CMPT 354: Database System I

Lecture 7. Basics of Query Optimization

Why should you care?

https://databricks.com/glossary/catalyst-optimizer

At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala's pattern matching and quasi quotes) in a novel way to build an extensible query optimizer.

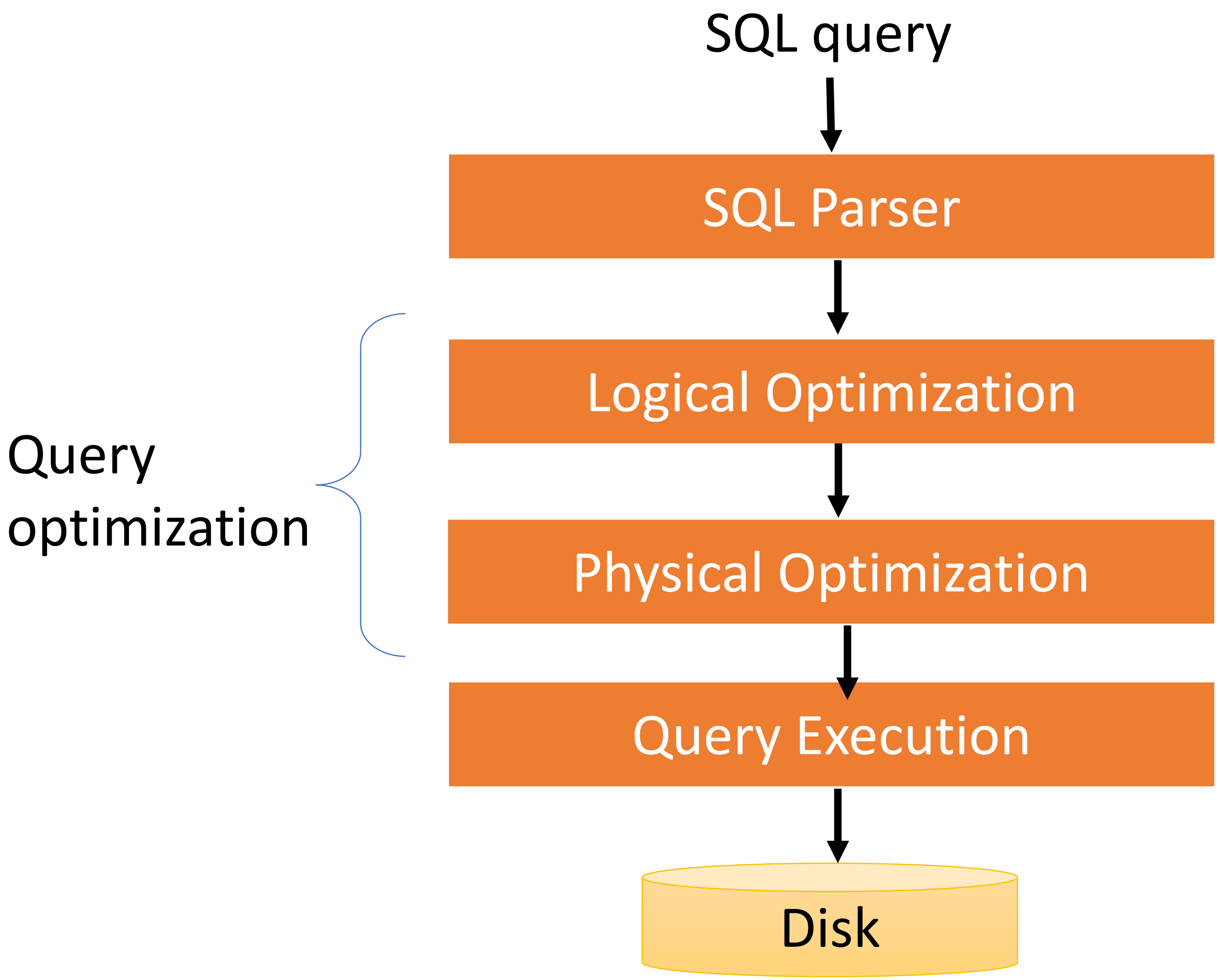
Goetz Graefe

2017 SIGMOD Edgar F. Codd Innovations Award

Professor Goetz Graefe is the recipient of the 2017 ACM SIGMOD Edgar F. Codd Innovations Award for his foundational contributions to the architecture and implementation of database query optimizers.

https://sigmod.org/sigmod-awards/people/goetz-graefe-2017-sigmod-edgar-f-codd-innovations-award/

Query Processing Steps



IBM System R Optimizer

- First implementation of a query optimizer
- Make people believe that the DBMS can beat a human developer
- A lot of the concepts are still used today

Access path selection in a relational database management system
https://dl.acm.org/citation.cfm?id=582099
by PG Selinger - 1979 - Cited by 2585 - Related articles
In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to **access paths**. This paper ...
Abstract · Authors · References · Cited By

How to build a query optimization?

1. Plan Space
   - Figure out all possible query plans

Too large, must be pruned

2. Cost Estimation
   - Estimate the cost of each plan

CPU + I/O

3. Search Algorithm
   - Find the best plan

don’t go for best plan, go for least worst plan

Outline

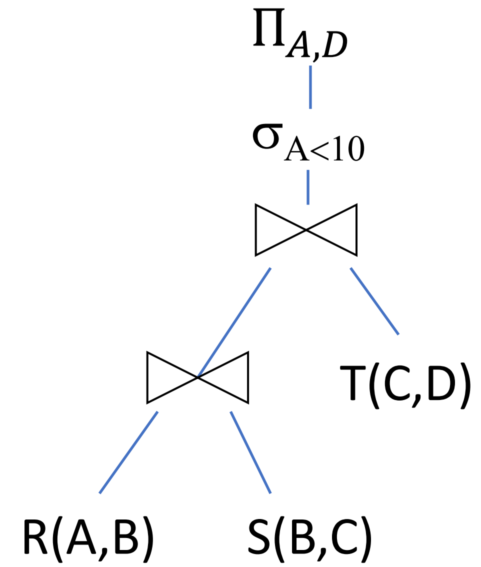
- **Recap of Logical Optimization**
  - Selection Pushdown
  - Projection Pushdown
- **Physical Optimization**
  - Join Algorithms
  - Selectivity Estimation

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;
```

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



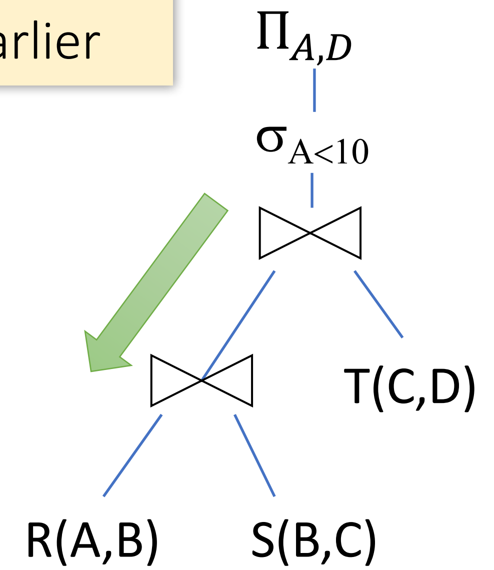
Optimizing RA Plan

```
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;
```

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

Push down selection on A so it occurs earlier



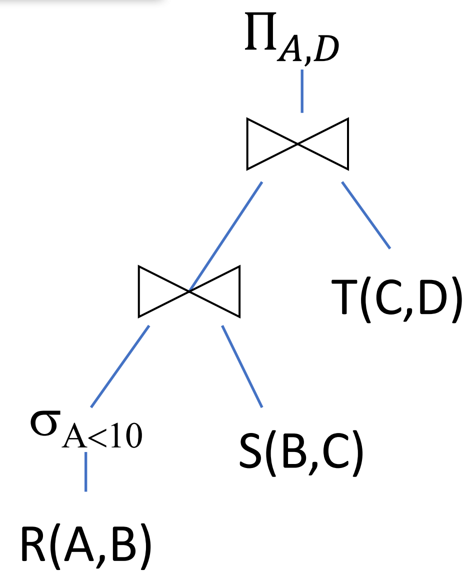
Optimizing RA Plan

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

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

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

Push down selection on A so it occurs earlier



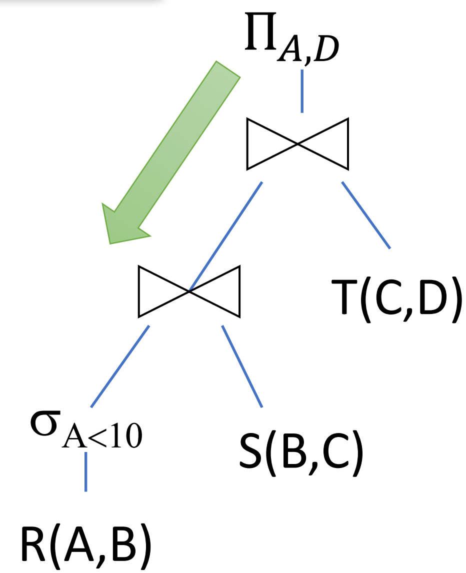
Optimizing RA Plan

```
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

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



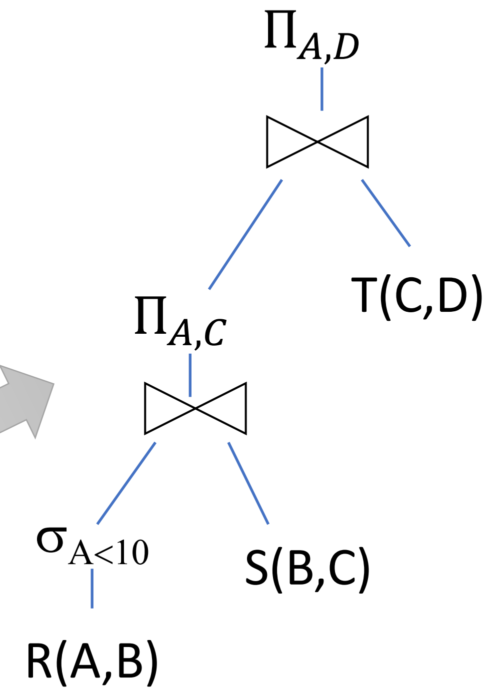
Optimizing RA Plan

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

We eliminate B earlier!

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

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



Outline

- **Recap of Logical Optimization**
  - Selection Pushdown
  - Projection Pushdown

- **Physical Optimization**
  - Join Algorithms
  - Histogram

Join Algorithms

- Nested loop Join
- Hash Join
- Sort-merge join

Student

sname	uID
Mike	0
Joe	1
Alice	0
Marry	1
Bob	0
Tim	2

University

uID	uname
0	SFU
1	UBC
2	UT

Student $\bowtie$ University

Dive into
Nested Loop Joins

Notes

- Cost = I/O + CPU + Network
- "IO aware" algorithms
  - **We will focus on I/O**
- Given a relation R, let:
  - T(R) = # of tuples in R
  - P(R) = # of pages in R

10	CMPT	345	SP 2018	Jiannan
20	CMPT	454	FA 2018	Martin

Page 1

30	…	…	…	…
40	…			

Page 2

50				
60				

Page 3

70				
80				

Page 4

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R)

1. **Loop over the tuples in R**

Note that our IO cost is based on the number of ***pages*** loaded, not the number of tuples!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. **For every tuple in R, loop over all the tuples in S**

Have to read ***all of S*** from disk for ***every tuple in R!***

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints… just check in the *if* statement!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S) + OUT

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Is this the same as a cross product?

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S) + OUT

What if R ("outer") and S ("inner") switched?

P(***S***) + T(***S***)*P(***R***) + OUT

Outer vs. inner selection could make a huge difference-
DBMS needs to know which relation is smaller!

IO-Aware Approach

Block Nested Loop Join (BNLJ)

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Given *B+1* pages of memory

Cost:

$P(R)$

1. **Load in B-1 pages of R at a time (leaving 1 page each free for S & output)**

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Given *B+1* pages of memory

Cost:

$$P(R) + \frac{P(R)}{B-1}P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. **For each (B-1)-page segment of R, load each page of S**

Note: Faster to iterate over the *smaller* relation first!

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions
4. **Write out**

BNLJ vs. NLJ: Benefits of IO Aware

- NLJ
  - Read all of S from disk for ***every page of R***
- BNLJ
  - Read all of S from disk for ***every (B-1)-page segment of R***

NLJ

$$P(R) + T(R)*P(S) + OUT$$

BNLJ

$$P(R) + \frac{P(R)}{B-1}P(S) + OUT$$

BNLJ is faster by roughly $\frac{(B-1)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
  - R: 500 pages
  - S: 1000 pages
  - 100 tuples / page
  - We have 12 pages of memory (B = 11)

Ignoring OUT here…

- NLJ: Cost = 500 + **50,000*1000** = **50 Million IOs**
- BNLJ: Cost = 500 + $\frac{500*1000}{10}$ = **50 *Thousand* IOs**

A very real difference from a small change in the algorithm!

Outline

- **Recap of Logical Optimization**
  - Selection Pushdown
  - Projection Pushdown

- **Physical Optimization**
  - Join Algorithms
  - Histogram

Motivation

- Your workload is

100000 queries

```
SELECT sID
FROM Student
WHERE  name = ?
```

100000 queries

```
SELECT sID
FROM Student
WHERE  gender = ?
```

Which one is better?

A. Index on name

B. Index on gender

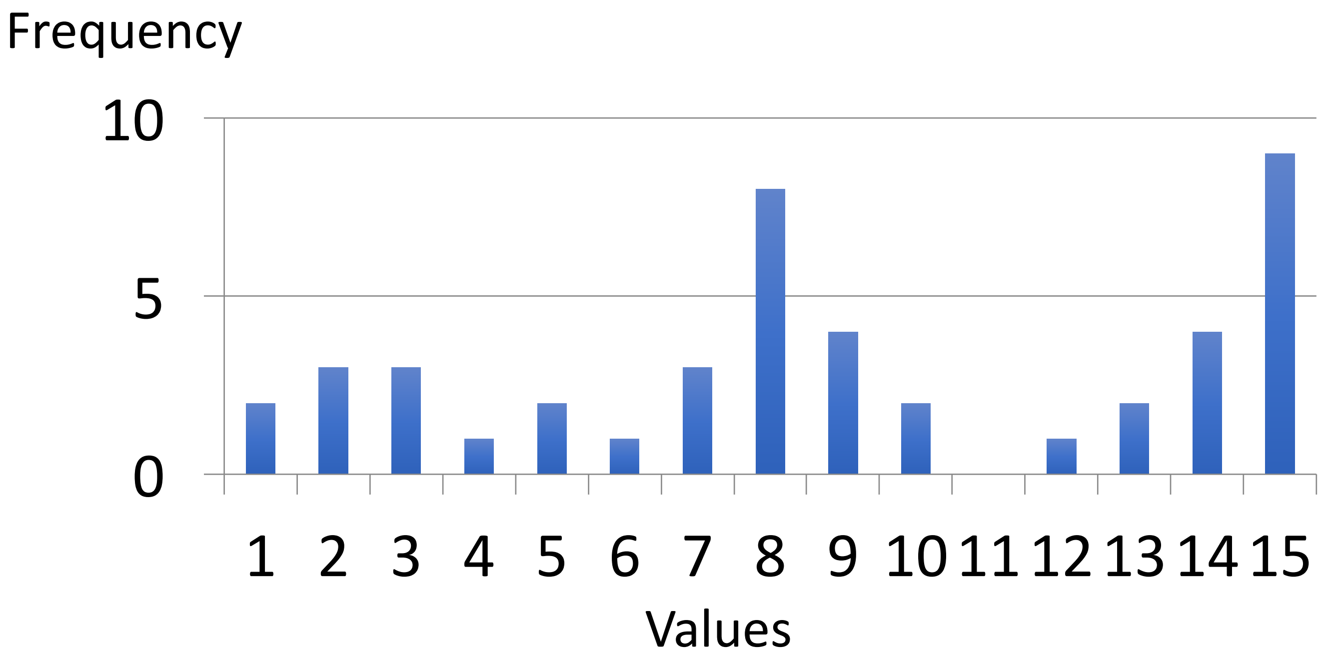
Histograms

- A histogram is a set of value ranges (“buckets”) and the frequencies of values in those buckets occurring

- How to choose the buckets?
    - Equiwidth & Equidepth

- Turns out high-frequency values are **very** important

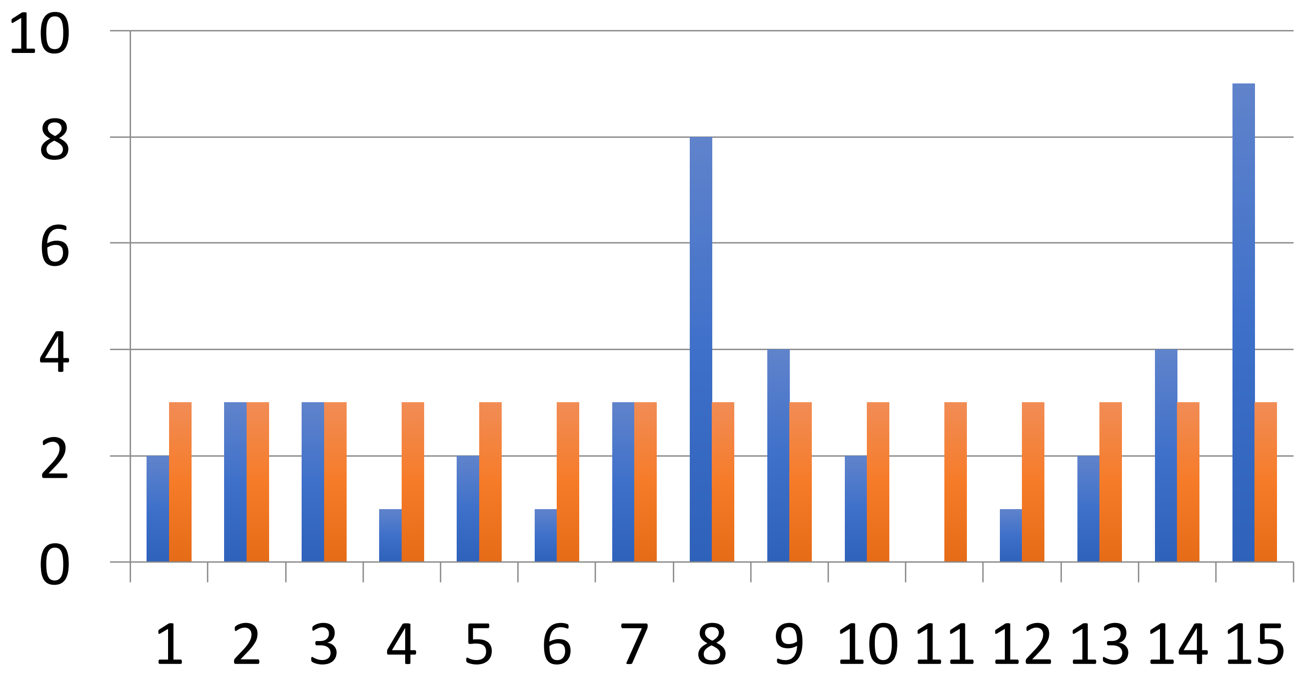
Example



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?

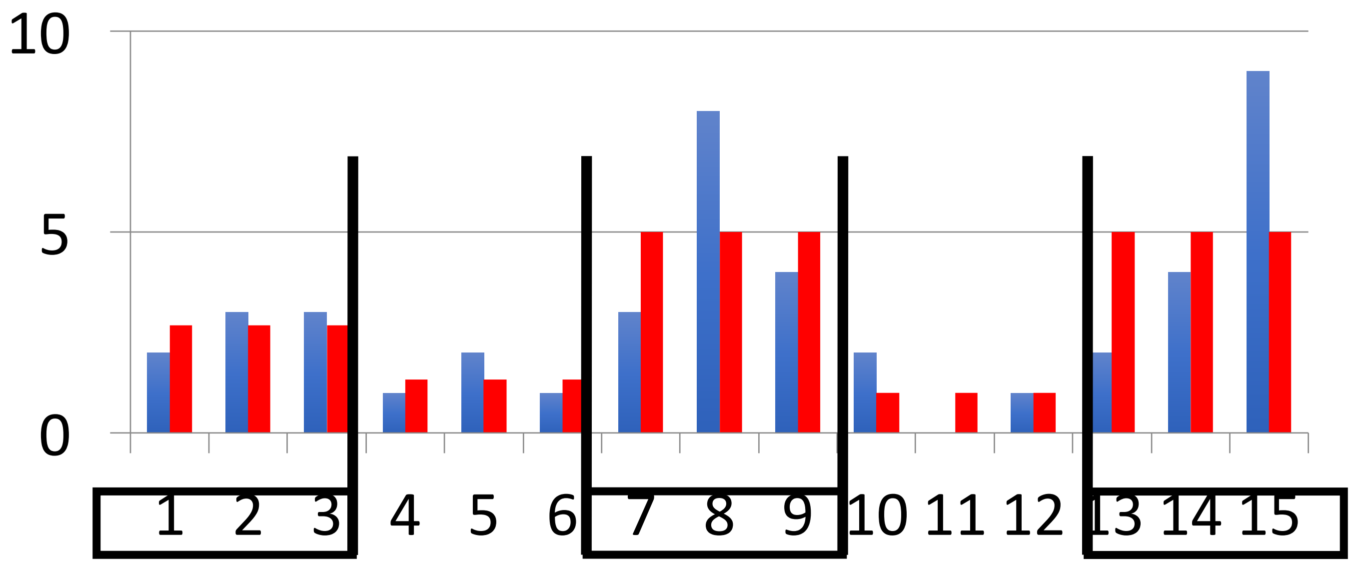
How much space do the uniform counts (bucket_size=ALL) take?

Fundamental Tradeoffs

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

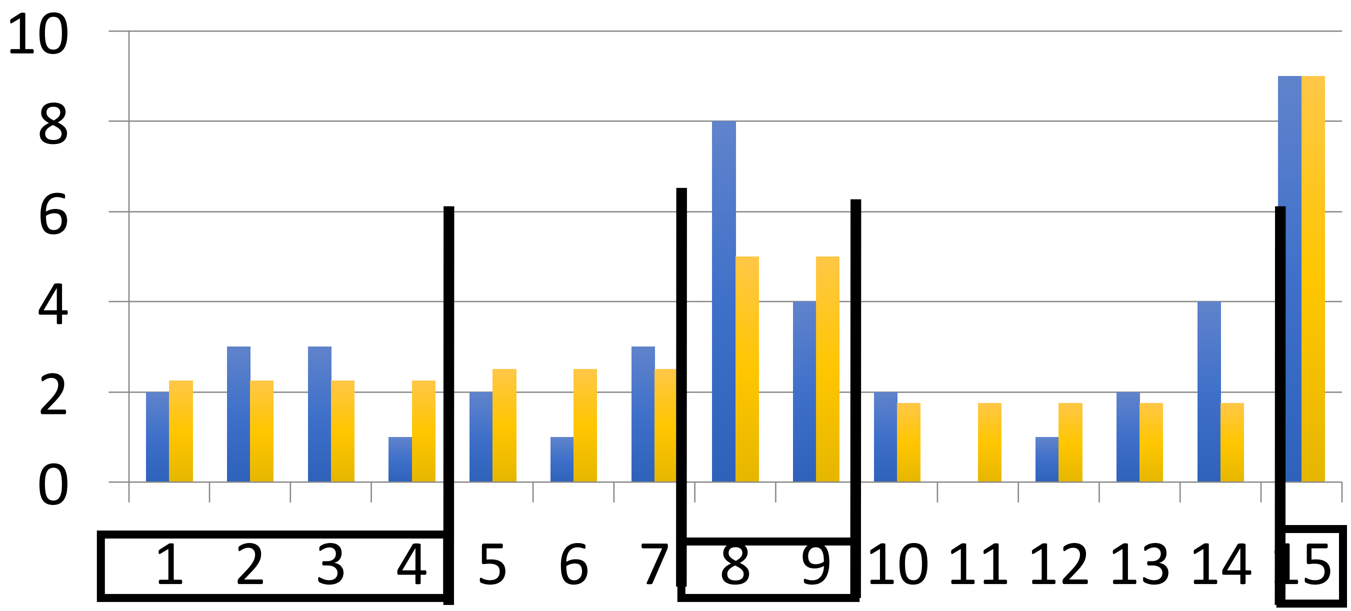
So how do we compute the “bucket” sizes?

Equi-width



All buckets roughly the same width

Equidepth



All buckets contain roughly the same number of items (total frequency)

Histograms

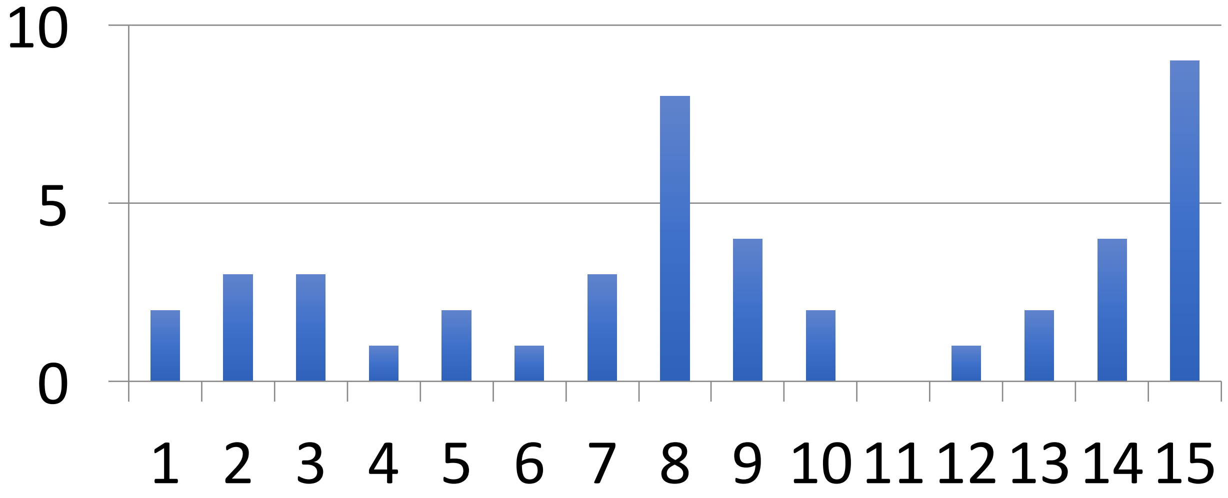
- Simple, intuitive and popular
- Parameters: # of buckets and type
- Can extend to many attributes (multidimensional)

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?

Compressed Histograms

- One popular approach:
  1. Store the most frequent values and their counts explicitly
  2. Keep an equiwidth or equidepth one for the rest of the values

People continue to try all manner of fanciness here
wavelets, graphical models, entropy models,…

Summary

- **Logical Optimization**
  - SQL -- > RA → RA Tree
  - Selection Pushdown
  - Projection Pushdown

- **Physical Optimization**
  - Nested Loop Join / Hash Join / Sort-Merge Join
  - I/O Aware Algorithm
  - Histogram

Acknowledge

- Some lecture slides were copied from or inspired by the following course materials
  - “W4111: Introduction to databases” by Eugene Wu at Columbia University
  - “CSE344: Introduction to Data Management” by Dan Suciu at University of Washington
  - “CMPT354: Database System I” by John Edgar at Simon Fraser University
  - “CS186: Introduction to Database Systems” by Joe Hellerstein at UC Berkeley
  - “CS145: Introduction to Databases” by Peter Bailis at Stanford
  - “CS 348: Introduction to Database Management” by Grant Weddell at University of Waterloo