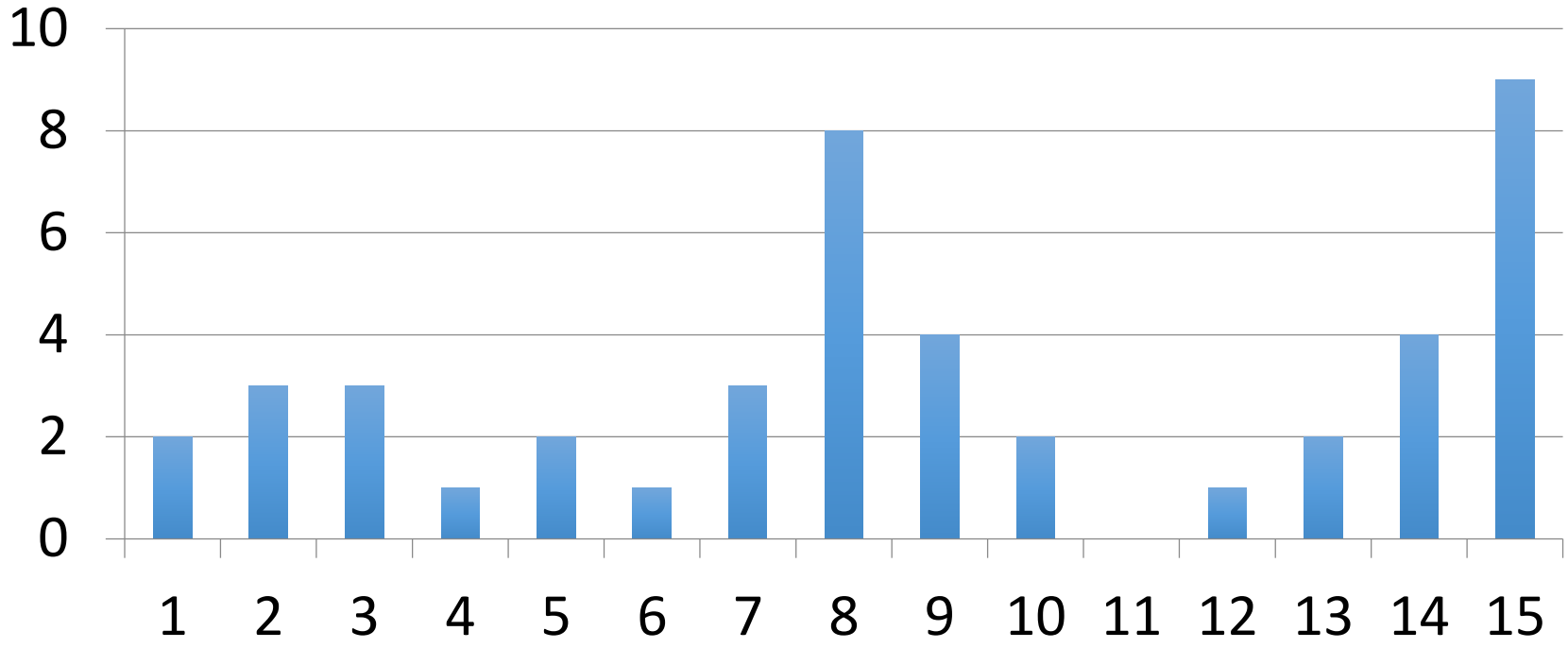
- Simple, intuitive and popular
- Parameters: # of buckets and type
- Can extend to many attributes (multidimensional)

Maintaining Histograms

- Histograms require that we update them!
  - Typically, you must run/schedule a command to update statistics on the database
  - Out of date histograms can be terrible!

- There is research work on self-tuning histograms and the use of query feedback
  - Oracle 11g

Nasty example



1. we insert many tuples with value > 16
2. we do **not** update the histogram
3. we ask for values > 20?

Precomputing Things

Views

- Views are relations, except that they are not physically stored.
- For presenting different information to different users
- Employee (ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
    SELECT name, project
    FROM  Employee
    WHERE department = "Development"
```

- Set privileges so that Payroll has access to Employee, others only to Developers

Example View Based on a Join

```
CREATE VIEW  Seattle-view  AS
      SELECT  buyer, seller, product, store
      FROM    Person, Purchase
      WHERE   Person.city = "Seattle"
                    AND Person.name = Purchase.buyer
```

We have a new virtual table:
Seattle-view (buyer, seller, product, store)

```
SELECT  name, store
FROM    Seattle-view, Product
WHERE   Seattle-view.product = Product.name
              AND Product.category = "shoes"
```

View is not Really a Table!

```
SELECT   name, Seattle-view.store
FROM     Seattle-view, Product
WHERE    Seattle-view.product = Product.name  AND
                  Product.category = "shoes"
```

This is what happens when you query a view

```
SELECT  name, Purchase.store
FROM    Person, Purchase, Product
WHERE   Person.city = "Seattle"    AND
                Person.name = Purchase.buyer  AND
                Purchase.product = Product.name  AND
                Product.category = "shoes"
```

Pros vs. Cons of Views

+ **Enforce Business Rules** – Use views to define business rules, such as when an items is active, or what is meant by “popular.”

+ **Consistency** – Simplify complicated query logic and calculations by hiding it behind the view’s definition.

+ **Security** – Restrict access to a table, yet allow users to access non-confidential data via views.

+ **Simplicity** – Databases with many tables possess complex relationships, which can be difficult to navigate if you aren’t comfortable using Joins.

− **Performance** – Each time a view is referenced, the query used to define it, is rerun.

− **Modifications** – Not all views support INSERT, UPDATE, or DELETE operations.

Materialized Views

- Unlike views, materialized views also store the results of the query in the database.

- Designed to improve the performance of the database by doing some intensive work in advance.
  - Can be used to pre-collect aggregate values
  - Assemble data that would come from many different tables, which would in turn require many different joins to be performed

Ordinary vs. Materialized Views

- Ordinary views
  - Virtual table
  - Named select statement
- Part of the SQL standard
- Syntax
  - CREATE VIEW *viewName* AS *selectStatement*

- Physical table
  - Replication of master data at a single point in time
- Not part of the SQL standard
- Syntax
  - CREATE **MATERIALIZED** VIEW *viewName* AS *selectStatement*

Why Use Materialized Views?

- Replicate data to non-master sites
  - To save network traffic when data is used in transactions
- Cache expensive queries
  - Expensive in terms of time or memory
  - Example: Sum, average or other calculations on large amounts of data

Types of Materialized Views

- Read-only
  - Insert, update or delete **NOT** allowed
- Updateable
  - Insert, update and delete on the view is allowed
  - Changes made to the view are pushed back to the master tables at refresh
- Writeable
  - Insert, update and delete on the view is allowed
  - Changes made to the view are **NOT** pushed back to the master tables at refresh

Refreshing a Materialized View

- Refresh types
  - Complete refresh
    - Recreates the materialized view
  - Fast (Incremental) refresh
    - Only changed data is refreshed
- Initiating a refresh
  - Scheduled refresh
  - On-demand refresh

Summary

- Database is doing lots of optimizations without you knowing
- Logical optimizations consider rewriting queries
- Physical optimizations uses additional structures to facilitate query evaluation

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).