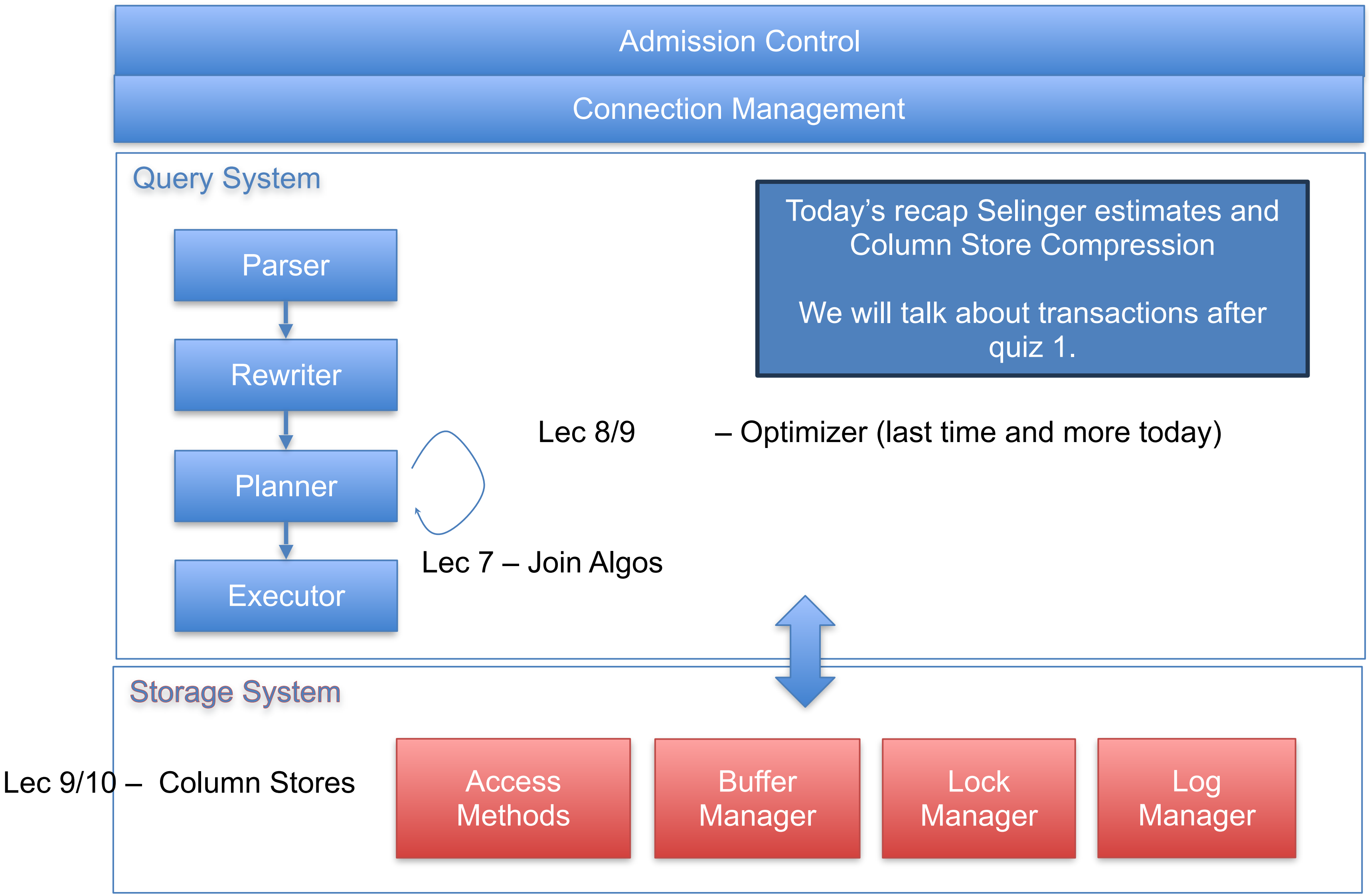
6.5830 Lecture 10 – Column Stores ctd

10/07/2024

Where are we


Admission Control
Connection Management
Query System
Parser
Rewriter
Planner
Executor
Today's recap Selinger estimates and Column Store Compression
We will talk about transactions after quiz 1.
Lec 8/9 – Optimizer (last time and more today)
Lec 7 – Join Algos
Storage System
Lec 9/10 – Column Stores
Access Methods
Buffer Manager
Lock Manager
Log Manager

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 Selectivities

F(pred) = Selectivity of predicate = Fraction of records that a predicate does not filter

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

Predicate types

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

2. col > val
(max key - value) / (max key - min key) *(if index available)*
1/3 otherwise

Modern DBs use fancier stats!

3. col1 = col2
1/MAX(ICARD(col1), ICARD(col2)) *(if index available)*
1/10 otherwise

Assumes key-foreign key join
Note a better estimate is 1/ICARD(PK table)
We use 1/ICARD(PK table) going forward

Important: not all joins are FK to PK
→ equation on the left is still important

Example

Product (<u>Pid</u>, Name, Price)
Order(<u>Oid</u>, CName, Address)
Customer(<u>CName</u>, Name)
Orderline (<u>Pid, Oid</u>, Amount)
Special_Products (Pid)

Product

<u>Pid</u>	Name	Price
1	Chocolate Donut	1
2	Glazed Donut	1
3	Boston Crème Donut	1.5
4	Sprinkles Donut	1
5	Cinnamon Donut	0.5
6	MIT special	1.5

$NCARD(P)=6$
$ICARD_{pid}(P)=6$
$ICARD_{price}(P)3$
$MIN_{price}(P)=0.5$
$MAX_{price}(P)=1.5$

Order

<u>Oid</u>	Cid	Address
1	Tim	Cambridg e
2	Tim	Arlington
3	Mike	Newton

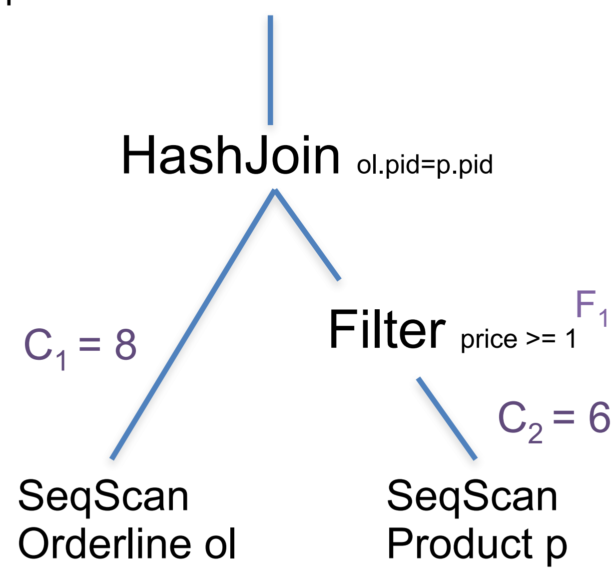
$NCARD(O)=3$
$ICARD_{oid}(O)=3$

OrderLine

<u>Oid</u>	<u>Pid</u>	Amount
1	1	1
1	4	1
2	1	3
2	3	2
3	5	3
3	1	3
3	4	2
3	3	2

$NCARD(OL)=8$
$ICARD_{oid,pid}(OL)=8$
$ICARD_{oid}(OL)=3$
$ICARD_{pid}(OL)=4$

```
SELECT *
FROM Orderline ol
    join Product p on ol.pid = p.pid
WHERE price >= 1
```



Clicker (http://clicker.mit.edu/6.5830)
What is the selectivity of F_1
(assuming only the Selinger stats)
A) 0.5
B) 0.3333
C) 0.7777
D) 0.83333333333

Example

Product (<u>Pid</u>, Name, Price)
Order(<u>Oid</u>, CName, Address)
Customer(<u>CName</u>, Name)
Orderline (<u>Pid, Oid</u>, Amount)
Special_Products (Pid)

Product

Pid	Name	Price
1	Chocolate Donut	1
2	Glazed Donut	1
3	Boston Crème Donut	1.5
4	Sprinkles Donut	1
5	Cinnamon Donut	0.5
6	MIT special	1.5

NCARD(P)=6
$ICARD_{pid}(P)=6$

$ICARD_{price}(P)3$
$MIN_{price}(P)=0.75$
$MAX_{price}(P)=1.5$

Order

Oid	Cid	Address
1	Tim	Cambridge
2	Tim	Arlington
3	Mike	Newton

NCARD(O)=3
$ICARD_{oid}(O)=3$

OrderLine

Oid	Pid	Amount
1	1	1
1	4	1
2	1	3
2	3	2
3	5	3
3	1	3
3	4	2
3	3	2

NCARD(OL)=8
$ICARD_{oid,pid}(OL)=8$
$ICARD_{oid}(OL)=3$
$ICARD_{pid}(OL)=4$

```
SELECT *
FROM Orderline ol
    join Product p on ol.pid = p.pid
WHERE price >= 1
```

HashJoin ol.pid=p.pid F_2

$C_3 = 6 * 0.5 = 3$

$C_1 = 8$

Filter price >= 1

F_1 = (max key - value) / (max key - min key)
= (1.5 – 1) / (1.5 – 0.5) = 0.5

$C_2 = 6$

SeqScan Orderline ol

SeqScan Product p

Clicker (http://clicker.mit.edu/6.5830)
What is the selectivity of F_2
(assuming only the Selinger stats)
A) 1/8
B) 1/6
C) 1/3
D) 1

Example

Product (Pid, Name, Price)
Order(Oid, CName, Address)
Customer(CName, Name)
Orderline (Pid, Oid, Amount)
Special_Products (Pid)

Product

Pid	Name	Price
1	Chocolate Donut	1
2	Glazed Donut	1
3	Boston Crème Donut	1.5
4	Sprinkles Donut	1
5	Cinnamon Donut	0.5
6	MIT special	1.5

$NCARD(P)=6$
$ICARD_{pid}(P)=6$

$ICARD_{price}(P)3$
$MIN_{price}(P)=0.75$
$MAX_{price}(P)=1.5$

Order

Oid	Cid	Address
1	Tim	Cambridge
2	Tim	Arlington
3	Mike	Newton

$NCARD(O)=3$
$ICARD_{oid}(O)=3$

OrderLine

Oid	Pid	Amount
1	1	1
1	4	1
2	1	3
2	3	2
3	5	3
3	1	3
3	4	2
3	3	2

$NCARD(OL)=8$
$ICARD_{oid,pid}(OL)=8$
$ICARD_{oid}(OL)=3$
$ICARD_{pid}(OL)=4$

```
SELECT *
FROM Orderline ol
    join Product p on ol.pid = p.pid
WHERE price >= 1
```

$C_4 = 24 * 1 / 6 = 4$

$F_2 = 1 / PK = 1 / 6$

HashJoin ol.pid=p.pid

C_{join} w/o predicate
$= 3 * 8 = 24$ (cartesian product)

$C_3 = 6 * 0.5 = 3$

$C_1 = 8$

Filter price >= 1

$F_1 = (\text{max key} - \text{value}) / (\text{max key} - \text{min key})$
$= (1.5 – 1) / (1.5 – 0.5) = 0.5$

$C_2 = 6$

SeqScan
Orderline ol

SeqScan
Product p

Example

Product (Pid, Name, Price)
Order(Oid, CName, Address)
Customer(CName, Name)
Orderline (Pid, Oid, Amount)
Reviews (Pid, Review)

Product

Pid	Name	Price
1	Chocolate Donut	1
2	Glazed Donut	1
3	Boston Crème Donut	1.5
4	Sprinkles Donut	1
5	Cinnamon Donut	0.5
6	MIT special	1.5

$NCARD(P)=6$
$ICARD_{pid}(P)=6$

$ICARD_{price}(P)3$
$MIN_{price}(P)=0.5$
$MAX_{price}(P)=1.5$

Reviews

Rid	Pid	Review
1	1	Good
2	1	Good
3	2	Bad
3	2	OK

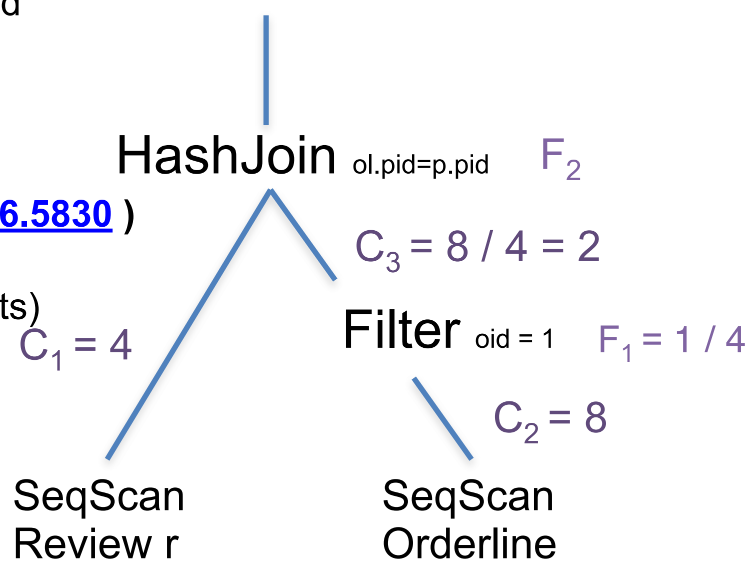
$NCARD(O)=3$
$ICARD_{rid}(O)=3$
$ICARD_{pid}(O)=2$

OrderLine

Oid	Pid	Amount
1	1	1
1	4	1
2	1	3
2	3	2
3	5	3
3	1	3
3	4	2
4	3	2

$NCARD(O)=8$
$ICARD_{oid,pid}(O)=8$
$ICARD_{oid}(O)=4$
$ICARD_{pid}(O)=4$

```
SELECT *
FROM Orderline
    join Review r on r.pid = ol.pid
WHERE oid = 1
```



Clicker (http://clicker.mit.edu/6.5830)
What is the selectivity of F_2
(assuming only the Selinger stats)

A) 1/8
B) 1/4
C) 1/2
D) 1

Example

Product (<u>Pid</u>, Name, Price)
Order(<u>Oid</u>, CName, Address)
Customer(<u>CName</u>, Name)
Orderline (<u>Pid, Oid</u>, Amount)
Reviews (Pid, Review)

Product

<u>Pid</u>	Name	Price
1	Chocolate Donut	1
2	Glazed Donut	1
3	Boston Crème Donut	1.5
4	Sprinkles Donut	1
5	Cinnamon Donut	0.5
6	MIT special	1.5

$NCARD(P)=6$
$ICARD_{pid}(P)=6$

$ICARD_{price}(P)=3$
$MIN_{price}(P)=0.5$
$MAX_{price}(P)=1.5$

Reviews

<u>Rid</u>	Pid	Review
1	1	Good
2	1	Good
3	2	Bad
3	2	OK

$NCARD(O)=3$
$ICARD_{rid}(O)=3$
$ICARD_{pid}(O)=2$

OrderLine

<u>Oid</u>	<u>Pid</u>	Amount
1	1	1
1	4	1
2	1	3
2	3	2
3	5	3
3	1	3
3	4	2
4	3	2

$NCARD(O)=8$
$ICARD_{oid,pid}(O)=8$
$ICARD_{oid}(O)=4$
$ICARD_{pid}(O)=4$

```
SELECT *
FROM Orderline
    join Review r on r.pid = ol.pid
WHERE oid = 1
```

$C_4 = 8 / 4 = 2$

HashJoin ol.pid=p.pid

C_{join} w/o predicate
$= 4 * 2 = 8$ (cartesian product)

$F_2 = 1 / \max(ICARD_{pid}(O), ICARD_{pid}(O))$
$= Max(2, 4) = 1/4$

$C_3 = 8 / 4 = 2$

$C_1 = 4$

Filter oid = 1 $F_1 = 1 / 4$

$C_2 = 8$

SeqScan Review r

SeqScan Orderline

Column Stores

A different way to build a database system

Linearizing a Table – Row store

C1	C2	C3	C4	C5	C6

<u>Memory/Disk</u>
<u>(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

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

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

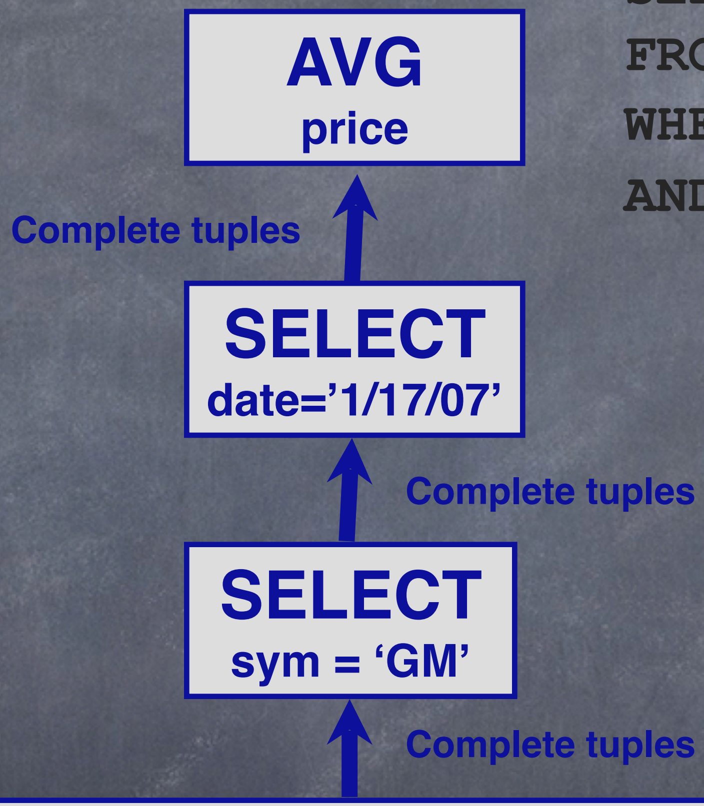
Top 2-3 tables involved in nearly every query

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

Query Processing Example

- **Traditional Row Store**

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



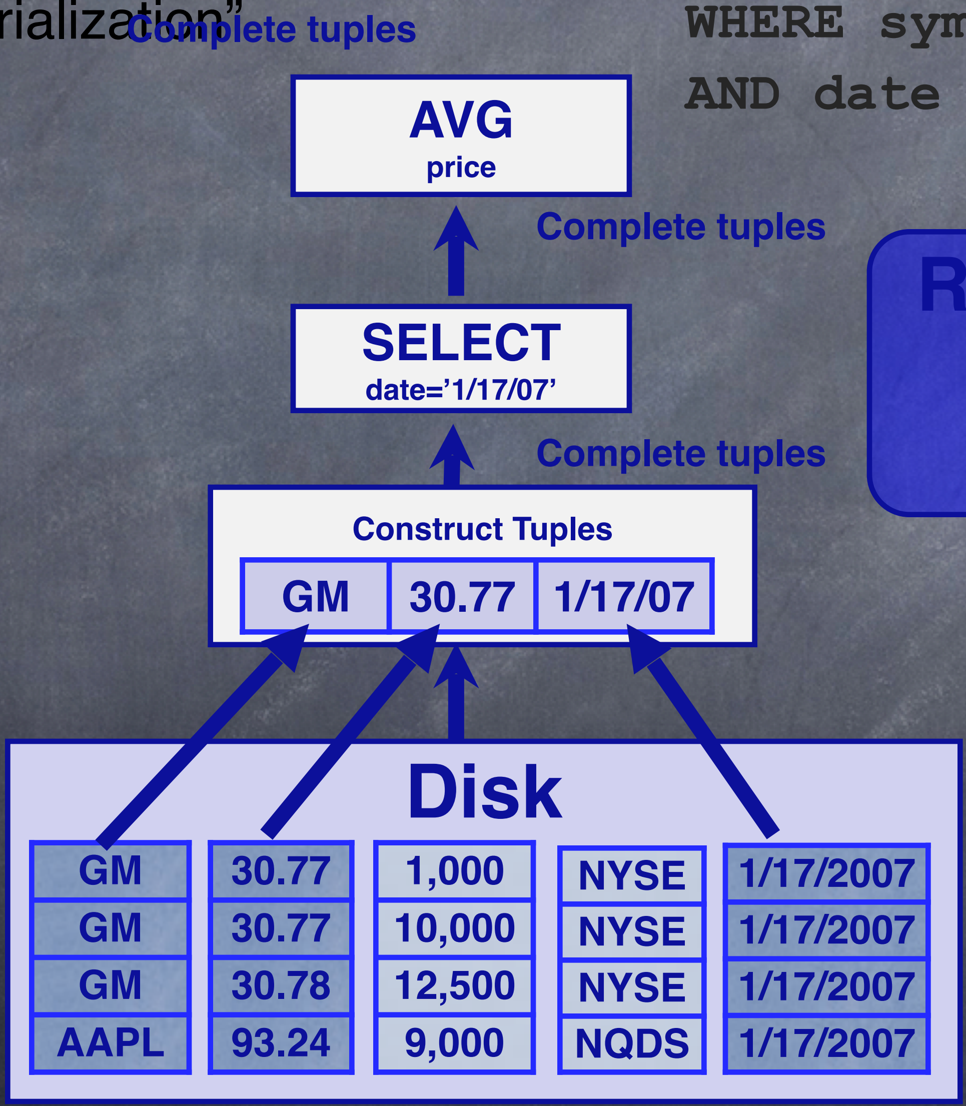
Disk

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'
```

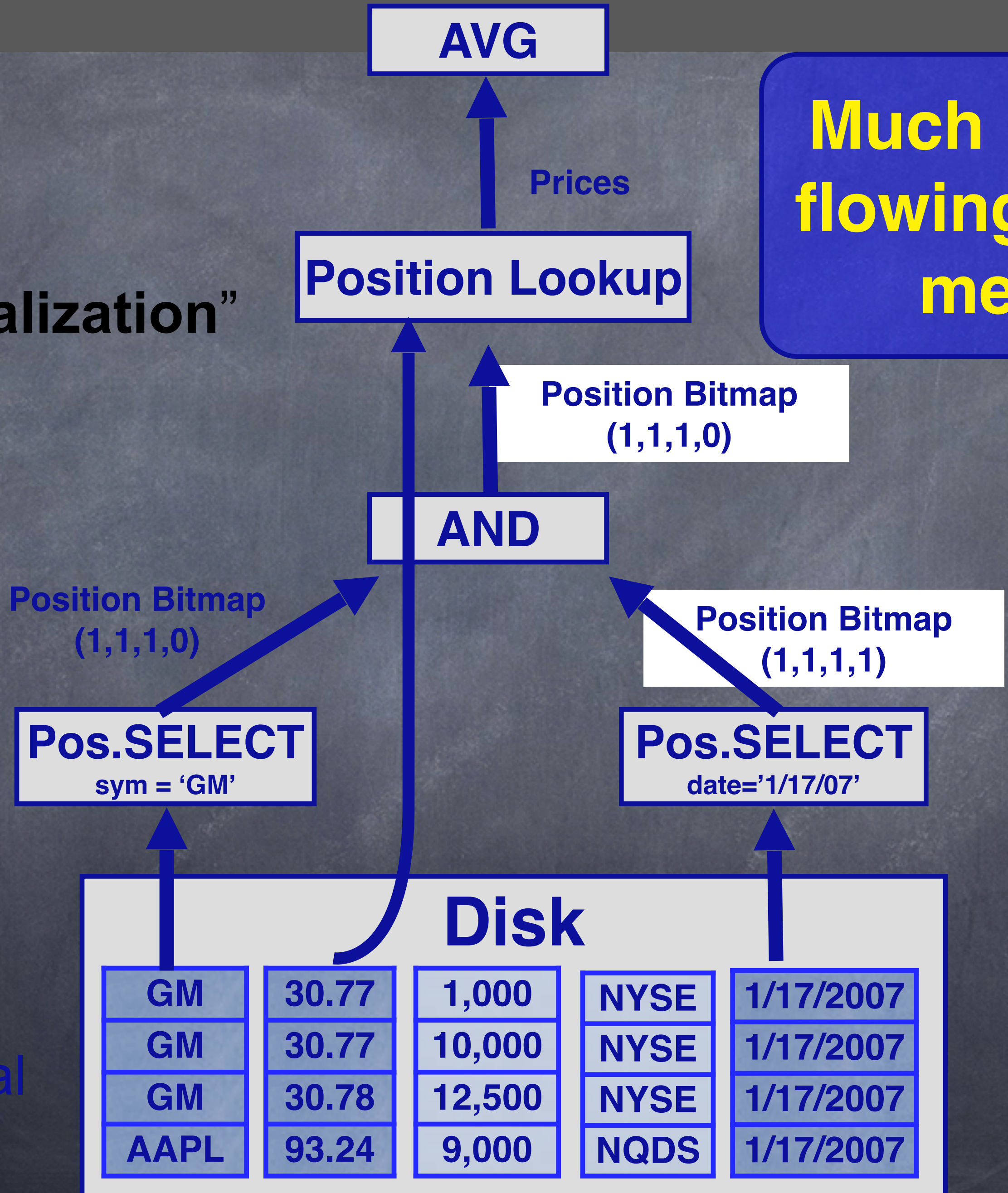


Row-oriented plan

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

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 10,000	3xNYSE	4 x 1/17/2007
1XAPPL	30.77	12,500	1XNQDS	7
GM	30.78	9,000	NYSE	1/17/200 7
AAPL	93.24	9,000	NQDS	1/17/200

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

Bit Packing

- Encode values with fewest possible bits
- Each value becomes bit-length (e.g., 0-8 or 0-32), followed by value in that many bits
- E.g.,: 1 2 37 7
  - Need 1, 2, 6, and 3 bits respectively
  - Each number becomes 3 bit header and encoded value
    - `1:      0x001, 0x1`
    - `2:      0x010, 0x10`
    - `37:     0x110, 0x100101`
    - `7:      0x011, 0x111`
  - 3 x 4 + 12 = 24 bits to encode, vs e.g., 8x4 = 32

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

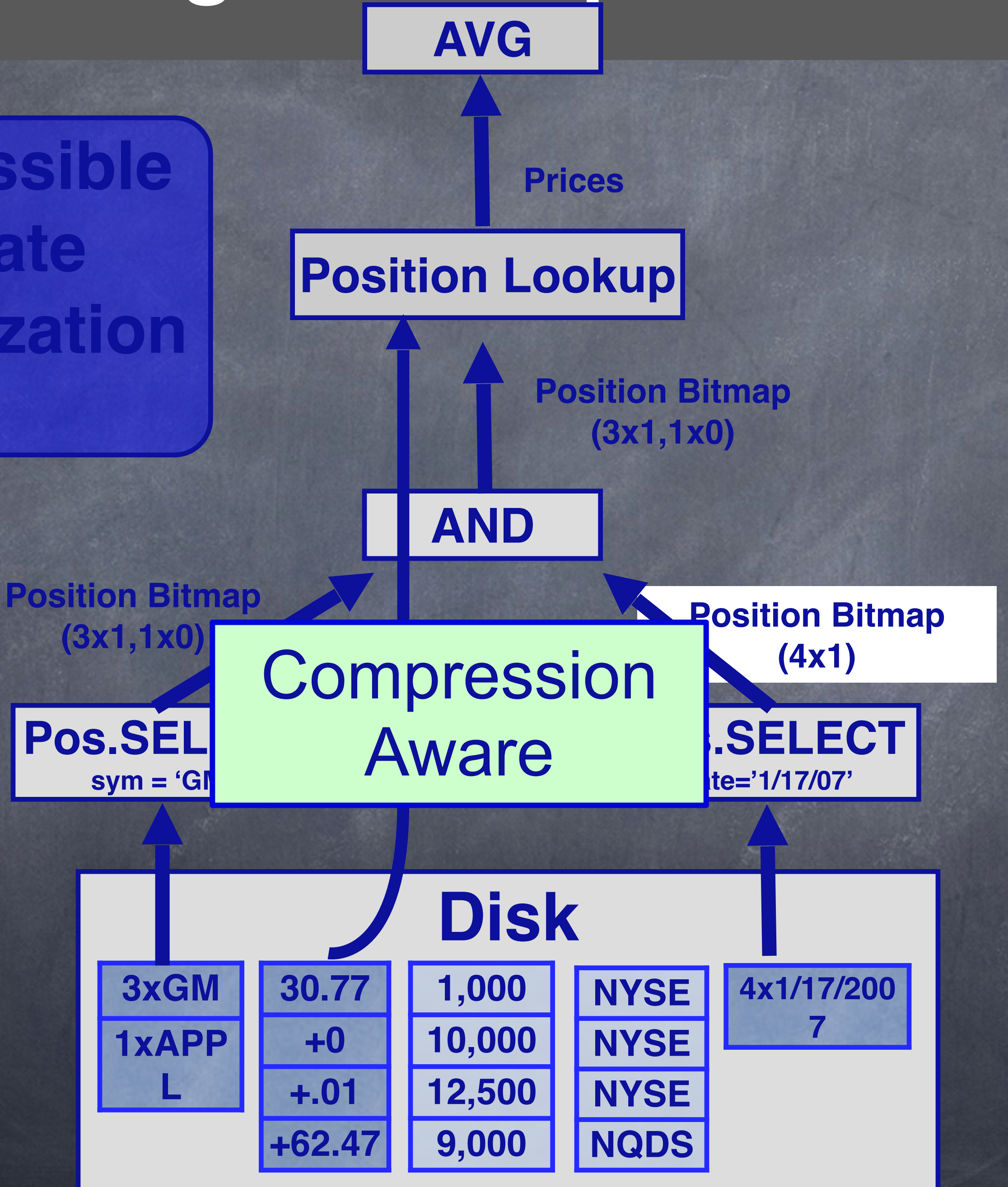
Sort on X, then Y →

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 any combination of A. RLE, B. Dictionary, C. Bitmap, D. Delta, E:LZ, F: 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 integers in the range 0-10

RLE

A sorted column of floats

Delta

An unsorted column of strings w/ 3 values

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

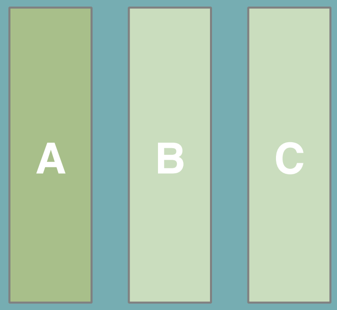
Amortizes seeks

Amortizes recompression

Enables continuous load

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

Disk: data is sorted and compressed

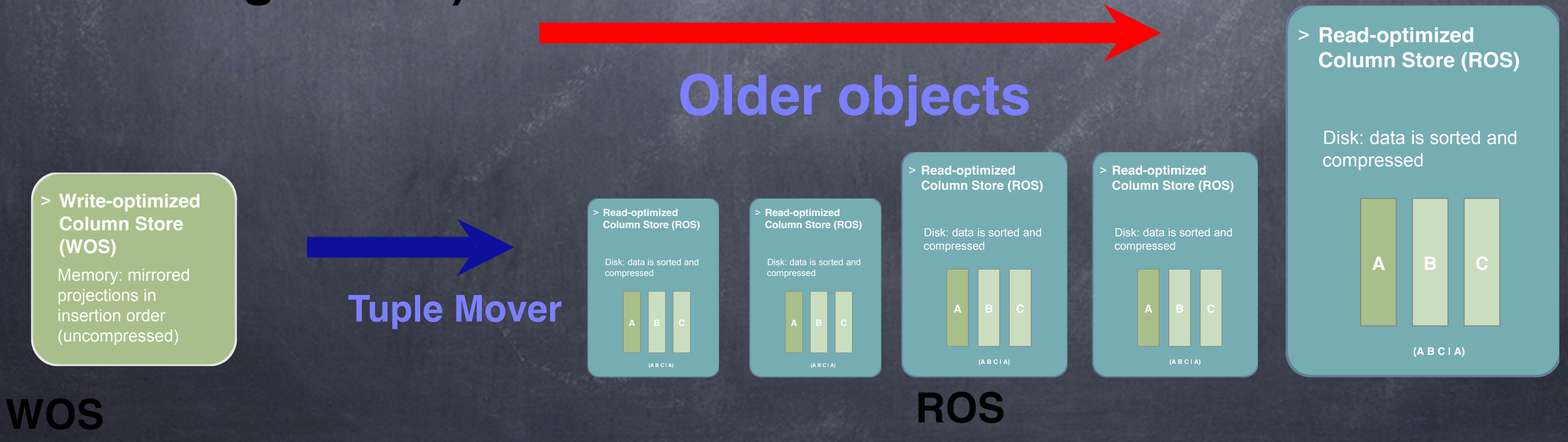


(A B C | A)

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

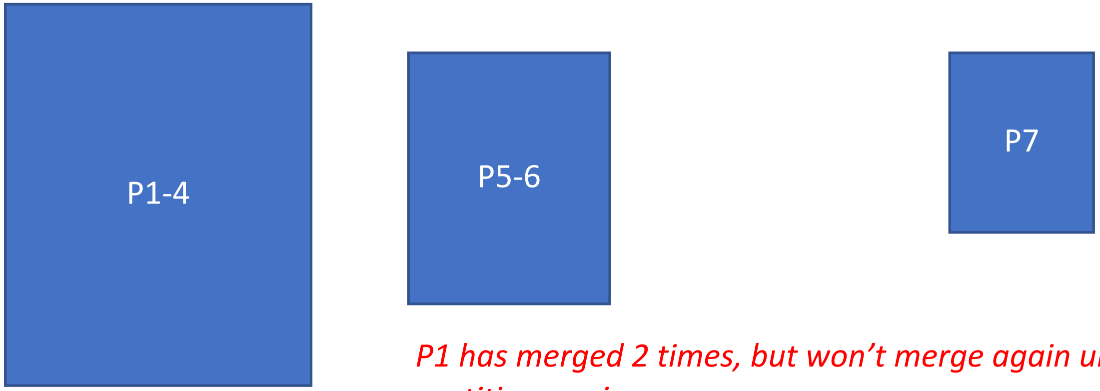
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- 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

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.

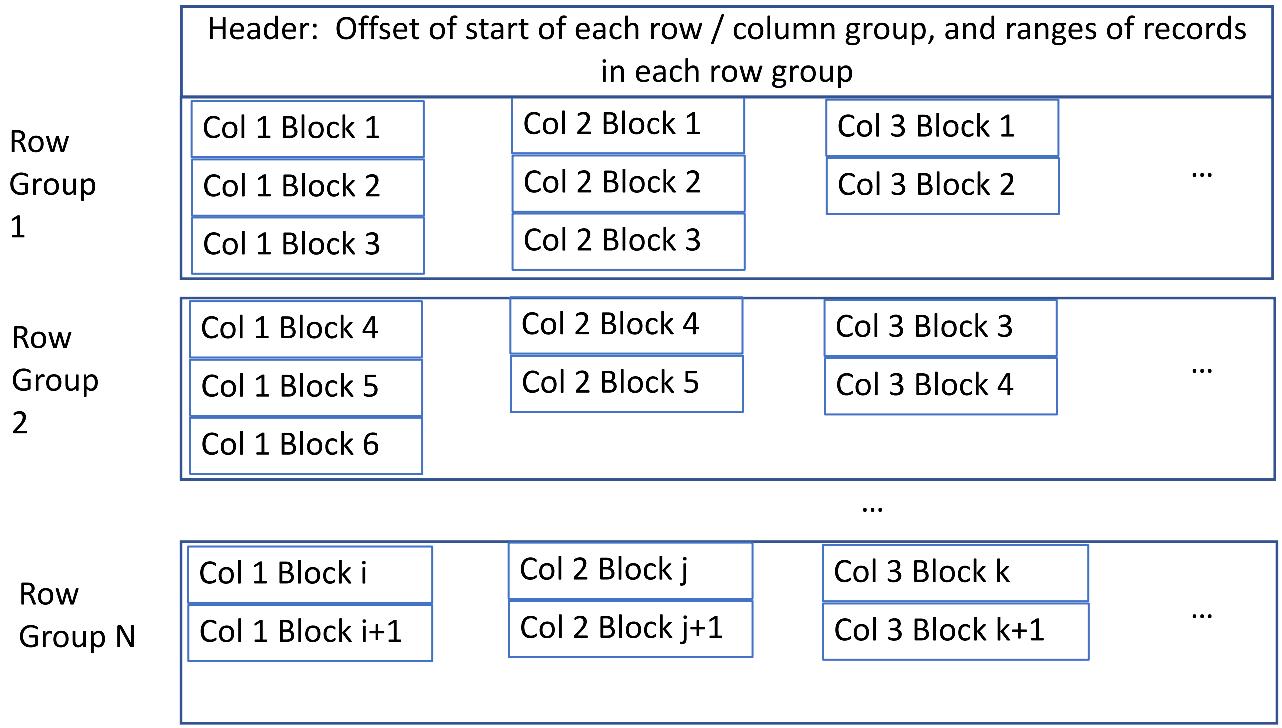
- One popular open-source option: Parquet

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



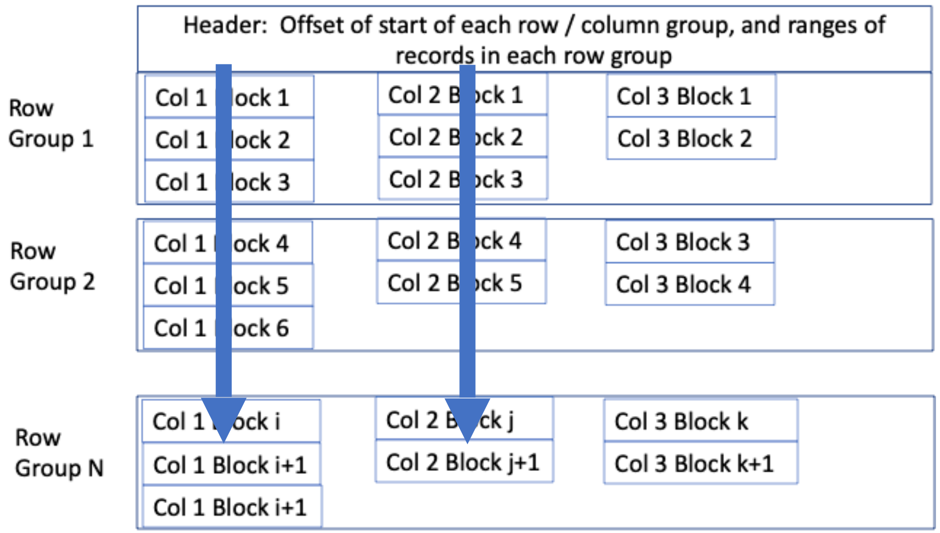
Using header, can efficiently read any subset of columns or rows without scanning whole file (unlike CSV)

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