6.5830 Lecture 9

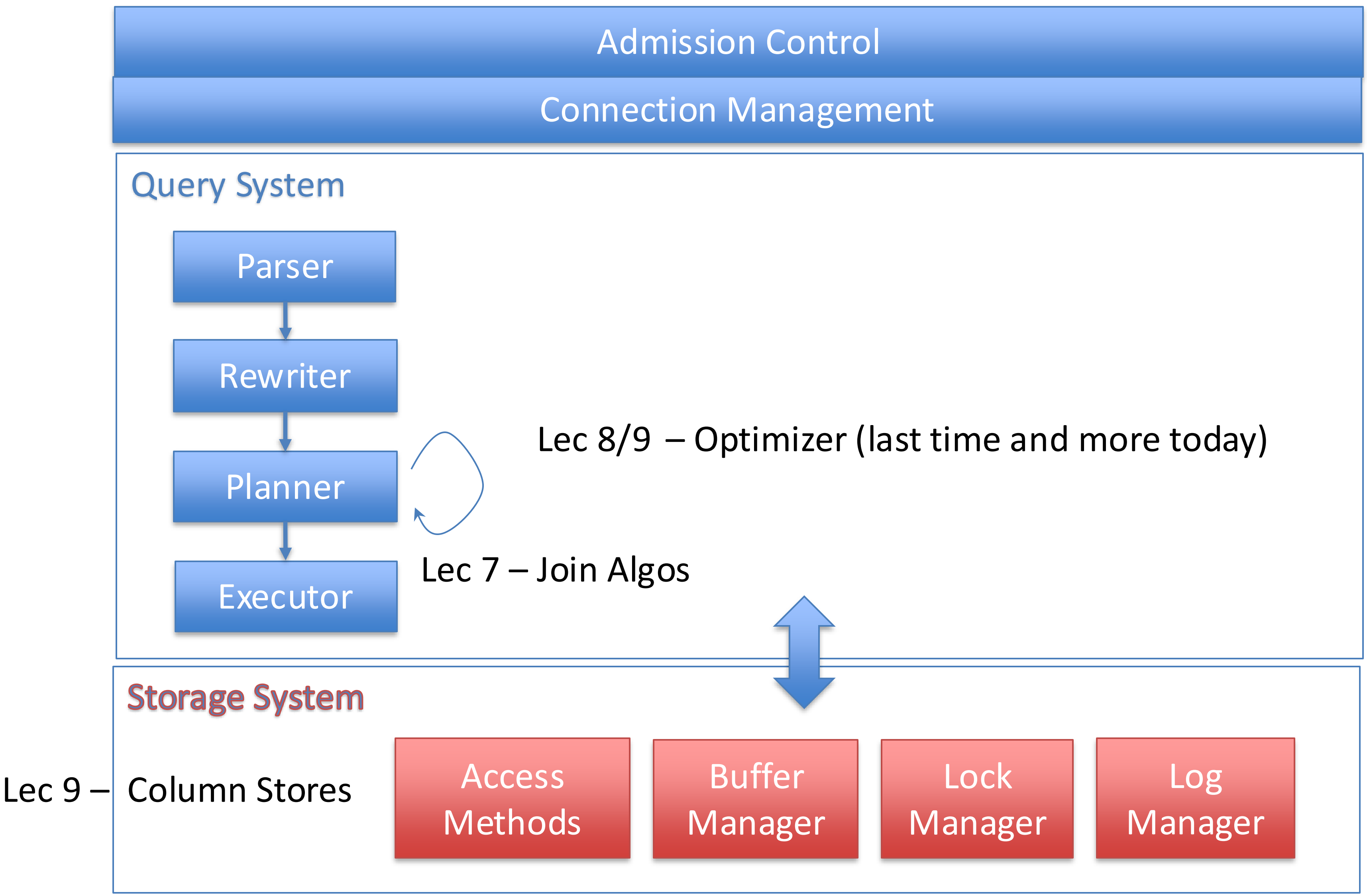
Column Stores

10/2/2024

Quiz 1 10/9

PS2 Due 10/7

Plan for Next Few Lectures



Recap - Join Algorithm

Algo	I/O cost	CPU cost	In Mem?
Nested loops	|R|+|S|	O({R}x{S})	R in mem
Nested loops	{S}|R| + |S|	O({R}x{S})	No
Index nested loops (R index)	|S| + {S}c (c <5)	O({S}log{R})	No
Block nested loops	|S| + B|R| (B=|S|/M)	O({R}x{S})	No
Sort-merge	|R|+|S|	O({S}log{S})	Both
Hash (Hash R)	|R|+|S|	O({S} + {R})	R in mem
Blocked hash (Hash S)	|S| + B|R| (B=|S|/M)	O({S} + B{R}) (*)	No
External Sort-merge	3(|R| + |S|)	O(P x {S}/P log {S}/P)	No
Simple hash (not covered)	P(|R|+|S|) (P=|S|/M)	O({R} + {S})	No
Grace hash	3(|R| + |S|)	O({R} + {S})	No

Grace hash is generally a safe bet, unless memory is close to size of tables, in which case simple can be preferable

Extra cost of sorting makes sort merge unattractive unless there is a way to access tables in sorted order (e.g., a clustered index), or a need to output data in sorted order (e.g., for a subsequent ORDER BY)

Recap Selinger Optimizer

Steps:
1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Selinger Statistics

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

Selectivity Estimates:

1. col = val

F = 1/ICARD() *(if index available)*
F = 1/10 otherwise

2. col > val

(max key - value) / (max key - min key) *(if index available)*
1/3 otherwise

3. col1 = col2

1/MAX(ICARD(PK table)) *(if index available)*
1/10 otherwise

P1 and P2:

F(P1) x F(P2)

P1 or P2

1 – P(neither predicate is satisfied) =
1 – (1-F(P1)) x (1-F(P2))

Note uniformity assumption

http://clicker.mit.edu/6.5830

Selinger Statistics

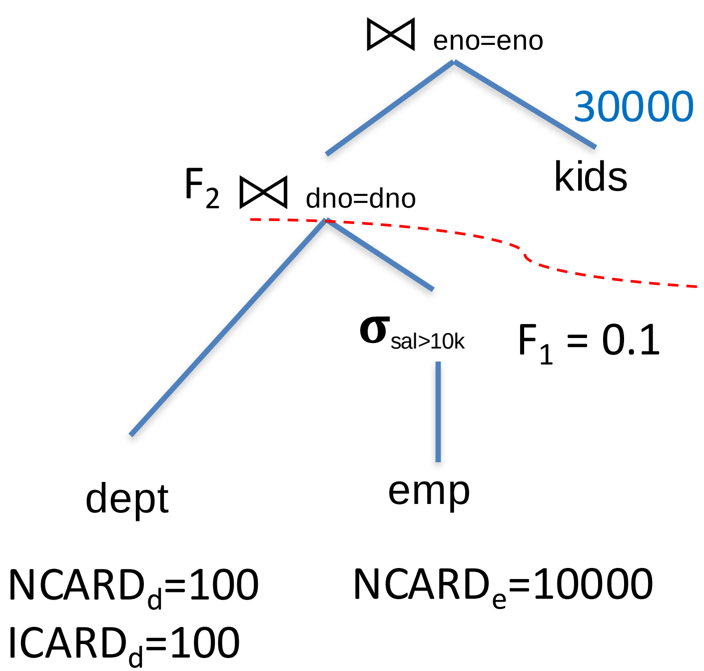
NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes



Clicker (http://clicker.mit.edu/6.5830)

What is the selectivity of F_2

A) 1

B) 0.1

C) 0.01

D) 0.001

Clicker - Intermediate Sizes

http://clicker.mit.edu/6.5830

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

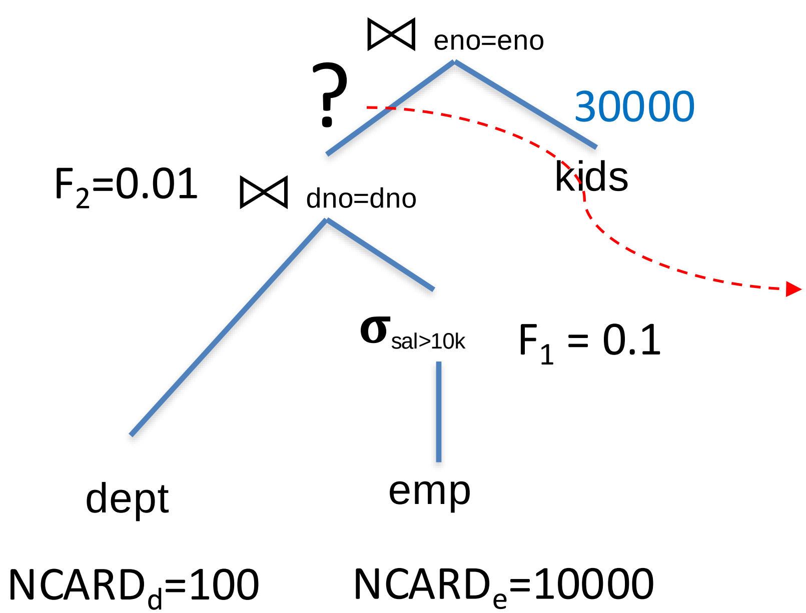
NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes



What is the intermediate size after the Dep-Emp Join?

A) $100 \times (10000 \times 0.1) \times 0.01 = 1000$

B) $10000 \times 0.1 = 1000$

C) $10000 \times 0.1 \times 0.01 = 10$

D) $10000 \times 0.1 \times 100 = 100000$

Intermediate Sizes

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

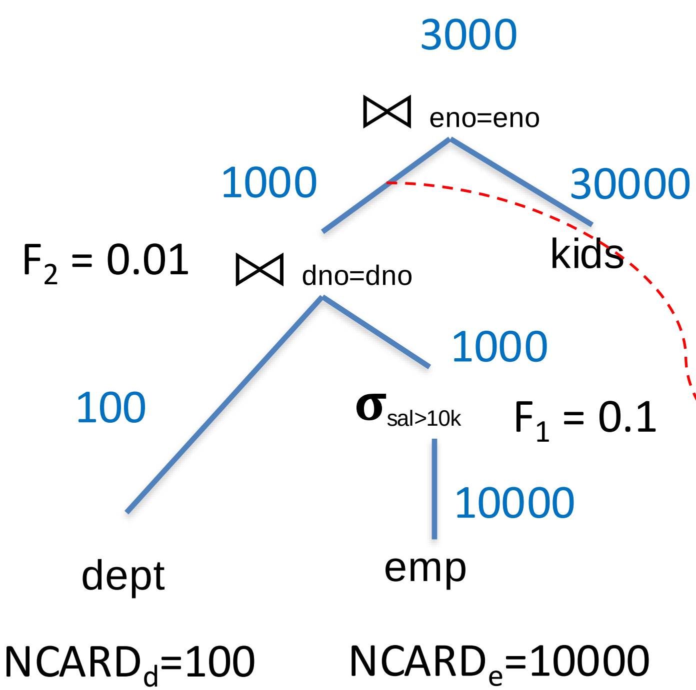
NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes



$$NCARD_d \times NCARD_e \times F_1 \times F_2 = 100 \times 10000 \times 0.1 \times 0.01 = 1000$$

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Base Table Operations

Cost = pages read +

weight x (records evaluated)

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

W: weight of CPU operations

Equality predicate with unique index: 1 + 1 + W

B+Tree lookup

Heap File lookup

Predicate evaluation

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Base Table Operations

Cost = pages read + weight x (records evaluated)

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

W: weight of CPU operations

Equality predicate with unique index: $\underbrace{1}_{\text{B+Tree lookup}} + \overset{\text{Heap File lookup}}{1} + \underbrace{W}_{\text{Predicate evaluation}}$

Clustered index, range w/ selectivity F

A: F x TCARD + W x (tuples read)
B: F x (NINDX + NCARD) + W x (tuples read)
C: F x NINDX + W x (tuples read)
D: F x (NINDX + TCARD) + W x (tuples read)

Clicker (http://clicker.mit.edu/6.5830)

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Base Table Operations

Cost = pages read +

weight x (records evaluated)

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

W: weight of CPU operations

Equality predicate with unique index: $\underbrace{1}_{\text{B+Tree lookup}} + \overbrace{1}^{\text{Heap File lookup}} + \underbrace{W}_{\text{Predicate evaluation}}$

Clustered index, range w/ selectivity F: F x (NINDX + TCARD) + W x (tuples read)

One I/O per page

Unclustered index, range w/ selectivity F : F x (NINDX + NCARD) + W x (tuples read)

One I/O per record

Seq (segment) scan: TCARD + W x (NCARD)

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Joins

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

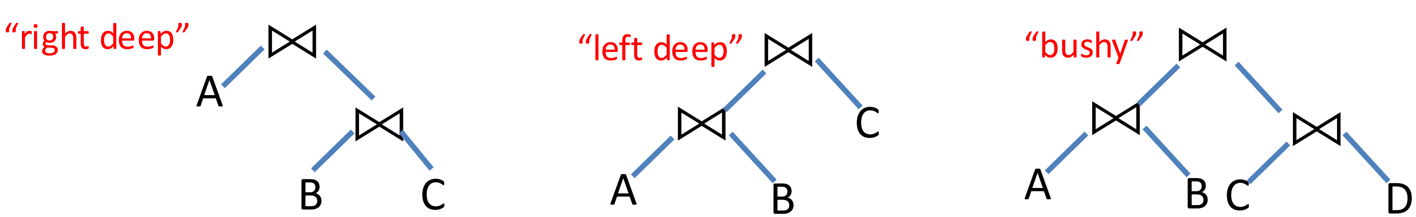
W: weight of CPU operations

NestedLoops(A,B,pred)

Cost(A) + NCARD(A) x Cost(B)

Outer Plan | Inner Plan

- Selinger only considers “left deep” plans, i.e., B is always a base table T_{right}
- In an index on T_{right}, Cost(B) = 1 + 1 + W
- If no index, Cost(B) = TCARD(T_{right}) + W x NCARD(T_{right})
- Cost(A) is just cost of outer subtree



Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Joins

Merge(A,B,pred)

Cost(A) + Cost(B) + sort cost

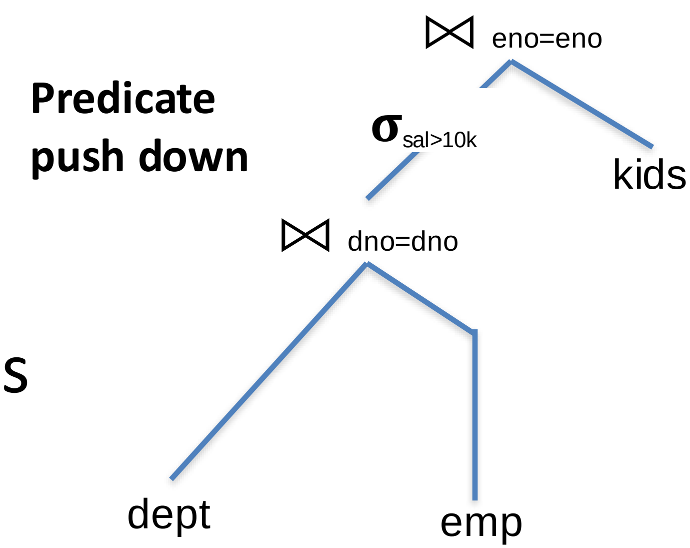
Varies depending on whether sort is in memory or on disk, and whether one or both tables are already sorted

If either table is a base table, cost is just the sequential scan cost

Steps:
1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Enumerating Plans

- Selinger combines several heuristics with a search over join orders
- Heuristics
  - Push down selections
  - Don't consider cross products
  - Only "left deep" plans
    - Right side of all joins is base relation
- Still have to order joins!



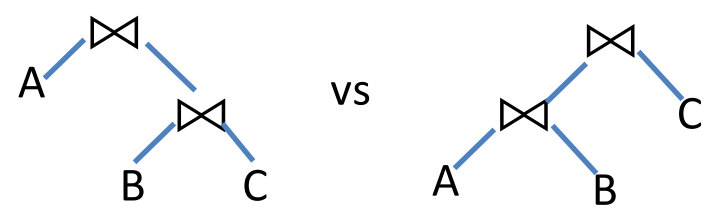
Join ordering

(not even factoring in choice of join method)

- Suppose I have 3 tables, A ⋈ B ⋈ C
  – Predicates between all 3 (no cross products)
- How many orderings?

ABC	A(BC)	(AB)C
ACB	A(CB)	(AC)B
BAC	B(AC)	(BA)C
BCA	B(CA)	(BC)A
CAB	C(AB)	(CA)B
CBA	C(BA)	(CB)A

n!



This plan is not left deep!

Left deep plans are all of the form (…(((AB)C)D)E)…)

n! left deep plans
10! = 3.6 M
15! = 1.3 T

Can we do better?

Dynamic Programming Algorithm

- **Idea**: compute the best way to join each sub-plan, from smallest to largest
  - Don't need to reconsider subplans in larger plans
- For example, if the best way to join ABC is (AC)B, that will always be the best way to join ABC, whenever* these three relations occur as a part of a subplan.

* *Except when considering interesting orders*

Postgres example

*explain select * from emp join kids using (eno);*

Hash Join (cost=34730.02..**132722.07** rows=3000001 width=35)

Hash Cond: (kids.eno = emp.eno)

-> Seq Scan on kids (cost=0.00..**49099.01** rows=3000001 width=18)

-> Hash (cost=16370.01..**16370.01** rows=1000001 width=21)

-> Seq Scan on emp (cost=0.00..**16370.01** rows=1000001 width=21)

Default PostgreSQL valueS:

- single sequential page read costing 1.0 units (seq_page_cost)
- Each row processed adds 0.01 (cpu_tuple_cost),
- each non-sequential page read adds 4.0 (random_page_cost).
- … ///there are many many more constants like this

First number is startup cost (i.e., cost to fetch the first row)
Second number is total cost

Postgres example

*explain select * from emp join kids using (eno);*

```
Hash Join  (cost=34730.02..132722.07 rows=3000001 width=35)
 Hash Cond: (kids.eno = emp.eno)
 ->  Seq Scan on kids  (cost=0.00..49099.01 rows=3000001 width=18)
 ->  Hash  (cost=16370.01..16370.01 rows=1000001 width=21)
       ->  Seq Scan on emp  (cost=0.00..16370.01 rows=1000001 width=21)
```

*explain select * from dept join emp using(dno) join kids using (eno);*

```
Hash Join  (cost=35000.04..140870.43 rows=3000001 width=39)
 Hash Cond: (emp.dno = dept.dno)
 ->  Hash Join  (cost=34730.02..132722.07 rows=3000001 width=35)
       Hash Cond: (kids.eno = emp.eno)
       ->  Seq Scan on kids  (cost=0.00..49099.01 rows=3000001 width=18)
       ->  Hash  (cost=16370.01..16370.01 rows=1000001 width=21)
             ->  Seq Scan on emp  (cost=0.00..16370.01 rows=1000001 width=21)
 ->  Hash  (cost=145.01..145.01 rows=10001 width=8)
       ->  Seq Scan on dept  (cost=0.00..145.01 rows=10001 width=8)
```

Identical subplans

Selinger Algorithm

1. Find all plans for accessing each base relation
   - Include index scans when available on push-down predicates
2. For each relation, save cheapest unordered plan (, and cheapest plan for each "interesting order".) Discard all others.
3. Now, try all ways of joining all pairs of 1-table plans saved so far. Save cheapest unordered 2-table plans (and cheapest "interesting ordered" 2-table plans)
4. Now try all ways of combining a 2-table plan with a 1-table plan. Save cheapest unordered (and interestingly ordered 3-way plans). You can now throw away the 2-way plans.
4. Continue combining *k*-way and 1-way plans until you have a collection of full plan trees
5. At top, satisfy GROUP BY and ORDER BY either by using interestingly ordered plan, or by adding a sort node to unordered plan, whichever is cheapest.

don't combine a *k*-way plan with a 1-way plan if there's no predicate between them, unless all predicates have been used up (i.e. postpone Cartesian products)

Selinger Algorithm

don't combine a *k*-way plan with a 1-way plan if there's no predicate between them, unless all predicates have been used up (i.e. postpone Cartesian products)

R $\leftarrow$ set of relations to join

For i in {1...|R|}:

    for S in {all length i subsets of R}:

        $optcost_S = \infty$

        $optjoin_S = \emptyset$

        for a in S: //a is a relation

            c_{sa} = $\boxed{optcost_{s-a} +}$ ***Cached in previous step!***

                min. cost to join (S-a) to a +

                min. access cost for a

            if $c_{sa} < optcost_s$:

                $optcost_s = c_{sa}$

                $optjoin_s$ = optjoin(S-a) joined optimally w/ a

Example

4 Relations: ABCD

Optjoin:

A = best way to access A

(e.g., sequential scan,

or predicate pushdown into index...)

B = "   "   "   " B

C = "   "   "   " C

D = "   "   "   " D

{A,B} = AB or BA

{A,C} = AC or CA

{B,C} = BC or CB

{A,D}

{B,D}

{C,D}

Relations	Best Plan	Cost

Dynamic Programming Table

Example (con't)

Optjoin

Already computed!

{A,B,C} = compare ({B,C})A to ({A,C})B to ({A,B})C

{A,B,D} = …

{B,C,D} = …

…

{A,B,C,D} = compare ({B,C,D})A to ({A,C,D})B to ({A,B,D})C to ({A,B,C})D

Relations	Best Plan	Cost
A	Index Scan	5
B	Seq Scan	15
…		
{A,B}	BA	75
{A,C}	AC	12
{B,C}	CB	22
..		

Complexity (cont.)

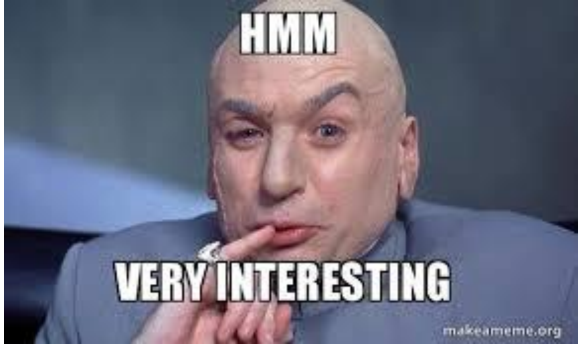
2^n Subsets

How much work per subset?

- Have to iterate through each element of each subset, so this at most n

$n2^n$ complexity (vs n!)

n=12 → 49K vs 479M

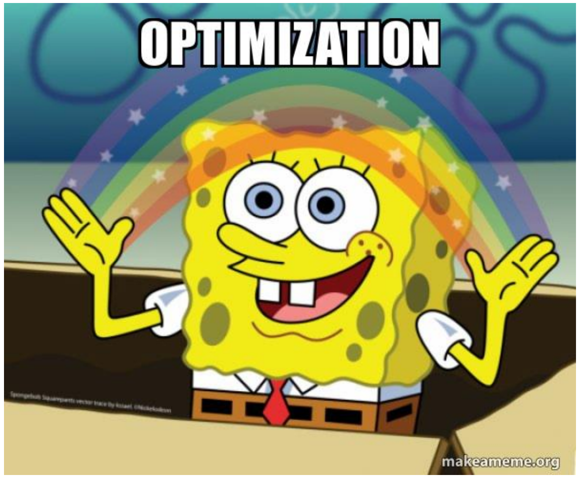


Interesting Orders

- Some query plans produce data in sorted order – E.g scan over a primary index, merge-join
  – Called an *interesting order*
- Next operator may use this order – E.g. can be another merge-join
- For each subset of relations, compute multiple optimal plans, one for each interesting order
- Increases complexity by factor k+1, where k=number of interesting orders

Optimization Recap

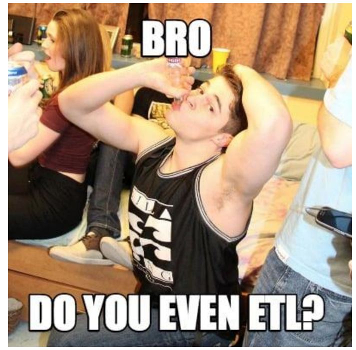
- Selinger Optimizer is the foundation of modern cost-based optimizers
  - Simple statistics
  - Several heuristics, e.g., left-deep
  - Dynamic programming algo for join ordering
- Easy to extend, e.g., with:
  - More sophisticated statistics
  - Fewer heuristics



Rest of today: Column Stores

A different way to build a database system

Typical Database Setup



“Extract, Transform, Load”

Transactional database
Lots of writes/updates
Reads of individual records

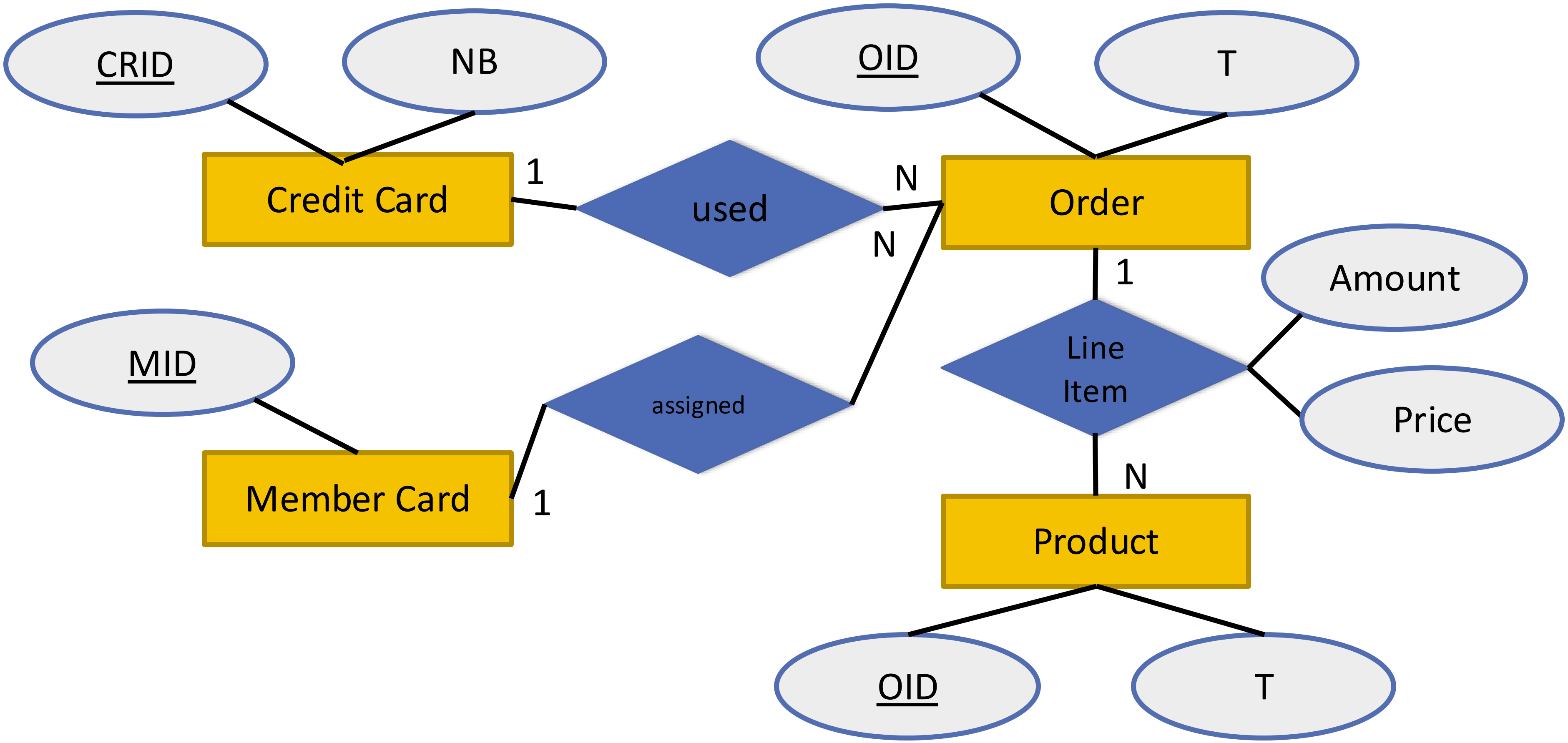
Analytics / Reporting Database
“Warehouse”
Lots of reads of many records
Bulk updates
Typical query touches a few columns

PROBLEM

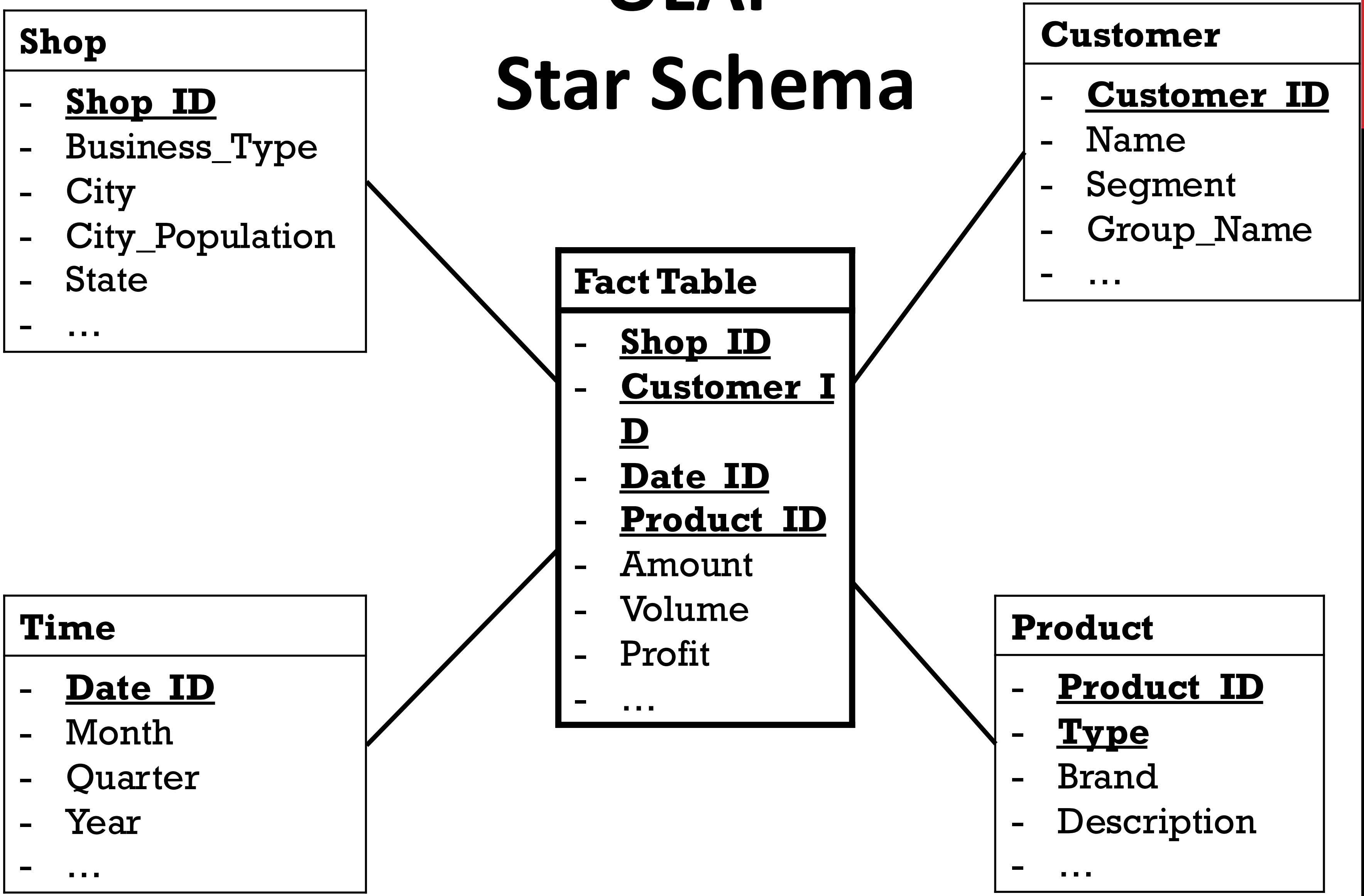
- **You are the new Data Scientist at New Market**
- New Market is tracking all customer purchases with their membership card or credit card
- They also have data about their customers (estimated income, family status,…)
- Recently, they are trying to improve their image for young mothers
- As a start they want to know the following information for mothers under 30 for 2013:
  - How much do they spend?
  - How much do they spend per state?
  - How does this compare to all customers under 30?
  - What are their favorite products?
  - How much do they spend per year?

Your first project: Design the schema for New Market!

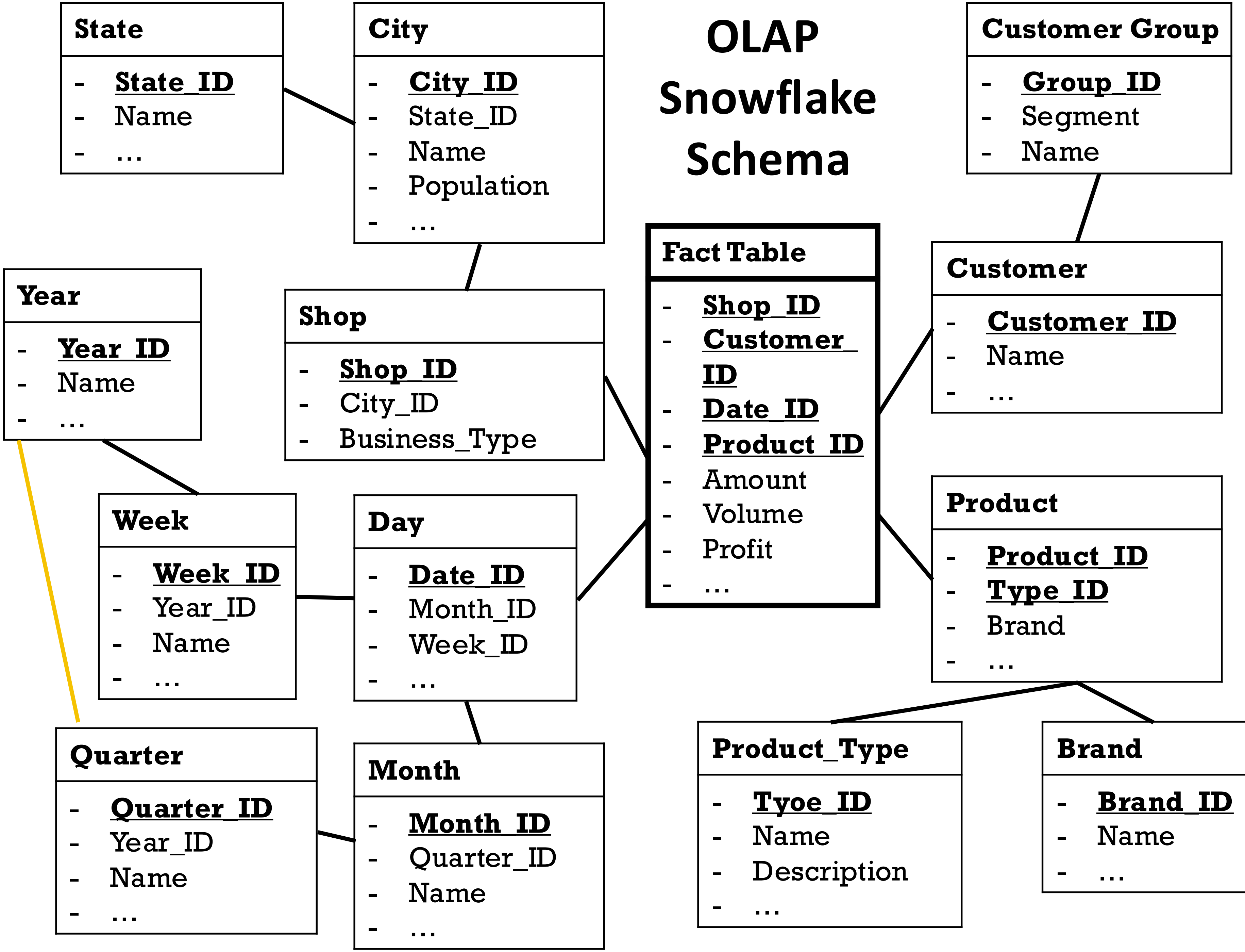
TYPICAL OLTP SCHEMA



OLAP
Star Schema



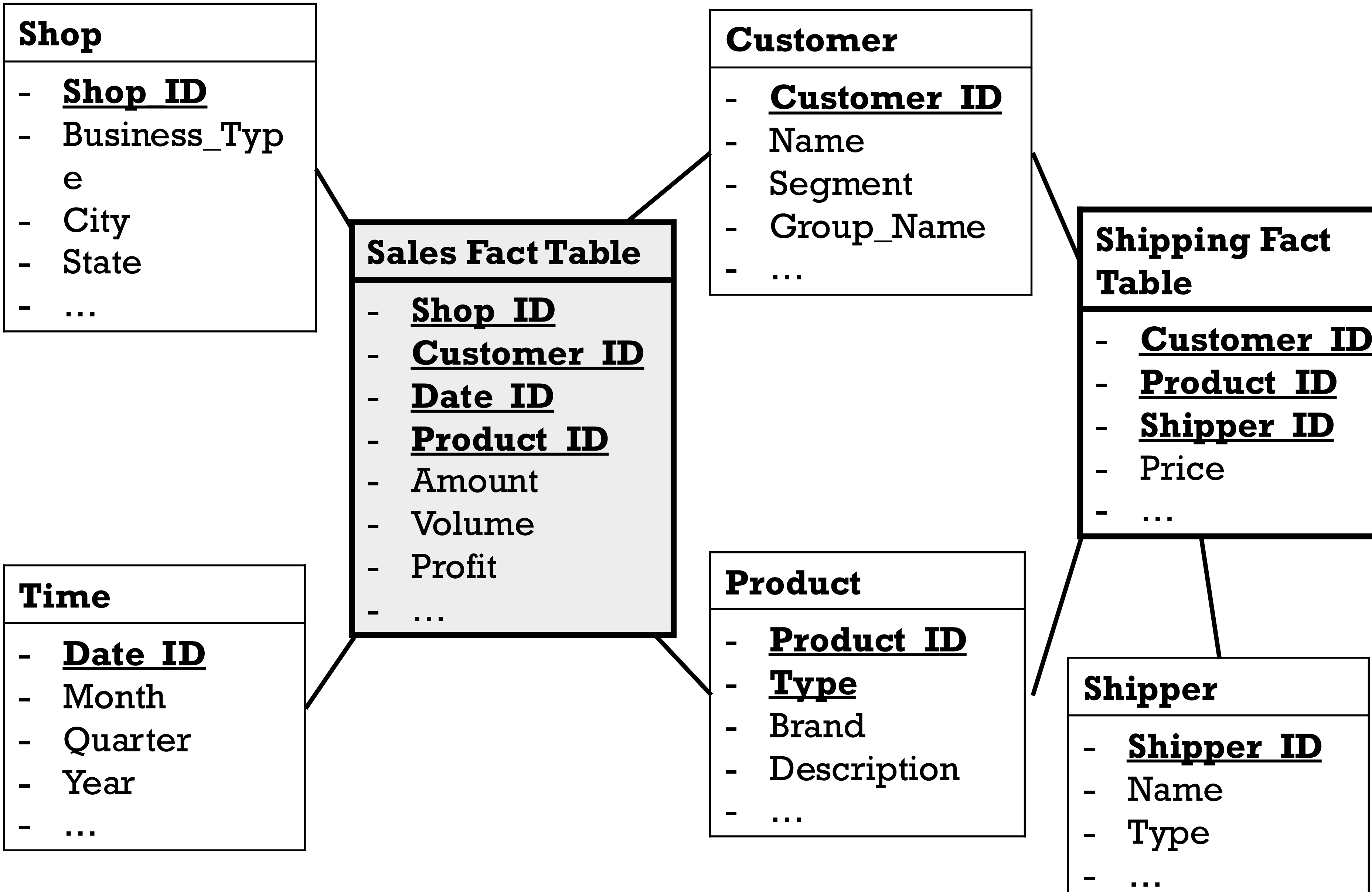
OLAP Snowflake Schema



STAR VS. SNOWFLAKE SCHEMA

	Snowflake	Star
Normalization/ De-Normalization	Dimension Tables are in Normalized form but Fact Table is still in De-Normalized form	Both Dimension and Fact Tables are in De-Normalized form
Space	Smaller	Bigger (**Redundancy**)
Query Performance	More Joins → slower	Fewer Joins → faster
Ease of Use	Complex Queries	Pretty Simply Queries
When to use	When dimension table is relatively big in size, snowflaking is better as it reduces space.	When dimension table contains less number of rows, we can go for Star schema.

Galaxy / Fact Constellation
Schema



2 DIMENSIONAL CASE

Dimensions

Date:month
June→Augu
2006

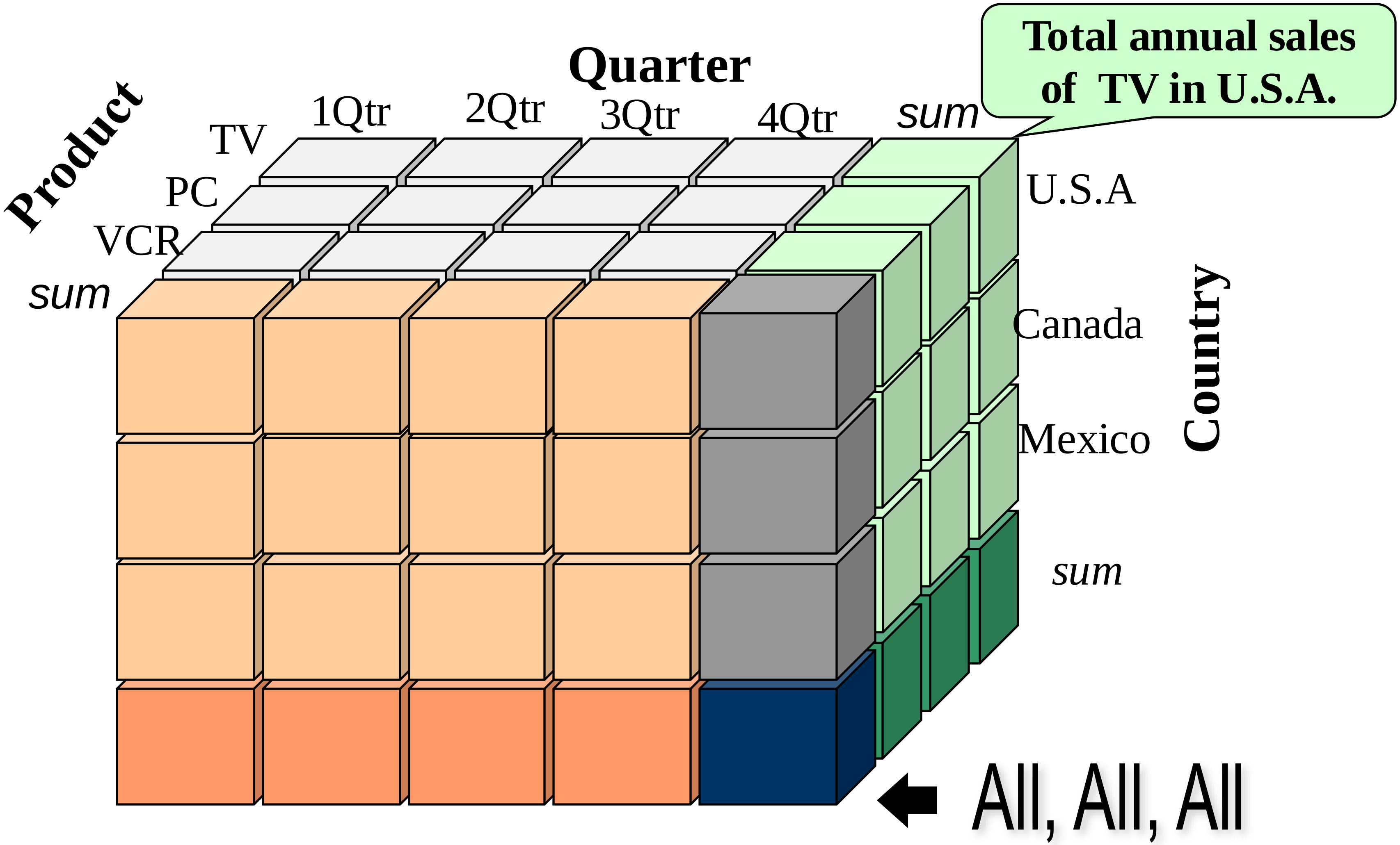
	Blue	Red	Orange	Total
June	51	25	158	234
July	58	20	120	198
August	65	22	51	138
Total	174	67	329	570

Data Points/
1st level of aggregation

Aggregated
w.r.t. X-dim

Aggregated
w.r.t. Y-dim

Aggregated
w.r.t. X and Y



Total annual sales of TV in U.S.A.
Quarter
Product
1Qtr
2Qtr
3Qtr
4Qtr
sum
TV
PC
VCR
sum
U.S.A
Canada
Mexico
sum
Country
All, All, All

TYPICAL OLAP OPERATIONS

Roll up (drill-up): summarize data

by climbing up hierarchy or by dimension reduction

Drill down (roll down): reverse of roll-up

from higher level summary to lower level summary or detailed data, or introducing new dimensions

Slice and dice: *project and select*

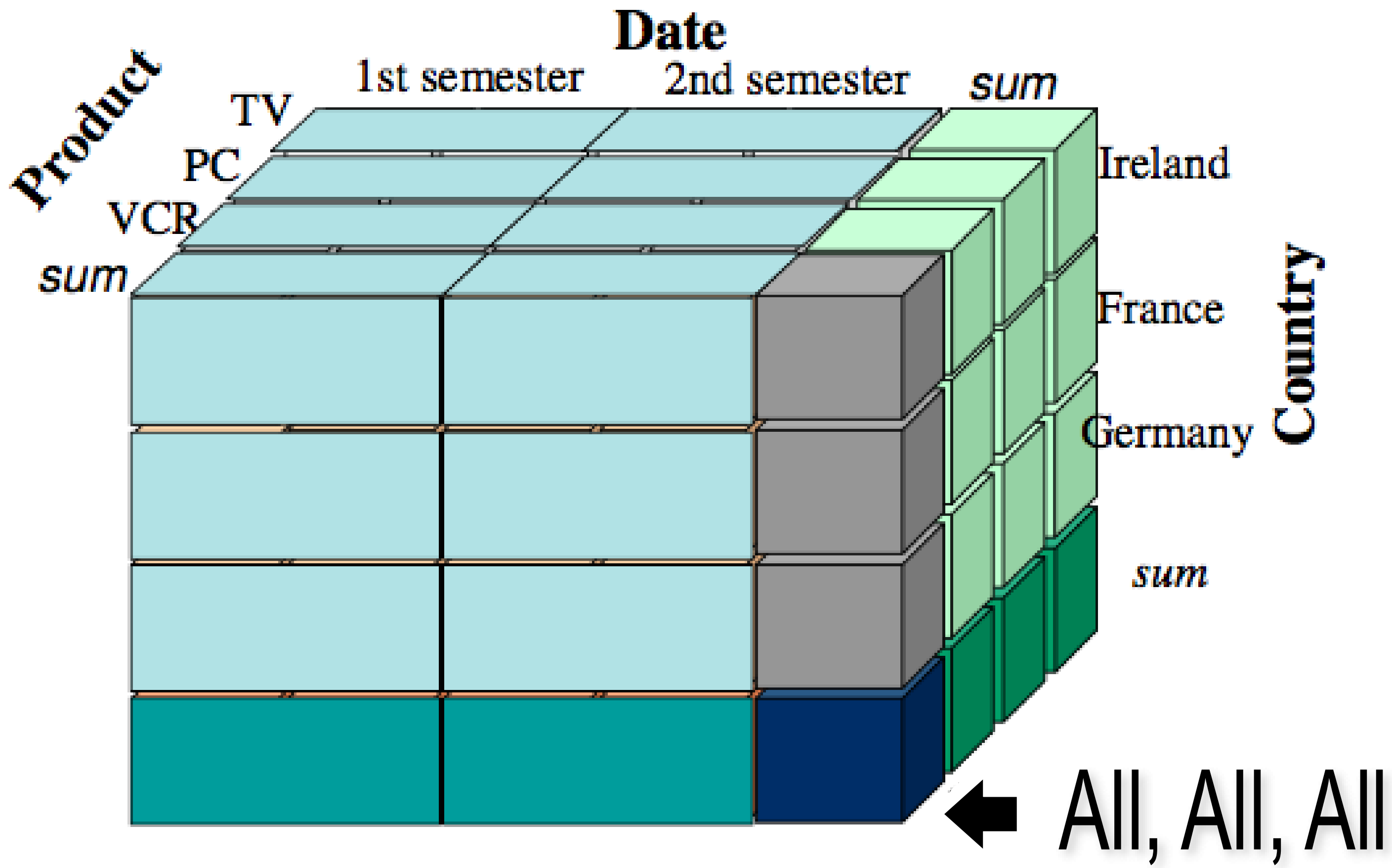
Pivot (rotate): *reorient the cube, visualization, 3D to series of 2D planes.*

Other operations

drill across: involving (across) more than one fact table

drill through: through the bottom level of the cube to its back-end relational tables (using SQL)

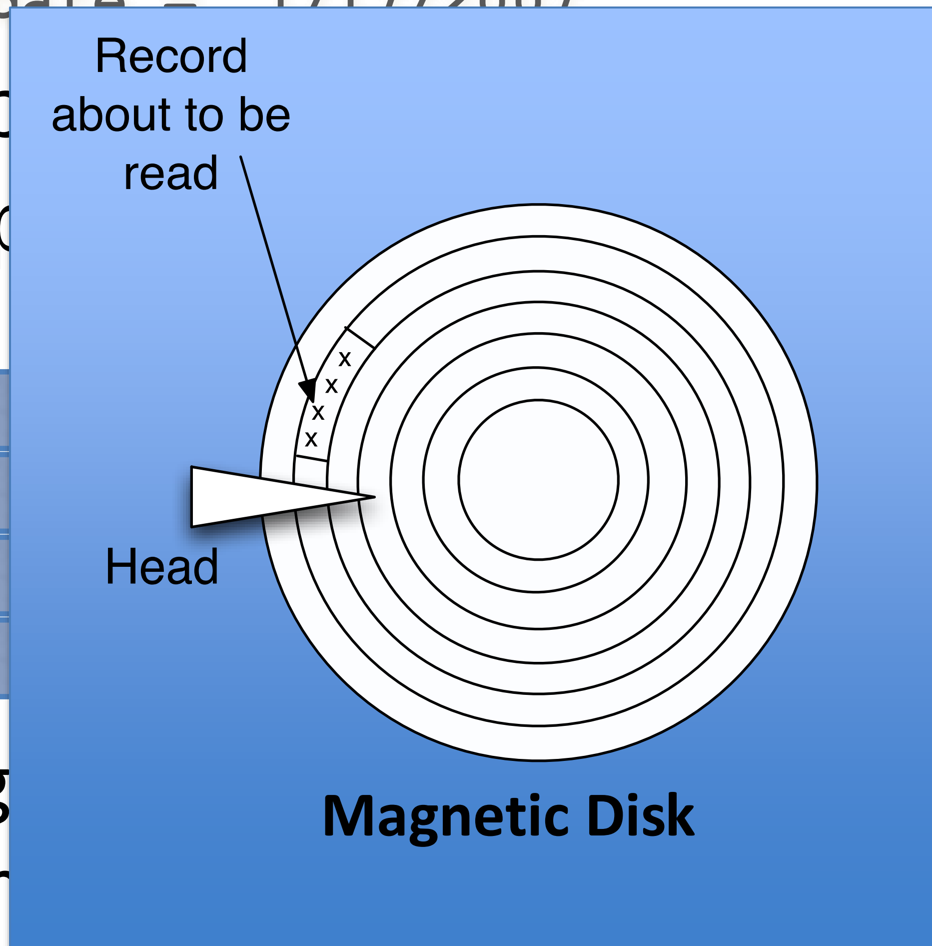
ROLLUP


Date
1st semester
2nd semester
sum
Product
TV
PC
VCR
sum
Ireland
France
Germany
sum
Country
All, All, All

How Long Does a Scan Take?

```
SELECT avg(price) FROM tickstore  WHERE symbol =
'GM' and date = '1/17/2007'
```

- Time p … data read
- Examp



…change	date
…YSE	1/17/2007
…YSE	1/17/2007
…YSE	1/17/2007
…QDS	1/17/2007

Even thoug… and *symbol*, if data is or… lumns

Memory and SSD also transfer a block at a time, so same issue arises.

Column Representation Reduces Scan Time

- Idea: Store each column in a separate file

Column Representation

Reads Just 3 Columns

GM	30.77	1,000	NYSE	1/17/2007
GM	30.77	10,000	NYSE	1/17/2007
GM	30.78	12,500	NYSE	1/17/2007
AAPL	93.24	9,000	NQDS	1/17/2007

Assuming each column is same size, reduces bytes read from disk by factor of 3/5

In reality, databases are often 100's of columns

Linearizing a Table – Row store

C1	C2	C3	C4	C5	C6

<u>Memory/Disk (Linear Array)</u>

R1 C1
R1 C2
R1 C3
R1 C4
R1 C5
R1 C6
R2 C1
R2 C2
R2 C3
R2 C4
R2 C5
R2 C6
R3 C1
R3 C2
R3 C3
R3 C4
R3 C5
R3 C6
R4 C1
R4 C2
R4 C3
R4 C4
R4 C5
R4 C6

Linearizing a Table – Column Store

C1	C2	C3	C4	C5	C6

Memory/Disk
(Linear Array)

R1 C1
R2 C1
R3 C1
R4 C1
R5 C1
R6 C1
R1 C2
R2 C2
R3 C2
R4 C2
R5 C2
R6 C2
R1 C3
R2 C3
R3 C3
R4 C3
R5 C3
R6 C3
R1 C4
R2 C4
R3 C4
R4 C4
R5 C4
R6 C4

Tables Often Super Wide

- Data warehouse at Cambridge Mobile Telematics

Table	#columns
t1	251
t2	248
t3	134
t4	107
t5	87
t6	83
t7	71
t8	54
t9	52
t10	45

Average query access 4-5 fields

Top 2-3 tables involved in nearly every query

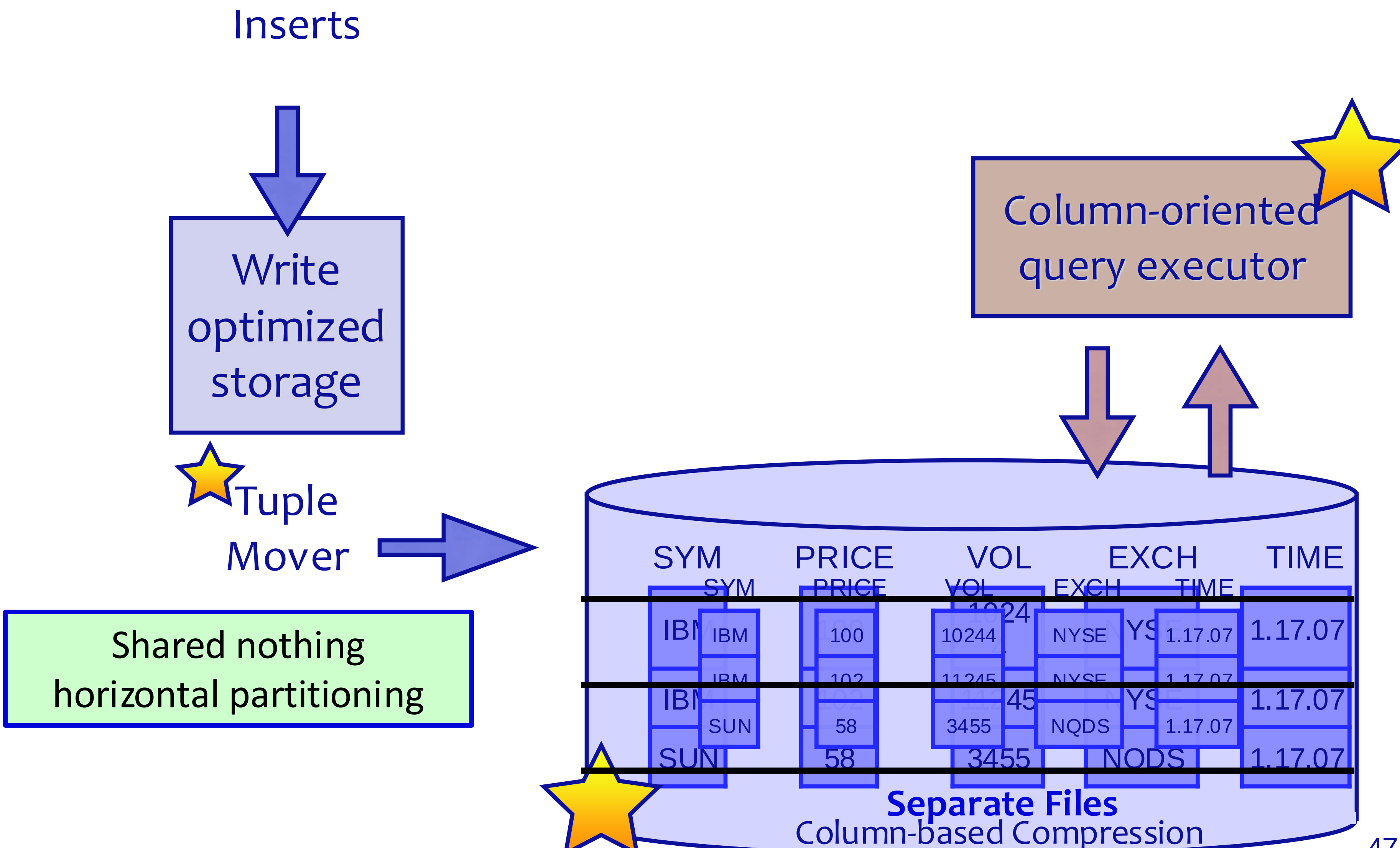
Using a row-store would impose ~200/4 = 50x performance overhead

When Are Columns Right?

- Warehousing (OLAP)
  - Read-mostly; batch update
  - Queries: Scan and aggregate a few columns
- Vs. Transaction Processing (OLTP)
  - Write-intensive, mostly single record ops.
- Column-stores: OLAP optimized
- In practice >10x performance on comparable HW, for many real world analytic applications
  - True even if w/ Flash or main memory!

Different architectures for different workloads

C-Store: Rethinking Database Design from the Ground Up

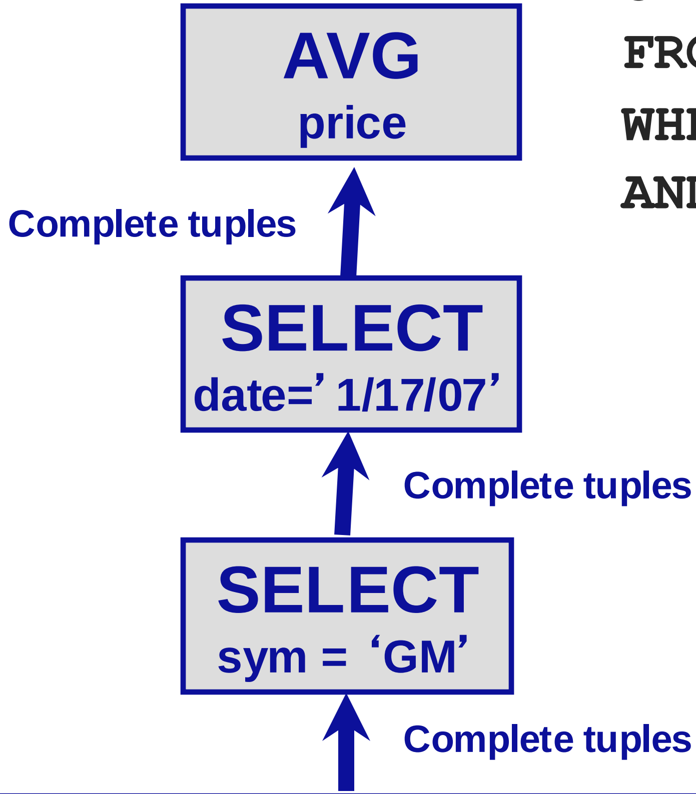


"C-Store: A Column-oriented DBMS" -- VLDB 05

Query Processing Example

- **Traditional Row Store**

```
SELECT avg(price)
FROM tickstore
WHERE symbol = 'GM'
AND date = '1/17/2007'
```



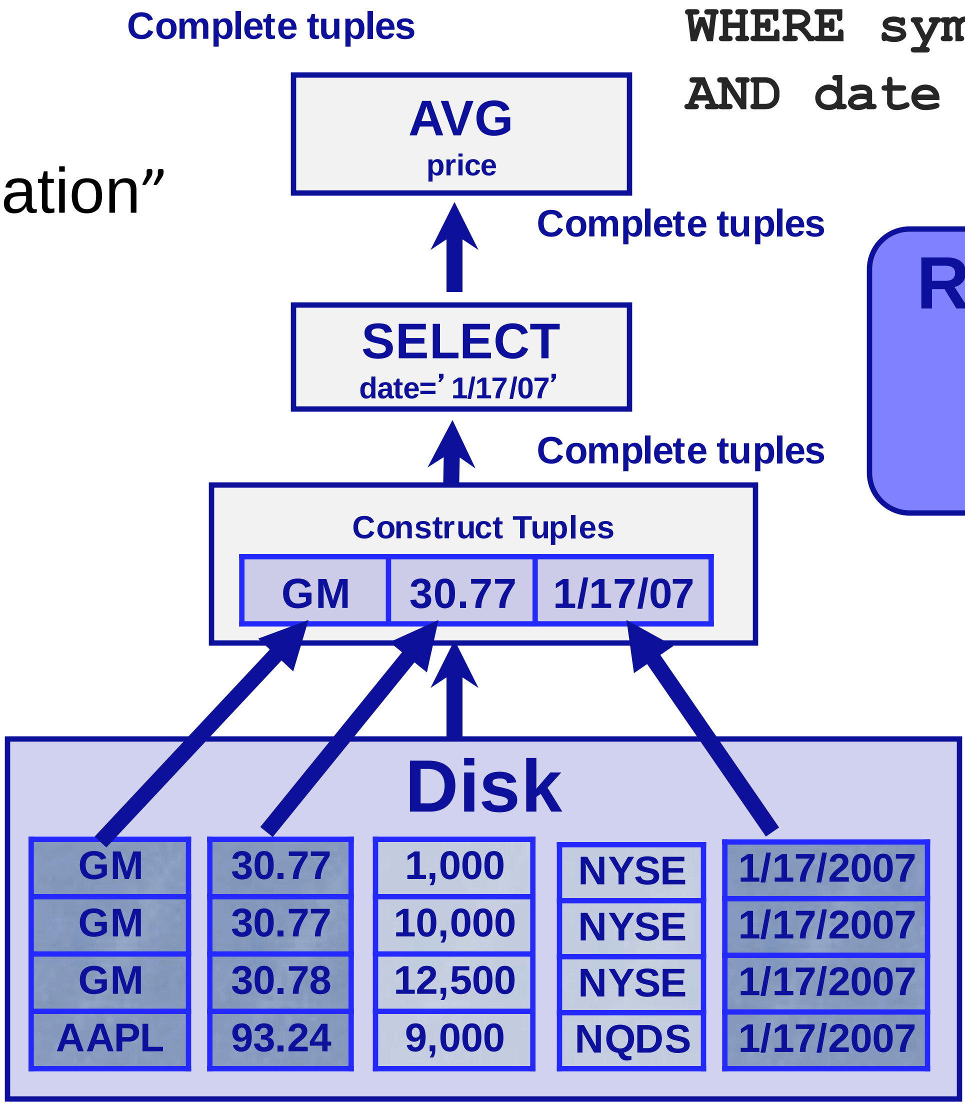
Disk

GM	30.77	1,000	NYSE	1/17/2007
GM	30.77	10,000	NYSE	1/17/2007
GM	30.78	12,500	NYSE	1/17/2007
AAPL	93.24	9,000	NQDS	1/17/2007

Query Processing Example

- **Basic Column Store**
- "Early Materialization"

```
SELECT avg(price)
FROM tickstore
WHERE symbol = 'GM'
AND date = '1/17/2007'
```

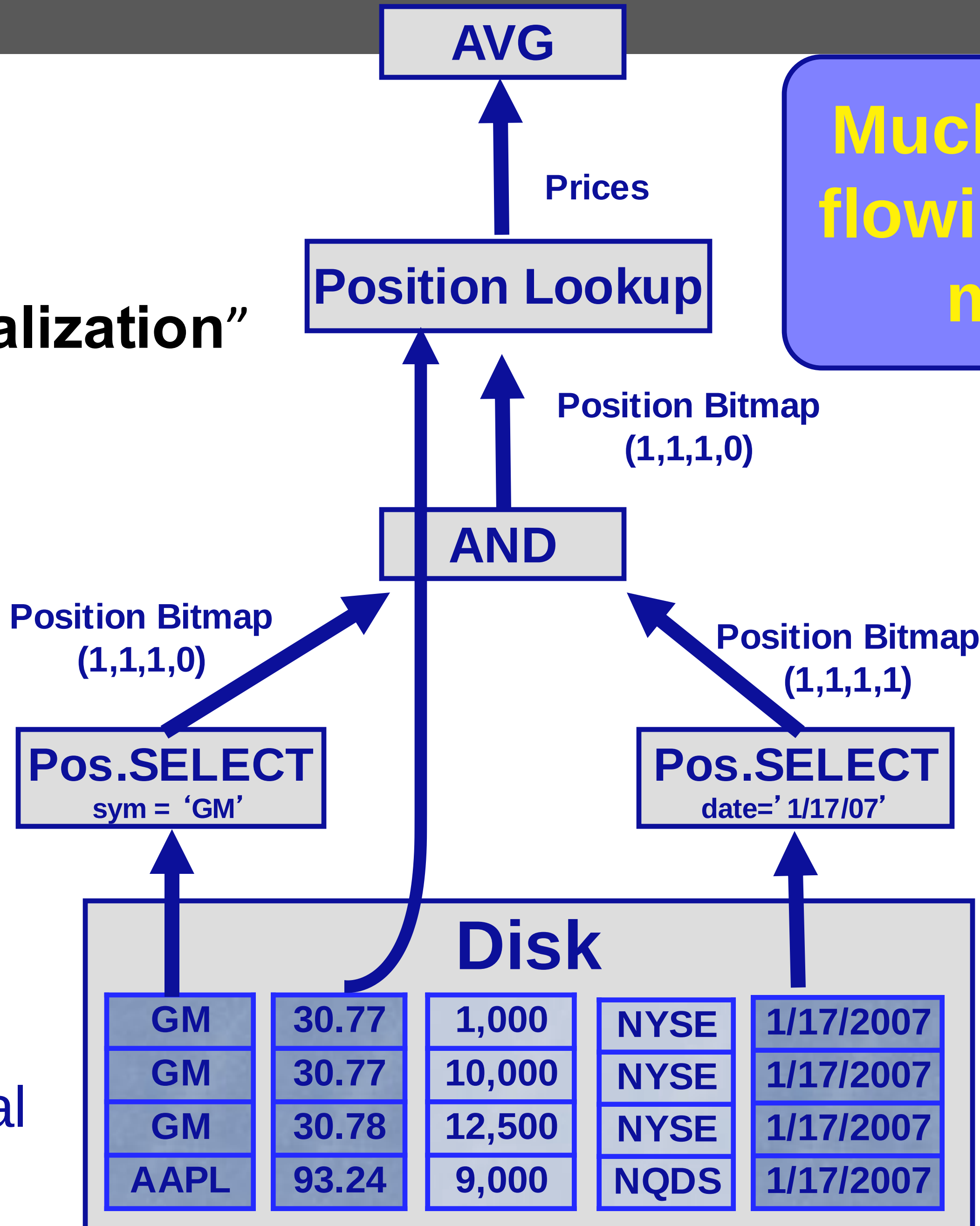


Row-oriented plan

Fields from same tuple at same index (position) in each column file

Query Processing Example

- **C-Store**
  - **“Late Materialization”**



Much less data flowing through memory

See Abadi et al ICDE 07

Why Compress?

- Database size is 2x-5x larger than the volume of data loaded into it
- Database performance is proportional to the amount of data flowing through the system

Abadi et al, SIGMOD 06

Column-Oriented Compression

- Query engine processes compressed data
- Transfers load from disk to CPU
- Multiple compression types
  - Run-Length Encoding (RLE), LZ, Delta Value, Block Dictionary Bitmaps, Null Suppression
- System chooses which to apply
- Typically see 50% - 90% compression
- NULLs take virtually no space

Columns contain similar data, which makes compression easy

RLE	Delta	LZ	RLE	RLE
3xGM	30.77	1,000	3xNYSE	4 x 1/17/2007
1XAPPL	30.77	10,000 12,500	1XNQDS	1/17/2007
GM	30.78	9,000 12,500	NYSE	1/17/2007
AAPL	93.24	9,000	NQDS	1/17/2007

Run Length Encoding

- Replace repeated values with a count and a value
- For single values, use a run length of 1
  - Naively, can increase storage space
  - Can use a shorter bit sequence for 1s, at the cost of more expensive decompression
- E.g., 1110002 → 3x1, 3x0, 1x2
- Works well for mostly-sorted, few-valued columns

Dictionary Encoding

- **Many variants; simplest is to replace string values with integers and maintain a dictionary**
- **I.e., AAPL, AAPL, IBM, MSFT →**

  1,1,2,3 + 1:AAPL, 2:IBM, 3:MSFT
- **Works well for few-valued string columns**
  - **Choice of dictionary not obvious**
  - **Words? Records?**

Lempel Ziv Encoding

- LZ (“Lempel Ziv”) Compression
- General purpose lossless data compression
- Builds data dictionary dynamically as it runs
    - Add new bit strings to the dictionary as they are encountered
- Treat entire column as a document

Delta Encoding

- **Consecutive values encoding as difference to previous values**
- 1.1, 1.2, 1.3 → 1.1, +.1, +1
  - After encoding as deltas, bit-pack
  - Works if deltas can be represented in fewer bits than whole values
- Works well for e.g., floats with small variations

Bitmap Encoding

- Encode few valued columns as bitmaps
- M M M F F → 11100, 00011
  - If fewer distinct values than bitwidth of field, saves space
  - Bitmaps can be further compressed, e.g., using RLE
- Bitmaps are very good for certain kinds of operations, e.g., filtering

Sorted Data

- Delta and RLE work great on sorted data
- Trick: Secondary sorting

X	Y
a	2
b	2
a	1
b	1

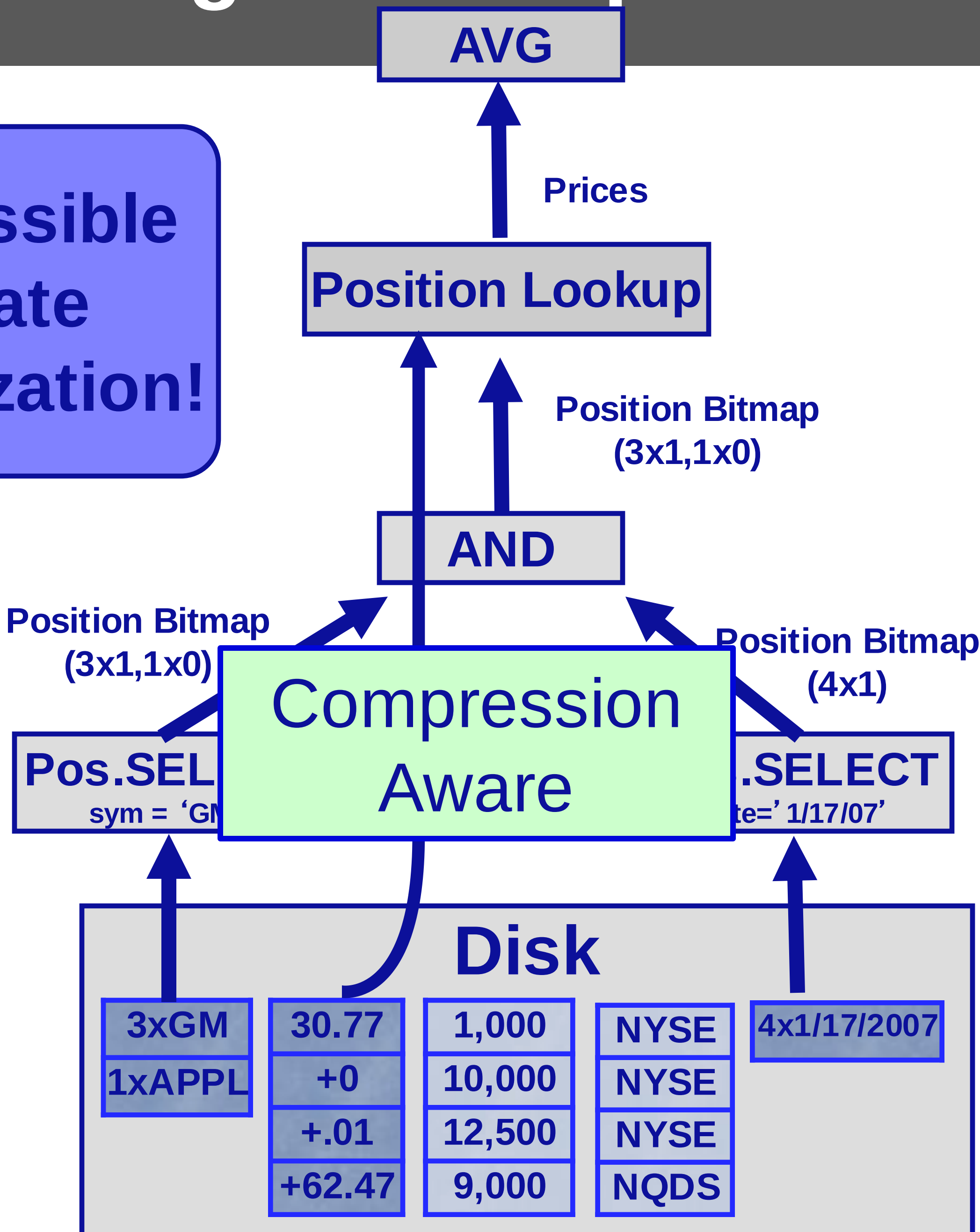
Sort on X, then Y →

X	Y
a	1
a	2
b	1
b	2

Y is not sorted, but if many duplicates of X, will be "mostly" sorted

Operating on Compressed Data

Only possible with late materialization!



Direct Operation Optimizations

- **Compressed data used directly for position lookup**
  - **RLE, Dictionary, Bitmap**

- **Direct Aggregation and GROUP BY on compressed blocks**
  - **RLE, Dictionary**

- **Join runs of compressed blocks**
  - **RLE, Dictionary**

- **Min/max directly extracted from sorted data**

Compression + Sorting is a Huge Win

- **How can we get more sorted data?**
- **Store duplicate copies of data**
  - **Use different physical orderings**

- **Improves ad-hoc query performance**
  - **Due to ability to directly operate on sorted, compressed data**

- **Supports fail-over / redundancy**

Study Break: Compression

- For each of the following columns, what compression method would you recommend?

(Choose from A. RLE, B. Dictionary, C. Bitmap, D. Delta, E. Bit-packing)

https://clicker.mit.edu/6.5830/

An unsorted column of integers in the range 0-100

Delta/Bit-packing (LZ/dictionary also OK)

A mostly sorted column of arbitrary strings

LZ

A mostly sorted column of integers in the range 0-10

RLE

A sorted column of floats

Delta

Bitmap

Write Performance

Trickle load: Very Fast Inserts

> **Write-optimized Column Store (WOS)**

Memory: mirrored projections in insertion order (uncompressed)

Tuple Mover

Asynchronous Data Movement

Batched

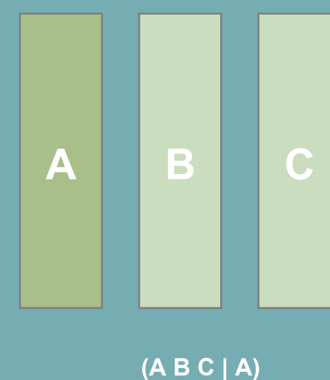
Amortizes seeks

Amortizes recompression

Enables continuous load

> **Read-optimized Column Store (ROS)**

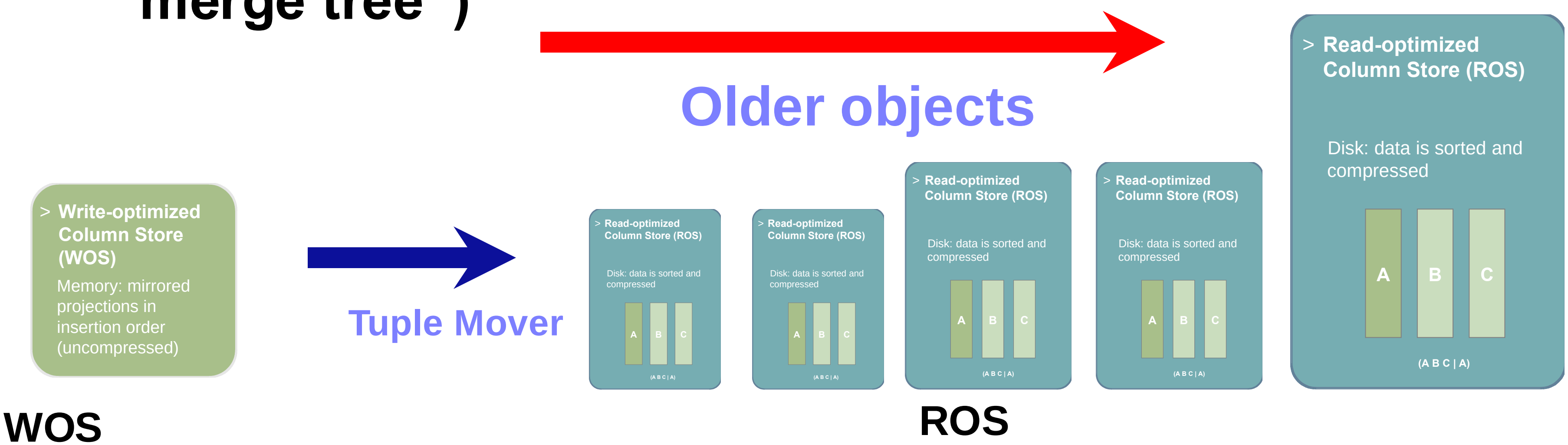
Disk: data is sorted and compressed



Queries read from both WOS and ROS

When to Rewrite ROS Objects?

- Store multiple ROS objects, instead of just one
  - Each of which must be scanned to answer a query
- Tuple mover writes new objects
  - Avoids rewriting whole ROS on merge
- Periodically merge ROS objects to limit number of distinct objects that must be scanned (“Log structured merge tree”)



Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

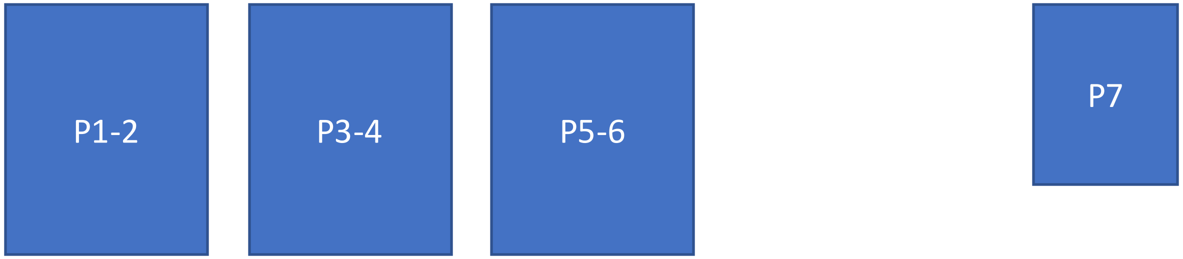
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

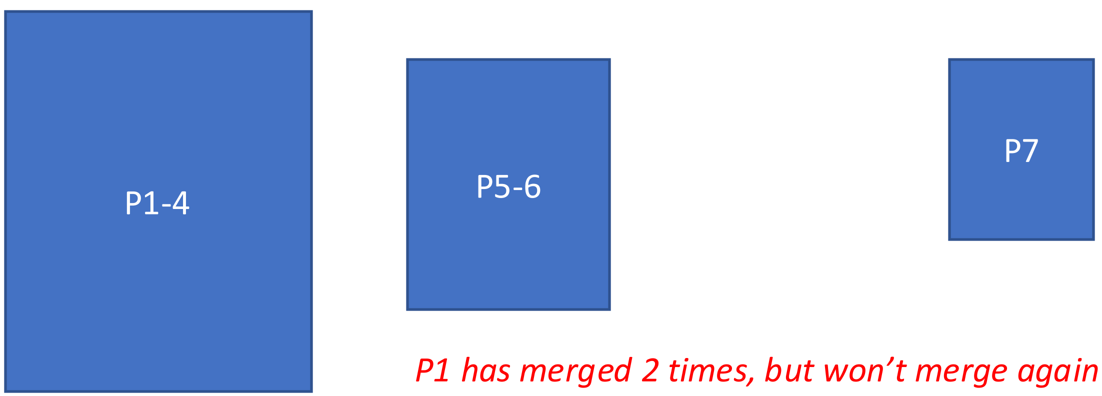
- Log structured merge tree: arrange so partitions merge a logarithmic number of times



Problem: Lots of Partitions

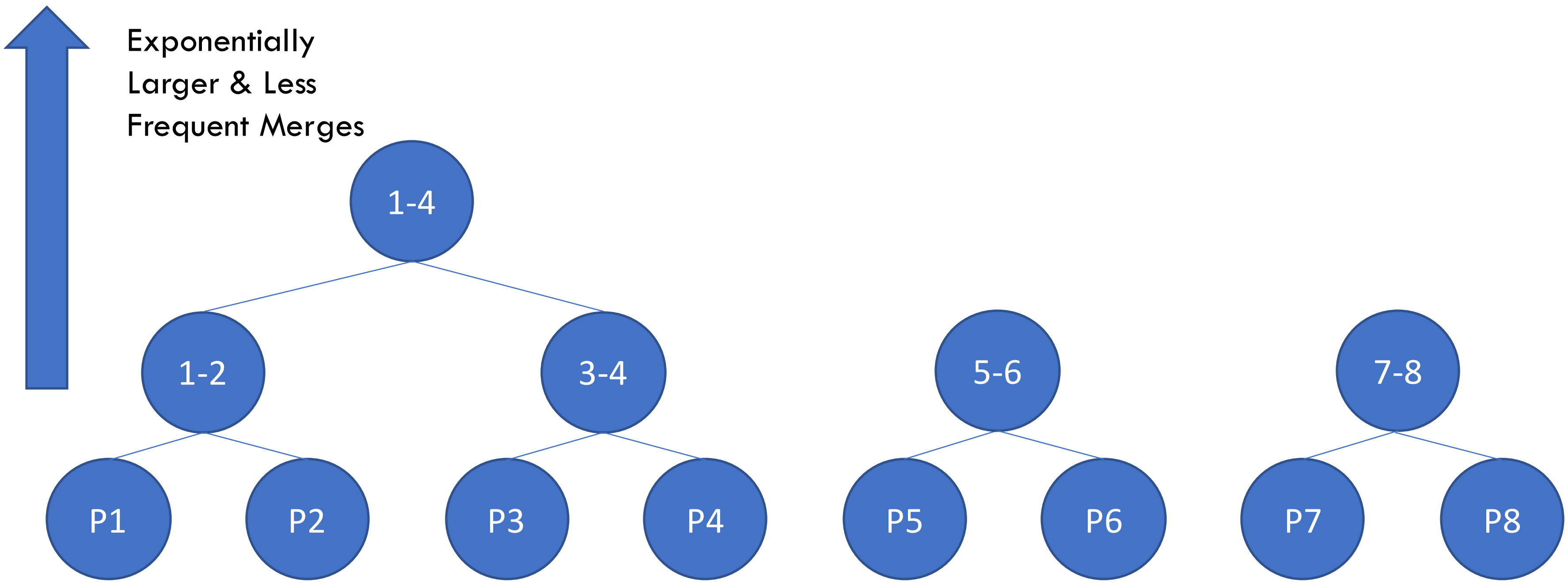
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times



P1 has merged 2 times, but won't merge again until after 8 more partitions arrive

Log Structure Merge Tree



Column-Oriented Data In Modern Systems

- C-Store commercialized as Vertica
- Although it wasn’t the first column-oriented DB, it led to a proliferation of commercial column-oriented systems
- Now the de-facto way that analytic database systems are built, including Snowflake, Redshift, and others.

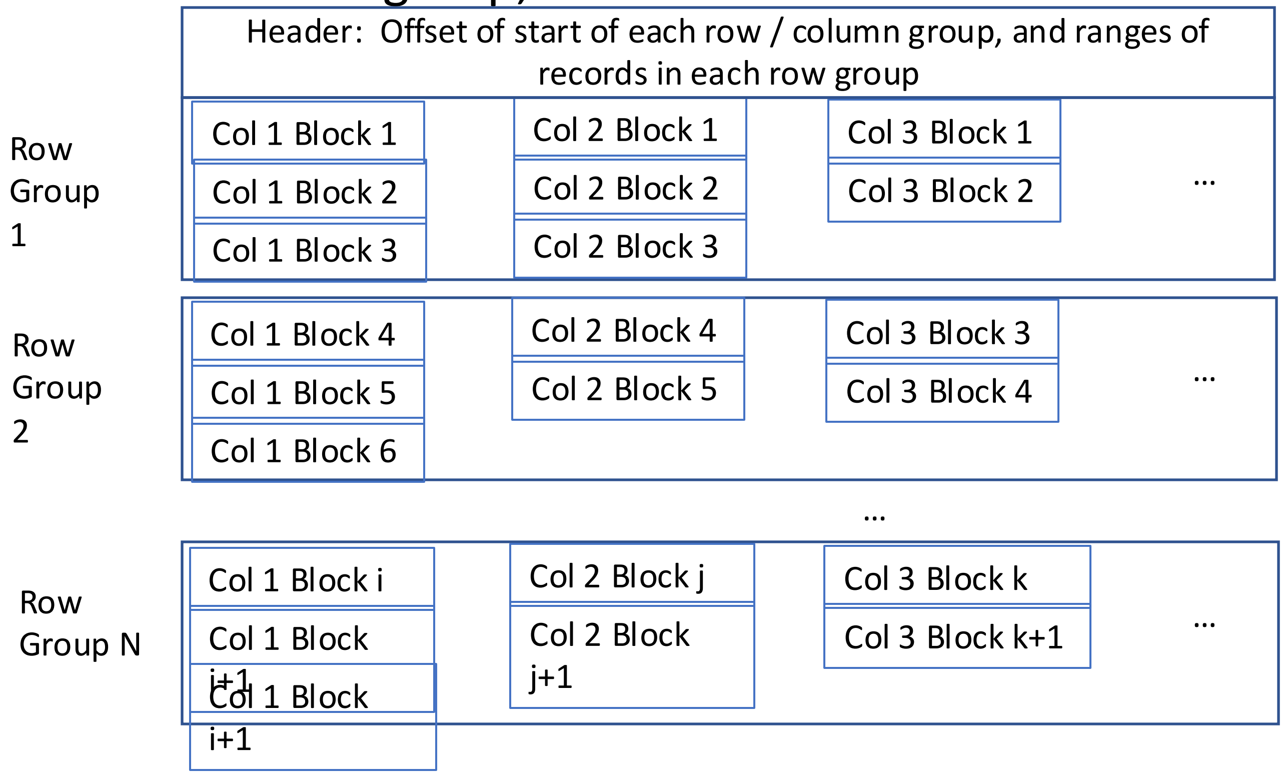
- One popular open-source option: Parquet

Efficient Data Loading: Parquet

- Parquet is column-oriented file format that is MUCH more efficient than CSV for storing tabular data
- Vs CSV, Parquet is stored in binary representation
  - Uses less space
  - Doesn’t require conversion from strings to internal types
  - Doesn’t require parsing or error detection
  - Column-oriented, making access to subsets of columns much faster

Parquet Format

- Data is partitioned sets of rows, called "row groups"
- Within each row group, data from different columns is stored separately



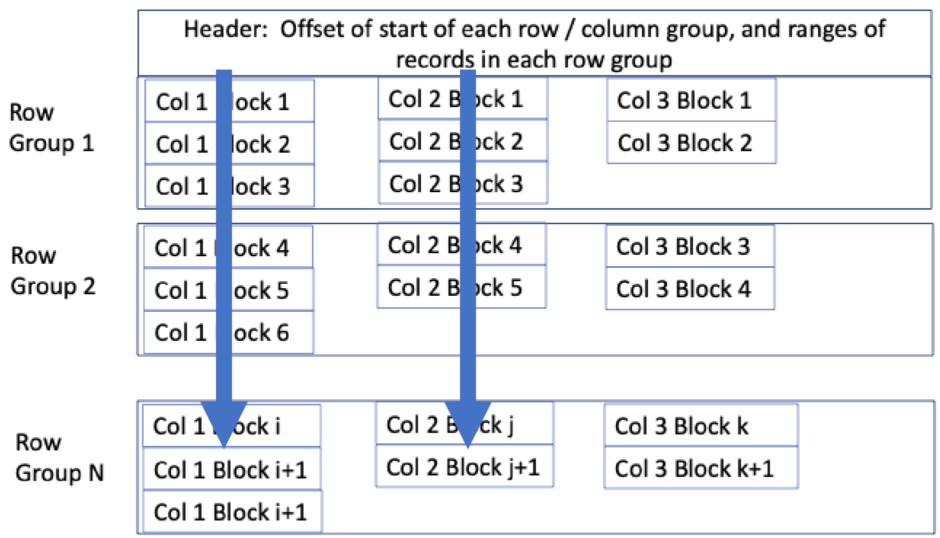
Using header, can efficiently read any subset of columns or rows without scanning whole file (unlike CSV)

Within a row group, data for each column is stored together

Predicate Pushdown w/ Parquet & Pandas

```
pd.read_parquet(‘file.pq’, columns=[‘Col 1’, ‘Col 2’])
```

- Only reads col1 and col2 from disk
- For a wide dataset saves a ton of I/O



Performance Measurement

- Compare reading CSV to parquet to just columns we need

```
t = time.perf_counter()
df = pd.read_csv("FARS2019NationalCSV/Person.CSV", encoding = "ISO-8859-1")
print(f"csv elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq")
print(f"parquet elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq", columns = ['STATE','ST_CASE','DRINKING','PER_TYP'])
print(f"parquet subset elapsed = {time.perf_counter() - t:.3} seconds")
```

```
csv elapsed = 1.18 seconds
parquet elapsed = 0.338 seconds
parquet subset elapsed = 0.025 seconds
```

47x speedup

When to Use Parquet?

- Will always be more efficient than CSV
- Converting from Parquet to CSV takes time, so only makes sense to do so if working repeatedly with a file
- Parquet requires a library to access/read it, whereas many tools can work with CSV
- Because CSV is text, it can have mixed types in columns, or other inconsistencies
  - May be useful sometimes, but also very annoying!
  - Parquet does not support mixed types in a column

Summary

- Column oriented databases are a different way to “linearize” data to disk than the row-oriented representation we have studied
- A good fit for “warehousing” workloads that mostly read many records of a few tables
- C-Store system implements many additional ideas:
  - “Late materialization” execution
  - Column-specific compression and direct execution on compressed data
  - Read/write optimized stores
- Ideas have found their way into many modern systems and libraries, e.g., Parquet