CS 6530: Advanced Database Systems Fall 2024

Lecture 14
Row Stores vs Column Stores

Prashant Pandey

prashant.pandey@utah.edu

Acknowledgement: Slides taken from Prof. Manos Athanassoulis, Boston University

Row-stores vs. Col-Stores: How Different Are They Really?

Are column-stores really novel?

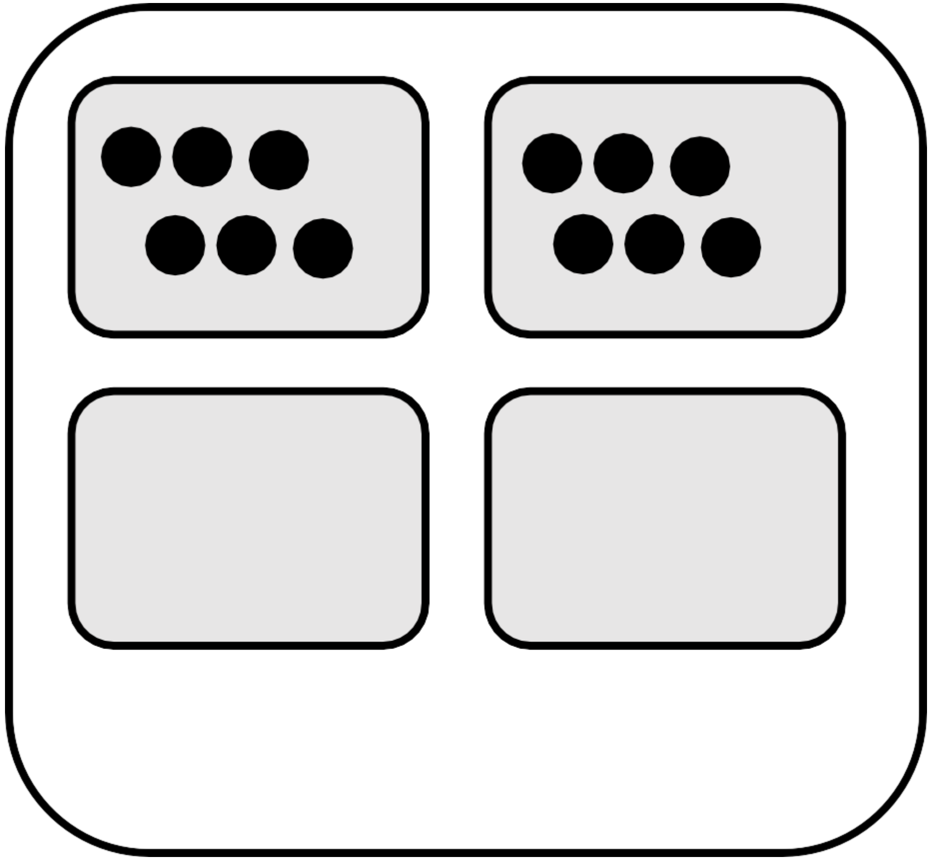
If we profile their performance, what is the breakdown? Why?

Row-Stores

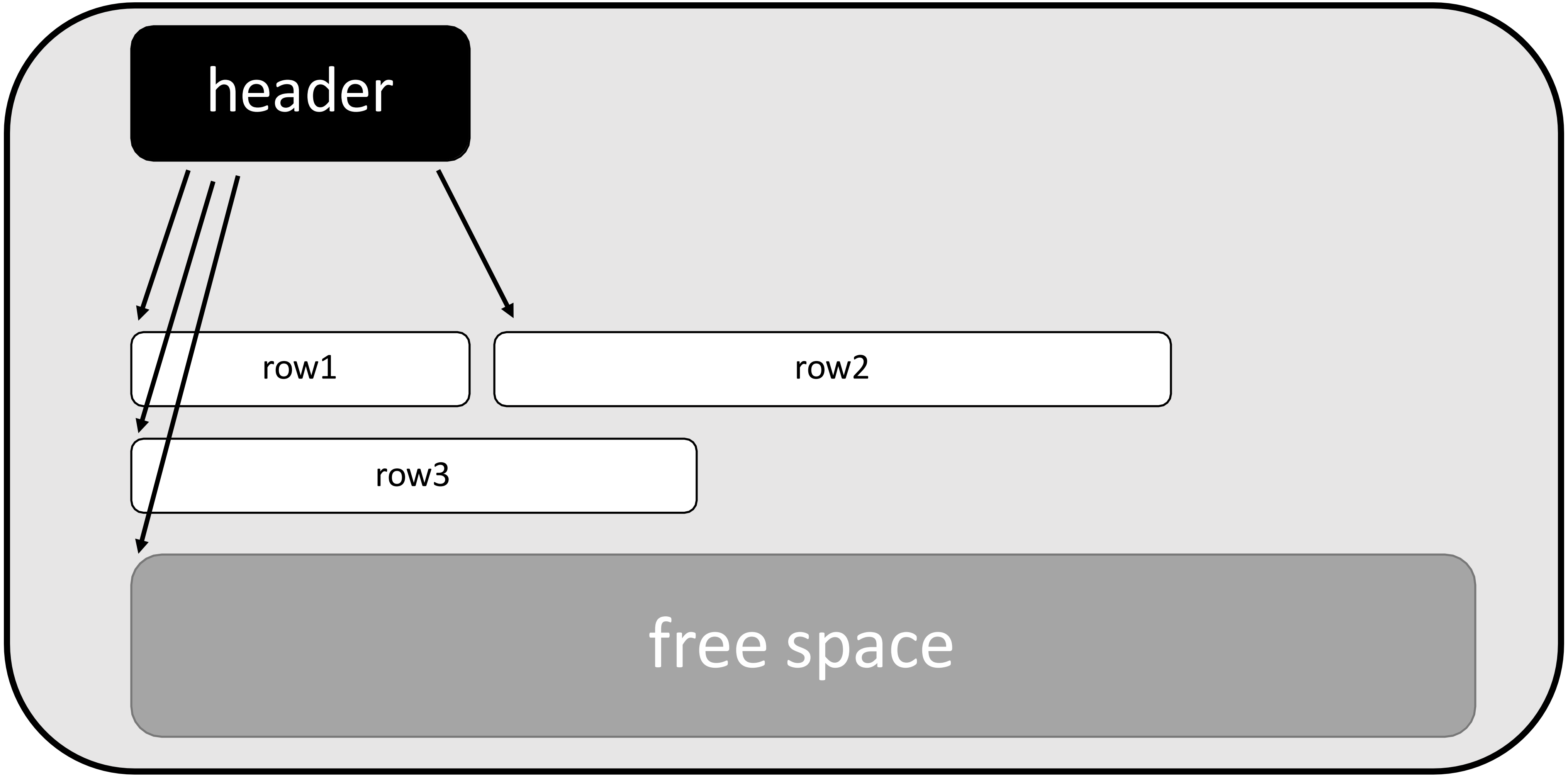
*Student (**sid**: string, **name**: string, **login**: string, **year_birth**: integer, **gpa**: real)*

student

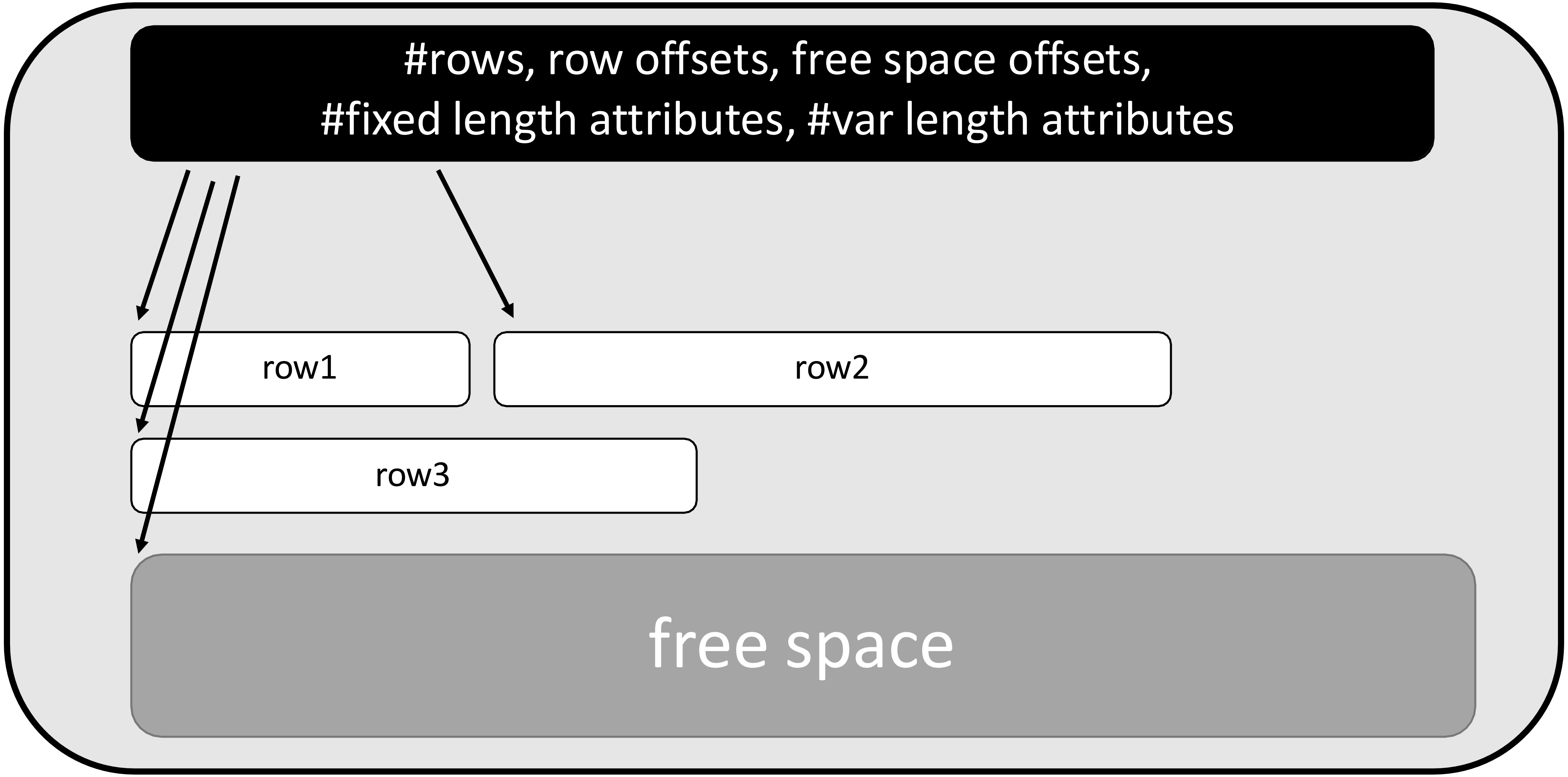
(sid1, name1, login1, year1, gpa1)
(sid2, name2, login2, year2, gpa2)
(sid3, name3, login3, year3, gpa3)
(sid4, name4, login4, year4, gpa4)
(sid5, name5, login5, year5, gpa5)
(sid6, name6, login6, year6, gpa6)
(sid7, name7, login7, year7, gpa7)
(sid8, name8, login8, year8, gpa8)
(sid9, name9, login9, year9, gpa9)



Row-Stores: slotted page



Row-Stores: slotted page

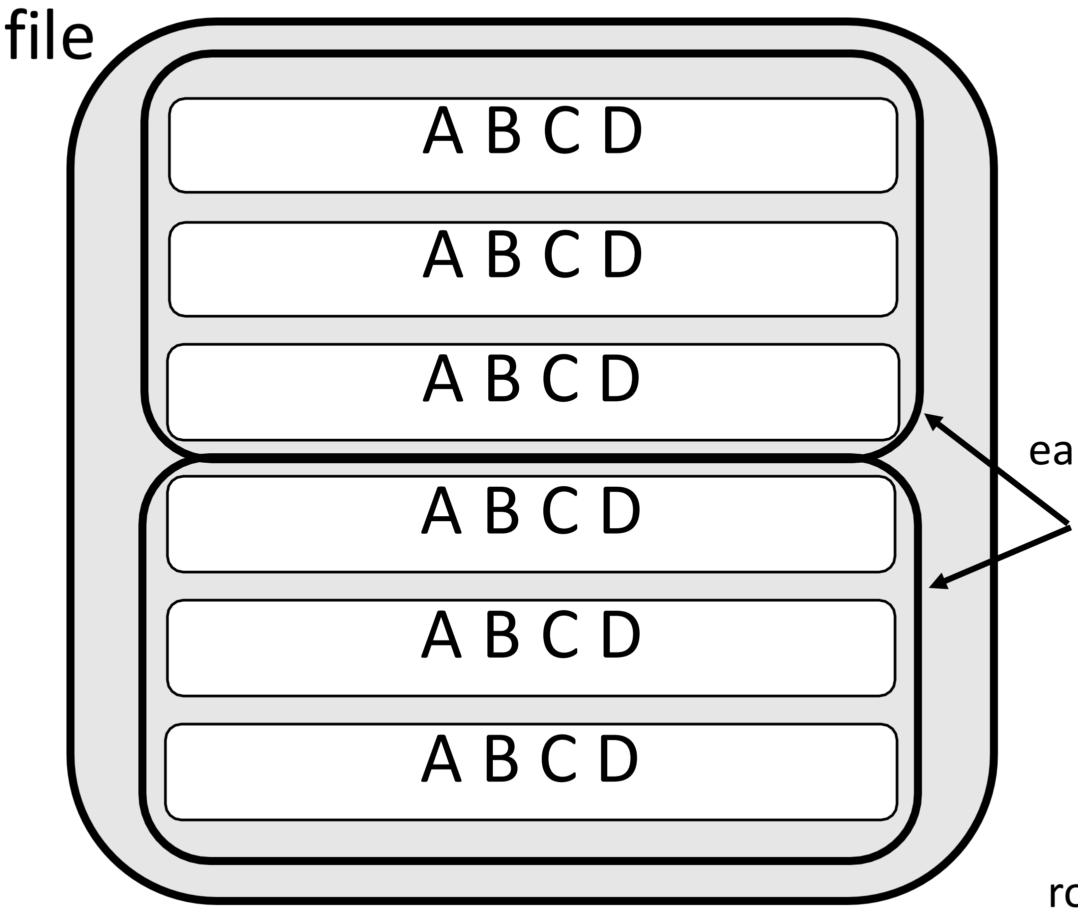


Row-Stores

pros and cons?

✅ Easy to add a new record

❌ Might access unnecessary data



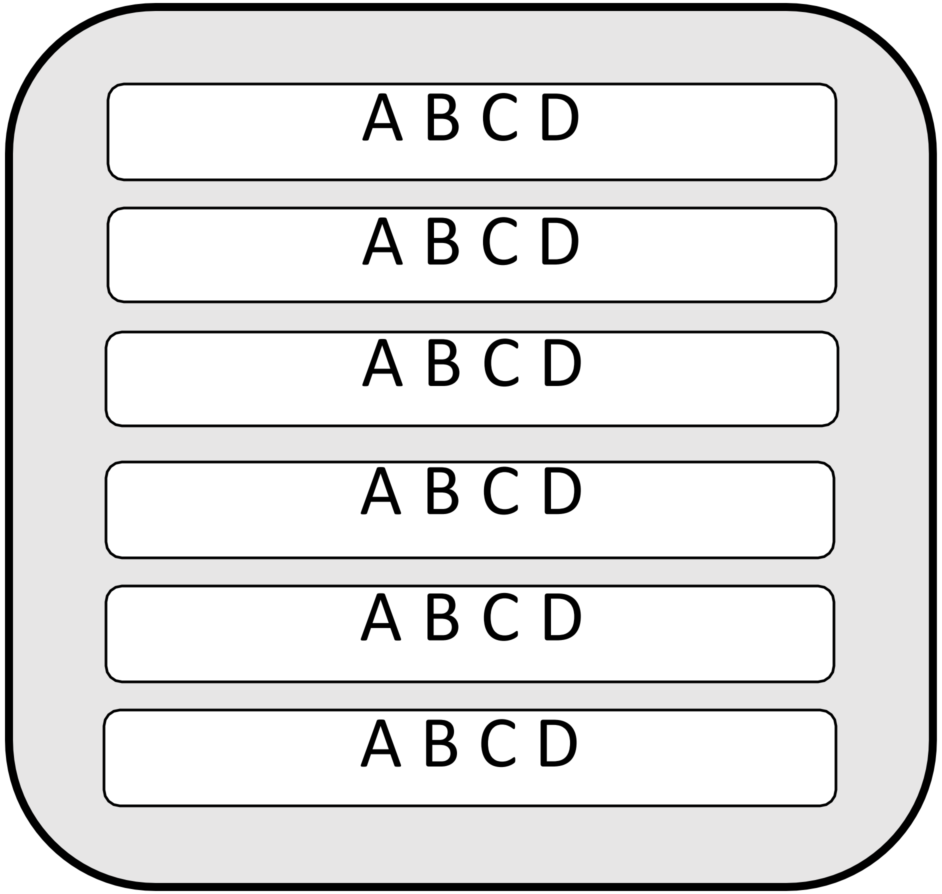
each page contains **entire** rows (all their columns)

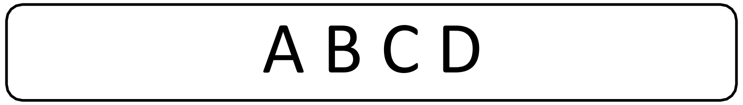
pages

rows are **contiguous**
(with possible free space at the end)

Row-stores: query processing

select max(B) **from** R **where** A>5 **and** C<10

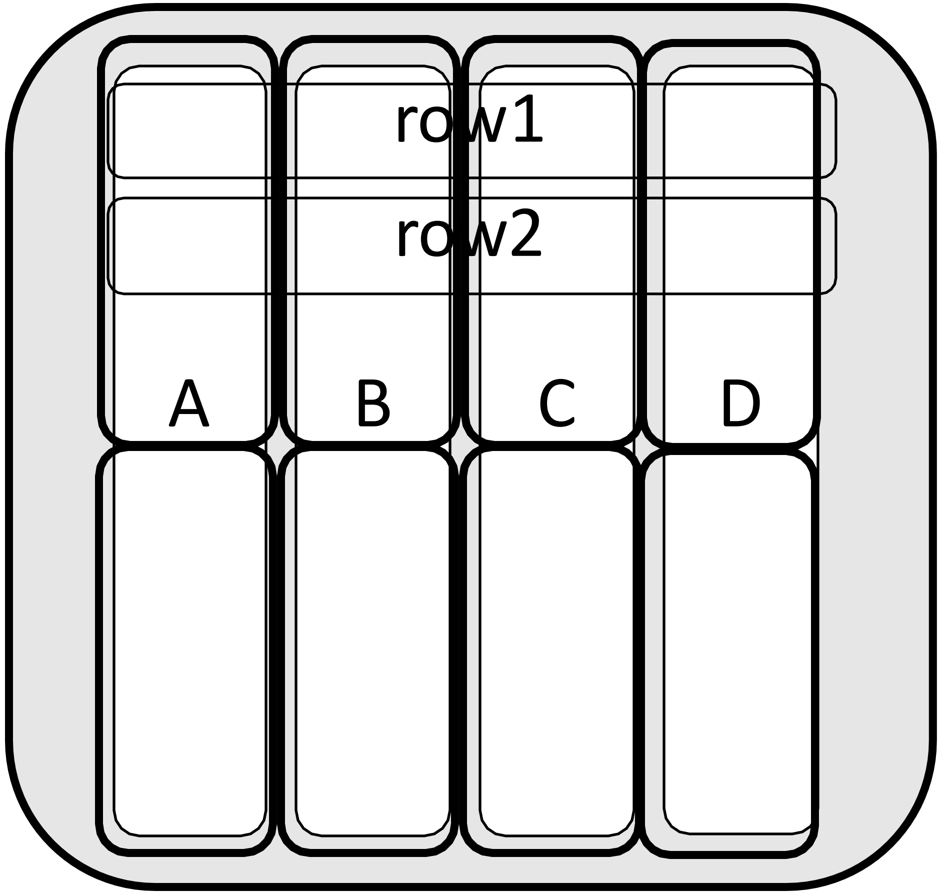




one row at a time

Column-Stores

pros and cons?



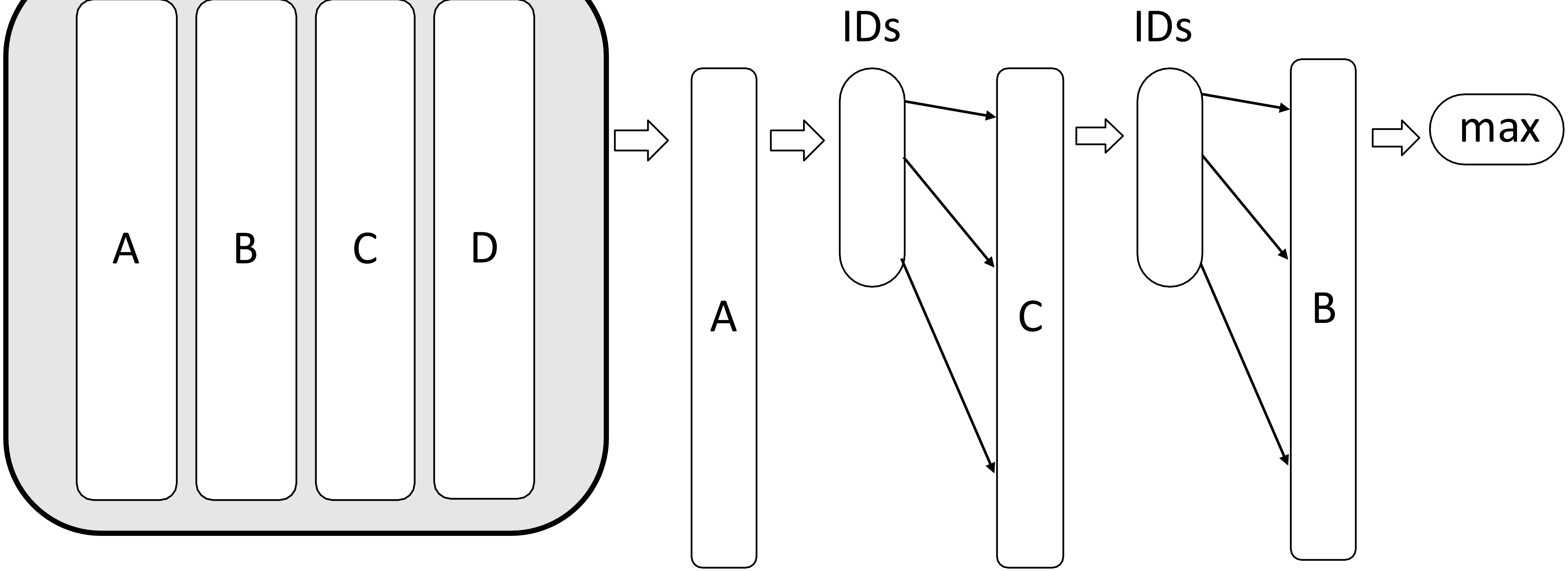
each page contains **columns!**

✅ Read only relevant data

❌ Tuple writes require multiple accesses

Column-stores: query processing

select max(B) **from** R **where** A>5 **and** C<10



Let's revisit the main question

There several studies showing

column-stores outperforming row-stores (~5x better performance in TPCH)

especially for

read-mostly data warehouses that have

1. column scans and aggregations
2. few and batched writes

Key question:

(a) are the benefits **inherent to the new column-store design**, or

(b)a **row-store with a "more columnar"** physical design can **achieve the same**?

In other words: ***can you "simulate a col-store in a row-store?"***

State-of-the-art Col-Store features

Late Materialization

“stich the column together as late as possible”

Block iteration

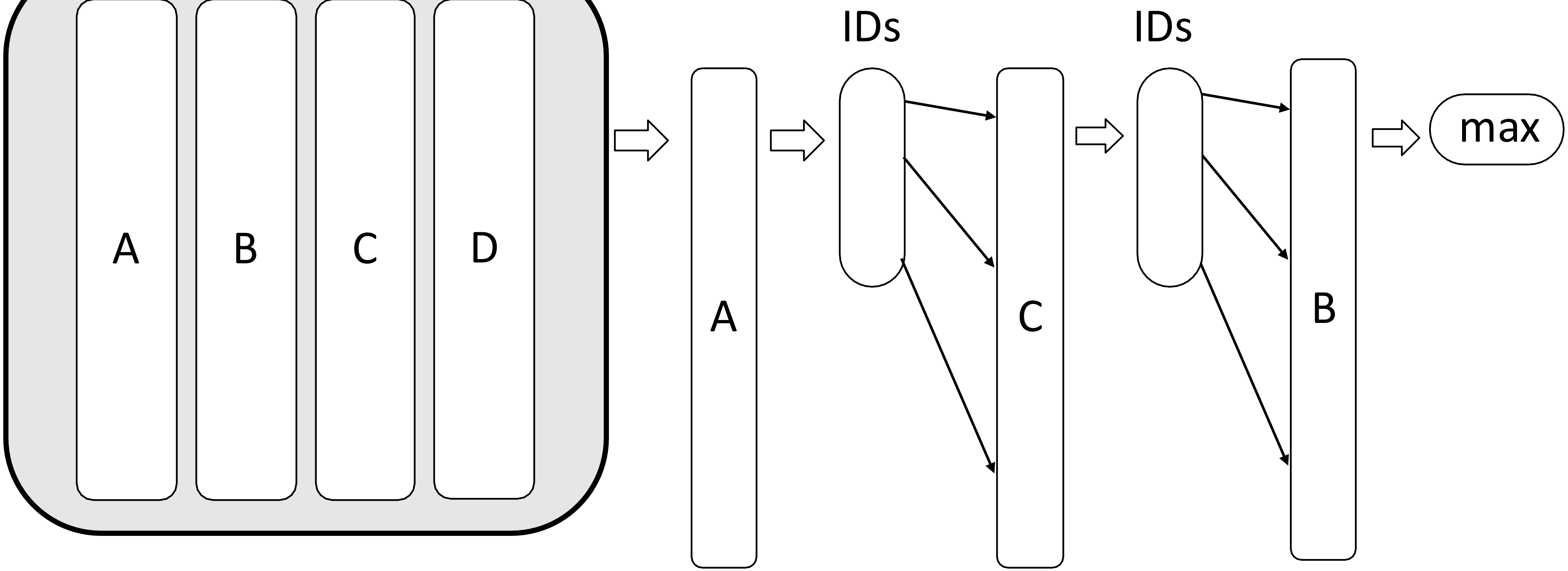
“execute the same columnar operation over a block of values”

Compression

“column-specific compression, due to the nature of data”

Late Materialization

select max(B) **from** R **where** A>5 **and** C<10

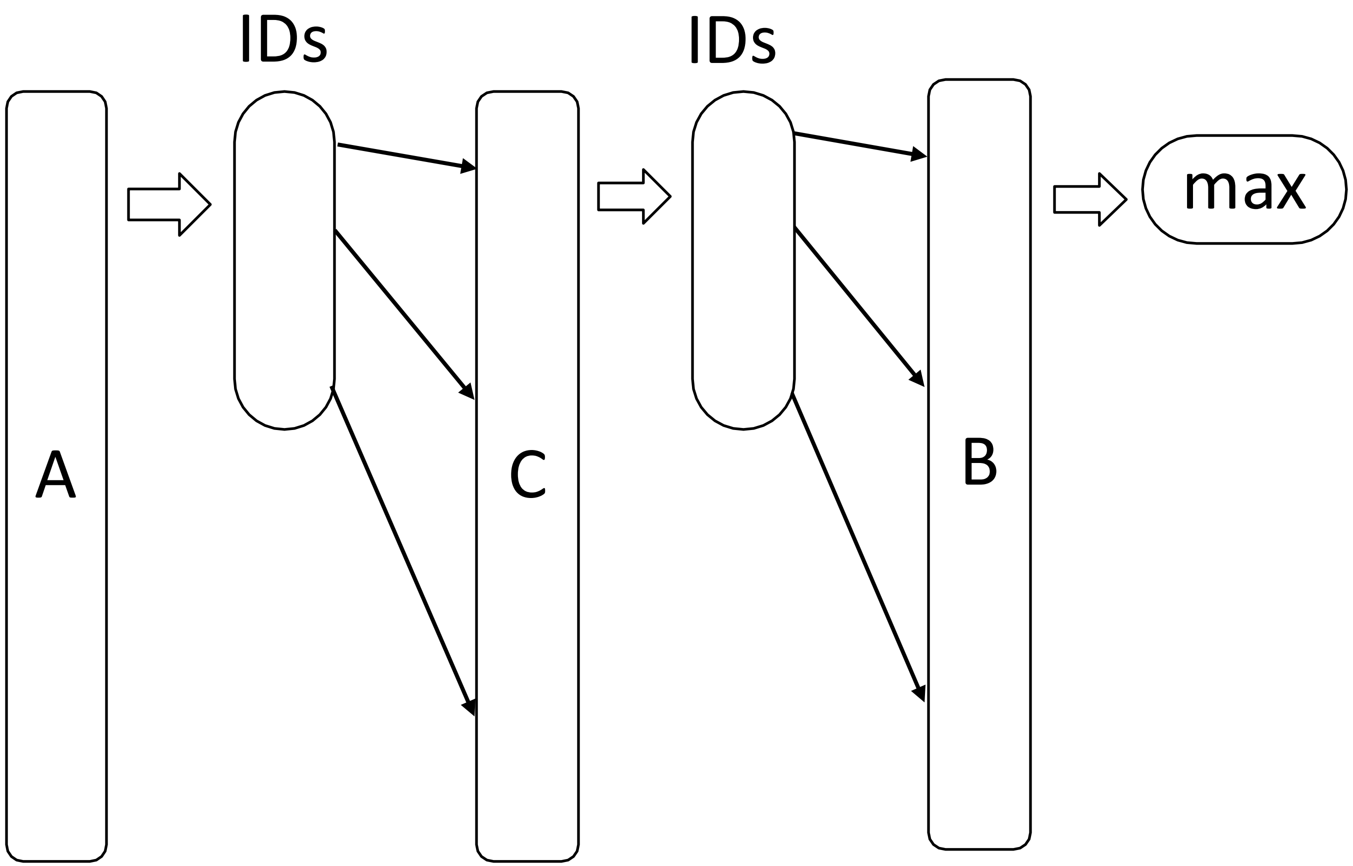


“the full tuple (or the necessary subset) is not materialized until it is needed”

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

“Column-at-a-time”

select max(B) **from** R **where** A>5 **and** C<10



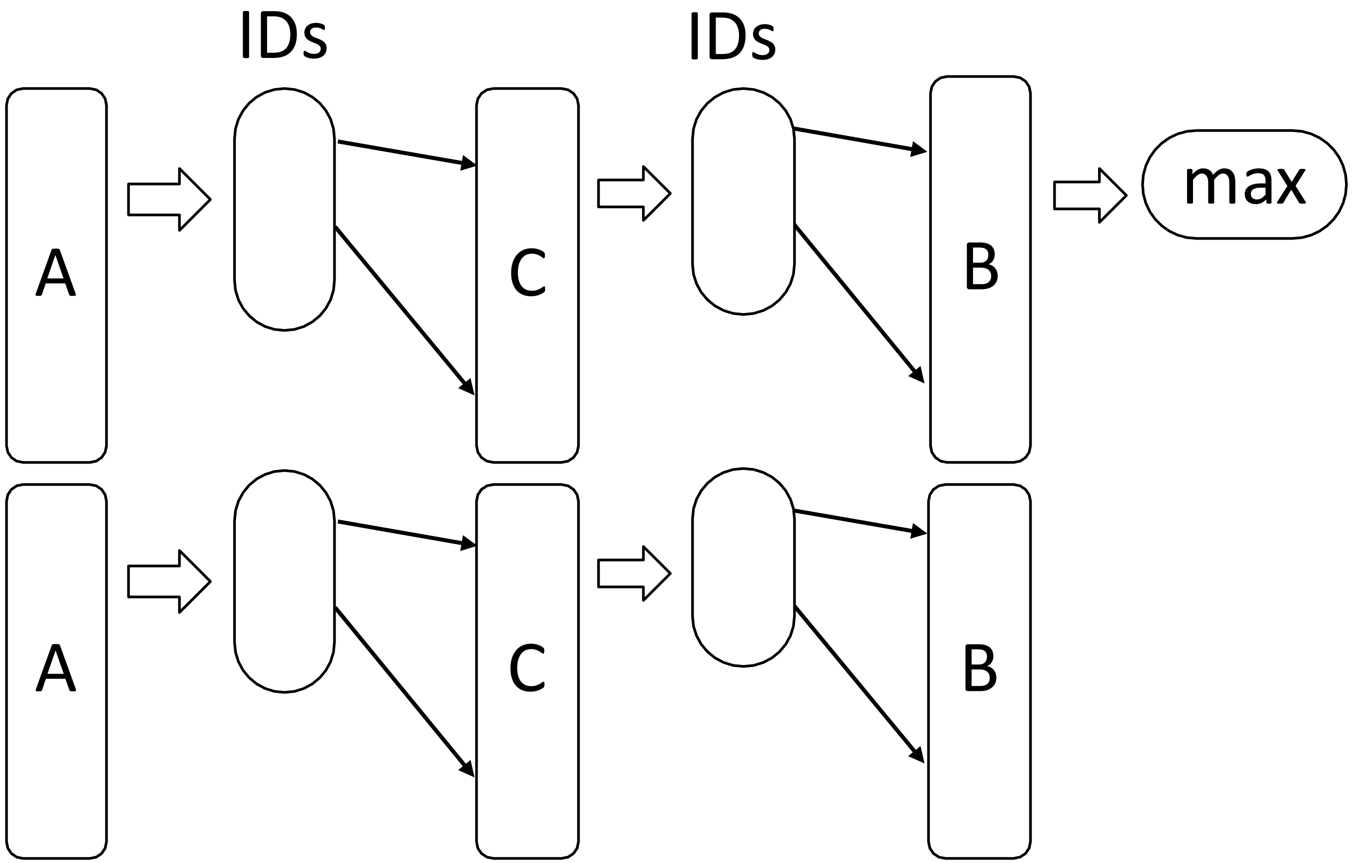
whole column?

column at a time

block/vector at a time

Block Iteration

select max(B) **from** R **where** A>5 **and** C<10



whole column?

column at a time

block/vector at a time

What is easier to compress?

#1, John, 2/4/88, Boston

#2, Joe, 2/1/87, New York

#3, Lina, 7/7/93, Boston

#4, Anna, 4/1/92, Chicago

#5, Tim, 3/9/91, Seattle

#6, Rose, 9/3/96, Boston

#1	John	2/4/88	Boston
#2	Joe	2/1/87	New York
#3	Lina	7/7/93	Boston
#4	Anna	4/1/92	Chicago
#5	Tim	3/9/91	Seattle
#6	Rose	9/3/96	Boston

exploit patterns, duplicates, small differences

How to simulate a col-store with a row-store?

Vertical Partitioning

“physically partition the data per column”

Index-only Plans

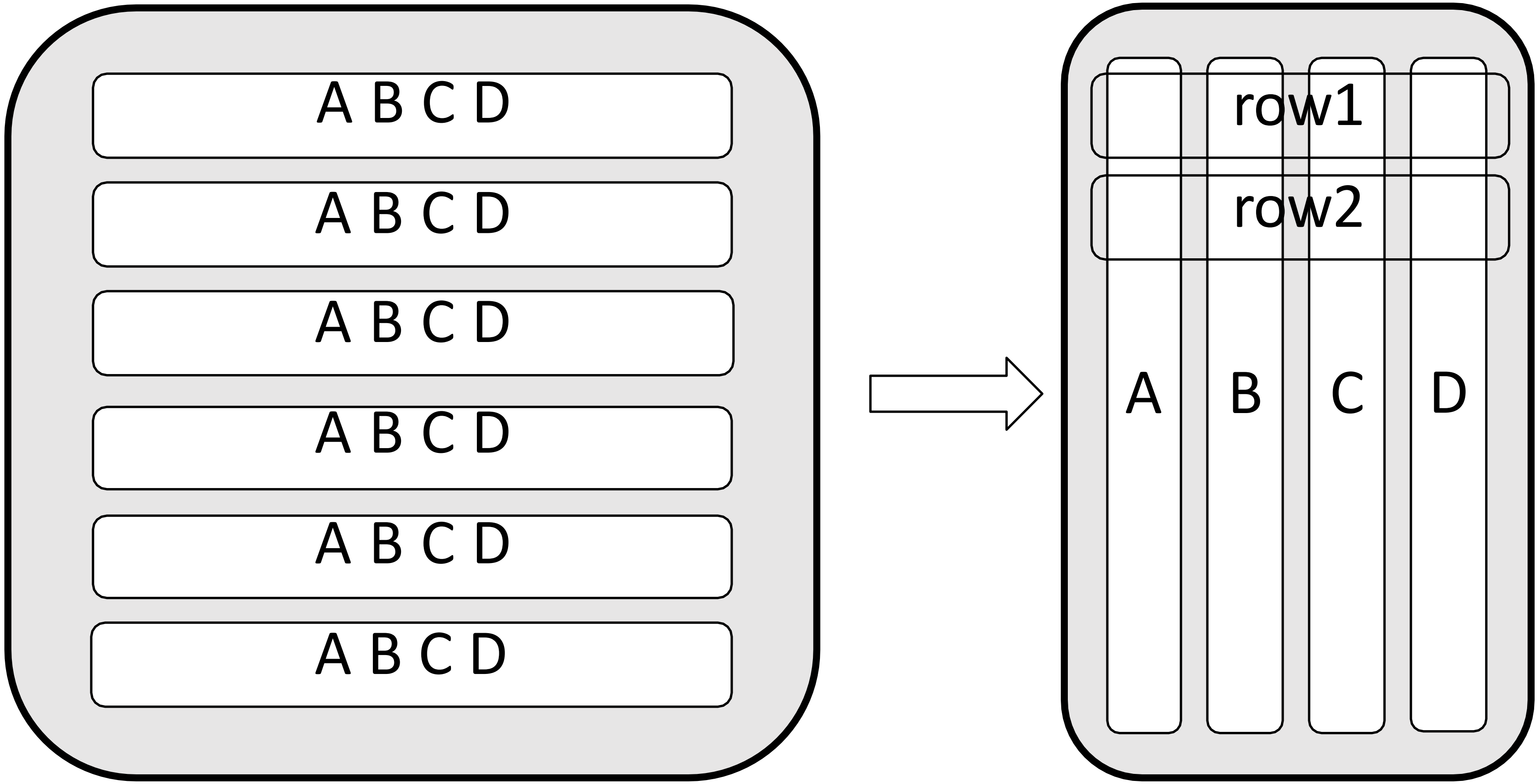
“use only indexes in query plans that contain only relevant columns”

Materialized Views

“temporary tables that contain exactly the answer to a query”

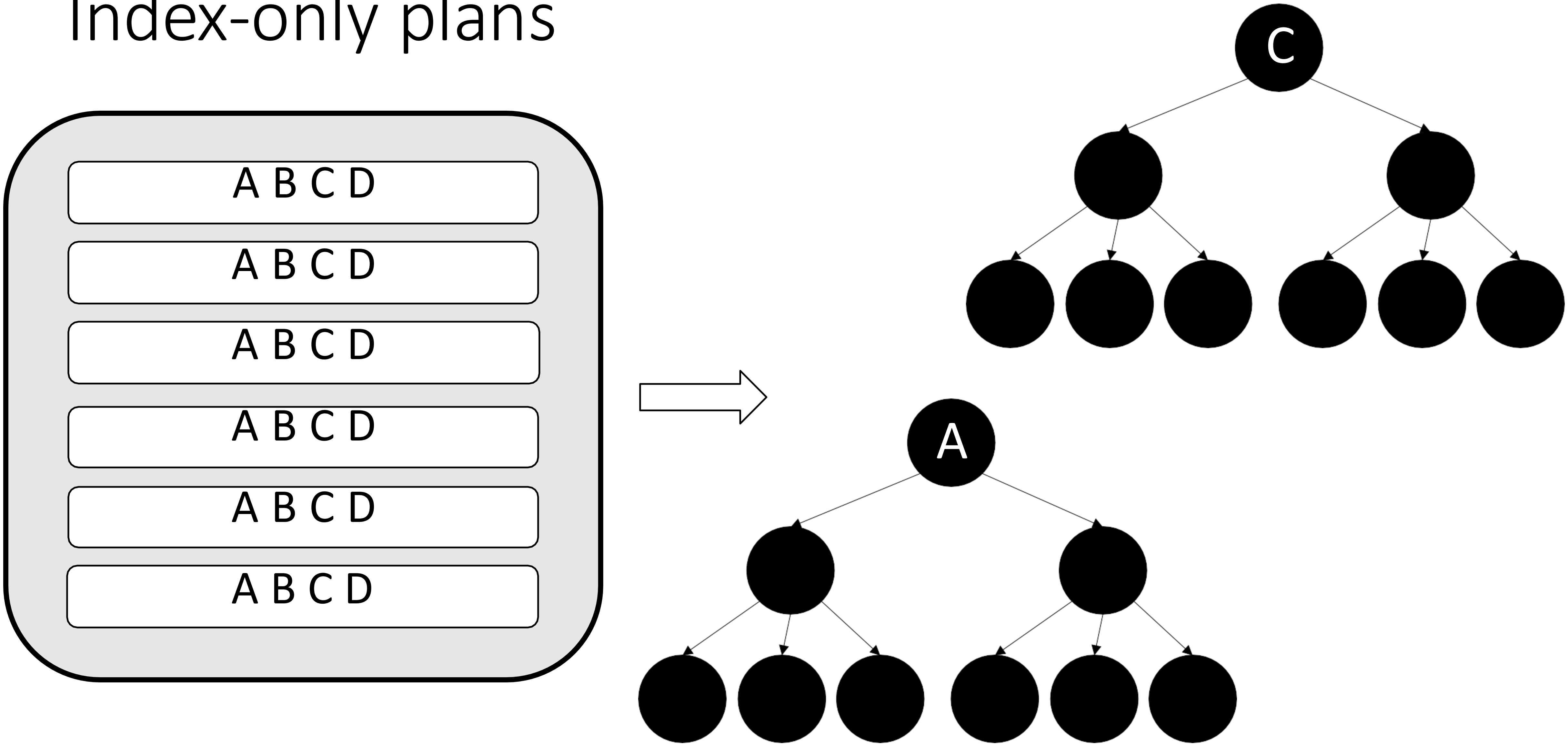
Vertical Partitioning

select max(B) **from** R **where** A>5 **and** C<10



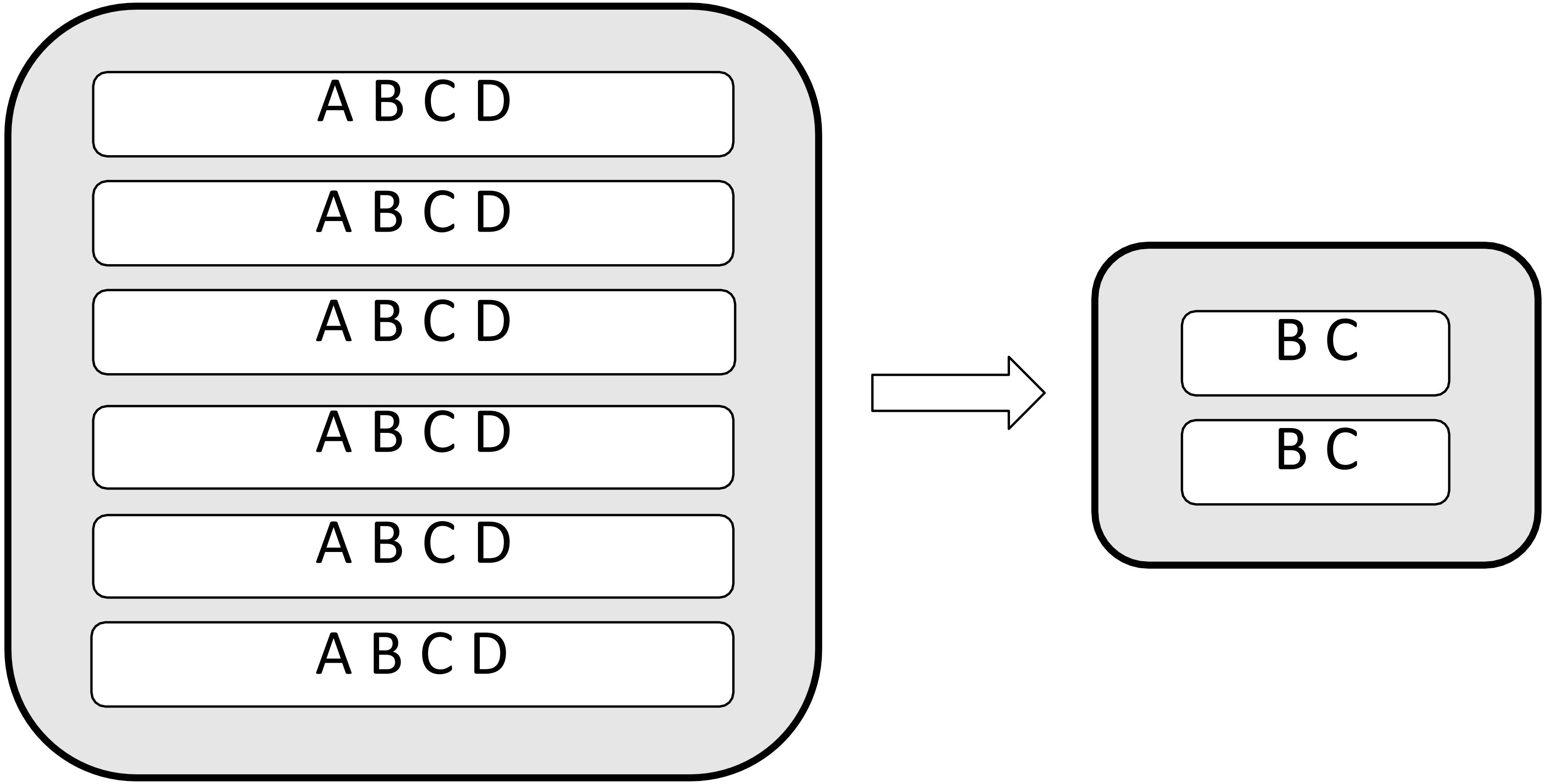
select max(B) **from** R **where** A>5 **and** C<10

Index-only plans



Materialized Views

select B, C **from** R **where** A>5 **and** C<10



Benchmarking

When comparing database systems we need a common “language”

Benchmarks from the ***Transaction Performance Council***

TPC-B, TPC-C, TPC-H, TPC-DS etc

Also, a benchmark for data warehousing:

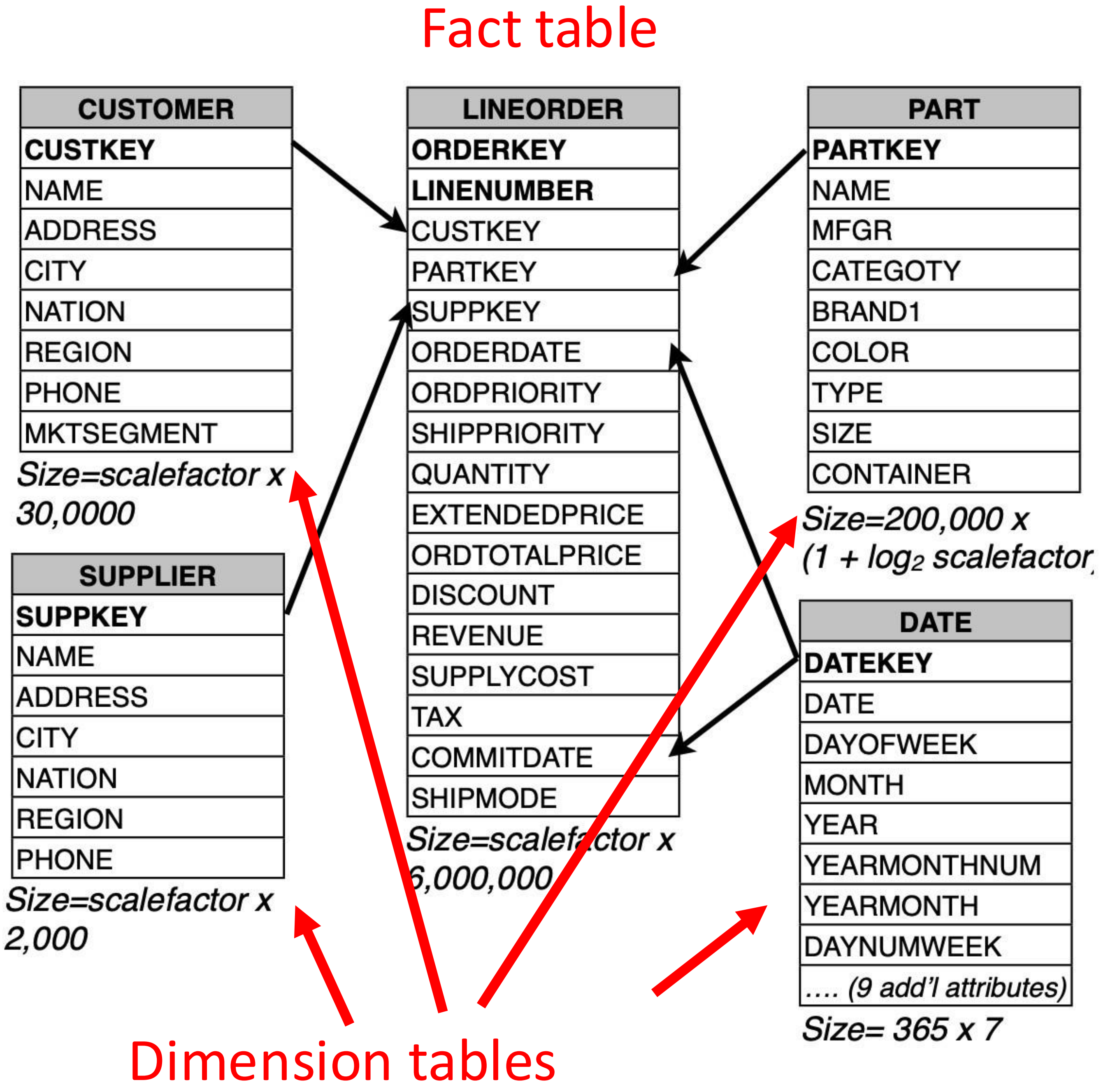
Star Schema Benchmark

Star-Schema Benchmark

13 queries

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey and
      d_year = 1993 and
      lo_discount between 1 and 3 and
      lo_quantity < 25;
```

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_category = 'MFGR#12' and
      s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```



Experiments

1 CPU 2.8GHz, 3GB RAM, Red Hat Linux 5

4-disk HDD array with 160-200MB/s aggregate bandwidth

(older paper, so small numbers!)

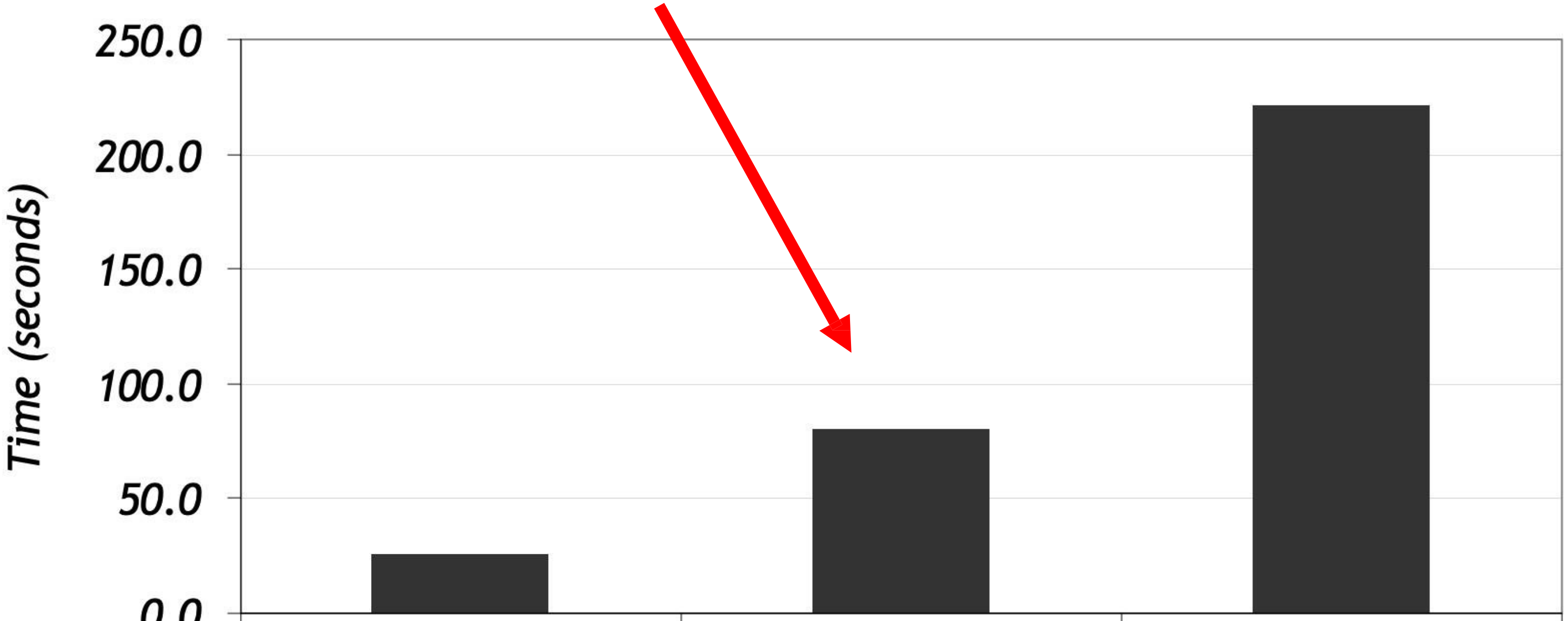
Report averages with “warm” bufferpool (smaller than data size)

Focus on SSB averages (the paper has more detailed graphs)

Experimenting with row-stores (SSB averages)

tuple overheads (additional record IDs)
+ could not horizontally partition + more expensive hash joins

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_category = 'MFGR#12' and
      s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```



	Normal Row-Store	Vertically Partitioned Row-Store	Row-Store With All Indexes
Average	25.7	79.9	221.2

Details on Vertical Partitioning

TID	Column Data
1	
2	
3	

TID	Column Data
1	
2	
3	

Tuple Header	TID	Column Data
	1	
	2	
	3	

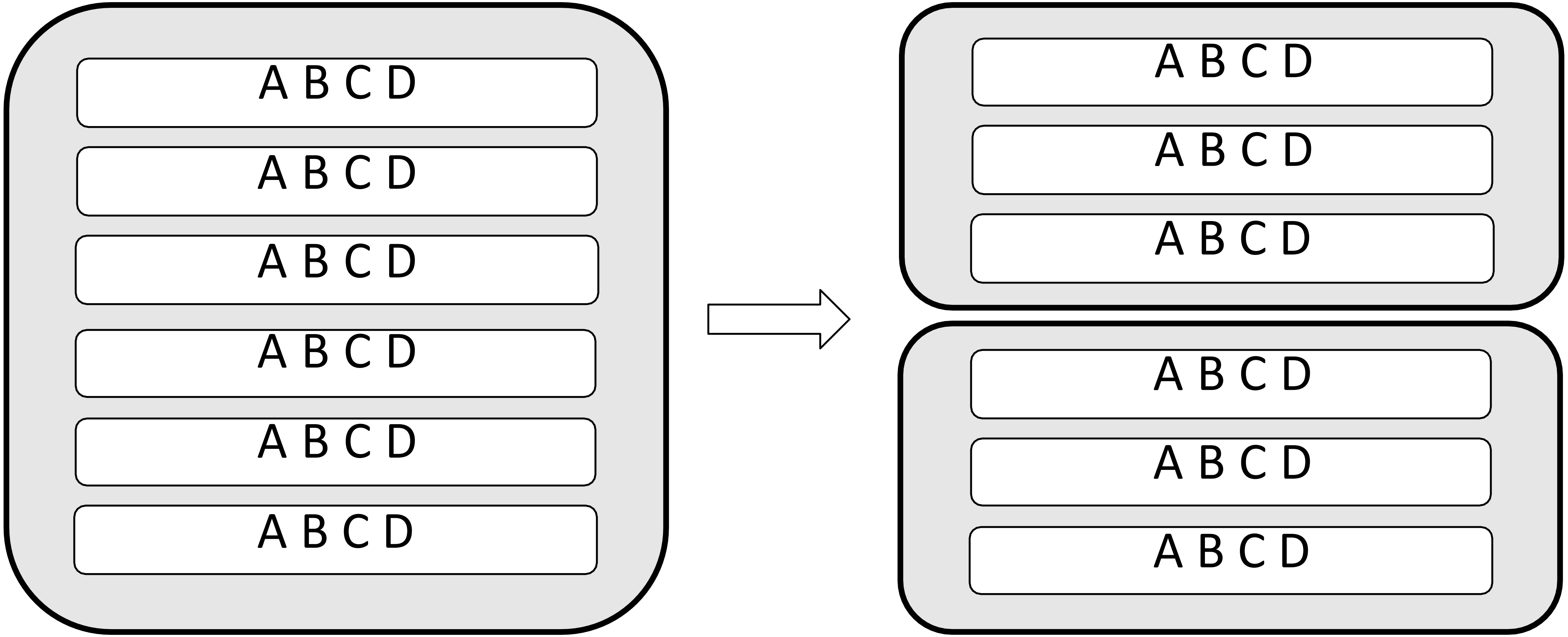
Complete fact table 4GB (compressed)

Vertical partitioned tables are 0.7-1.1GB per column (compressed)

Note that a "real column-store" would only store the raw values as an array.
In this example it would be only 240MB.

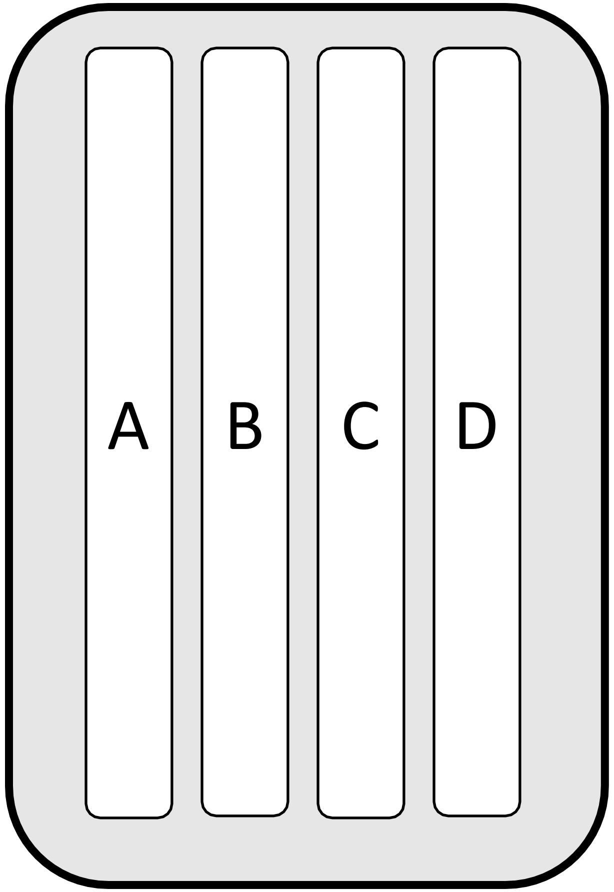
Vertical Partitioning Interferes With Horizontal Partitioning

The fact table is horizontally partitioned (on date, allows to skip lots of data)



Vertical Partitioning Interferes With Horizontal Partitioning

The fact table is horizontally partitioned (on date, allows to skip lots of data)



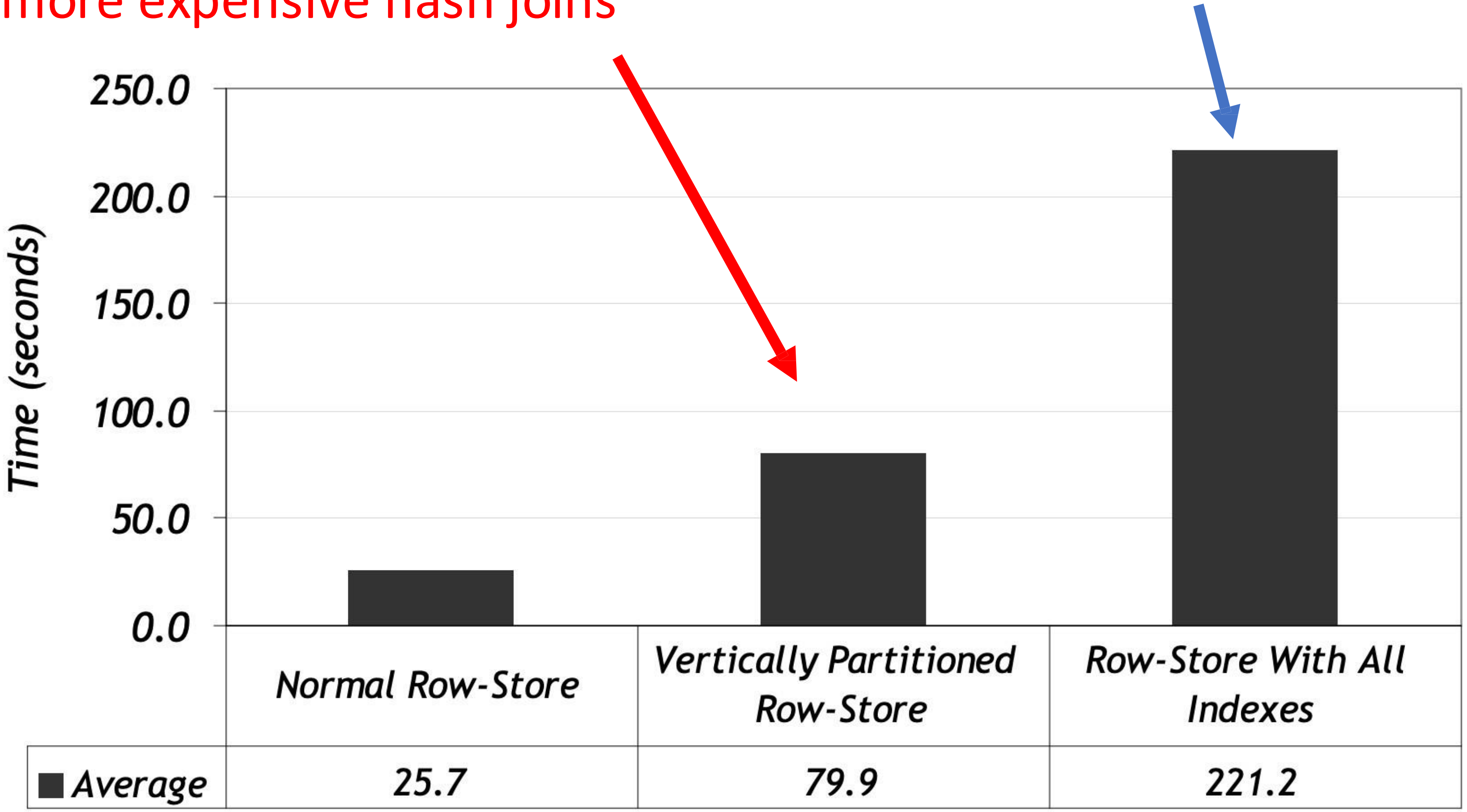
Cannot horizontally partition because the vertical partitions do not contain date info

Experimenting with row-stores (SSB averages)

tuple reconstruction (via expensive joins) prior to the join between tables

tuple overheads (additional record IDs)
+ could not horizontally partition + more expensive hash joins

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_category = 'MFGR#12' and
      s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```



Details on All Indexes

A common query pattern:

```
SELECT store_name, SUM(revenue)
FROM Facts, Stores
WHERE fact.store_id = stores.store_id AND
      stores.country = "Canada"
GROUP BY store_name
```

All qualifying tuples (based on where clause) are selected and reconstructed ("stitched together")

Note that indexes map to TIDs, and then from TIDs we get the column's value

Tuple reconstruction is SLOW!

Can we simulate a column-store with a row-store?

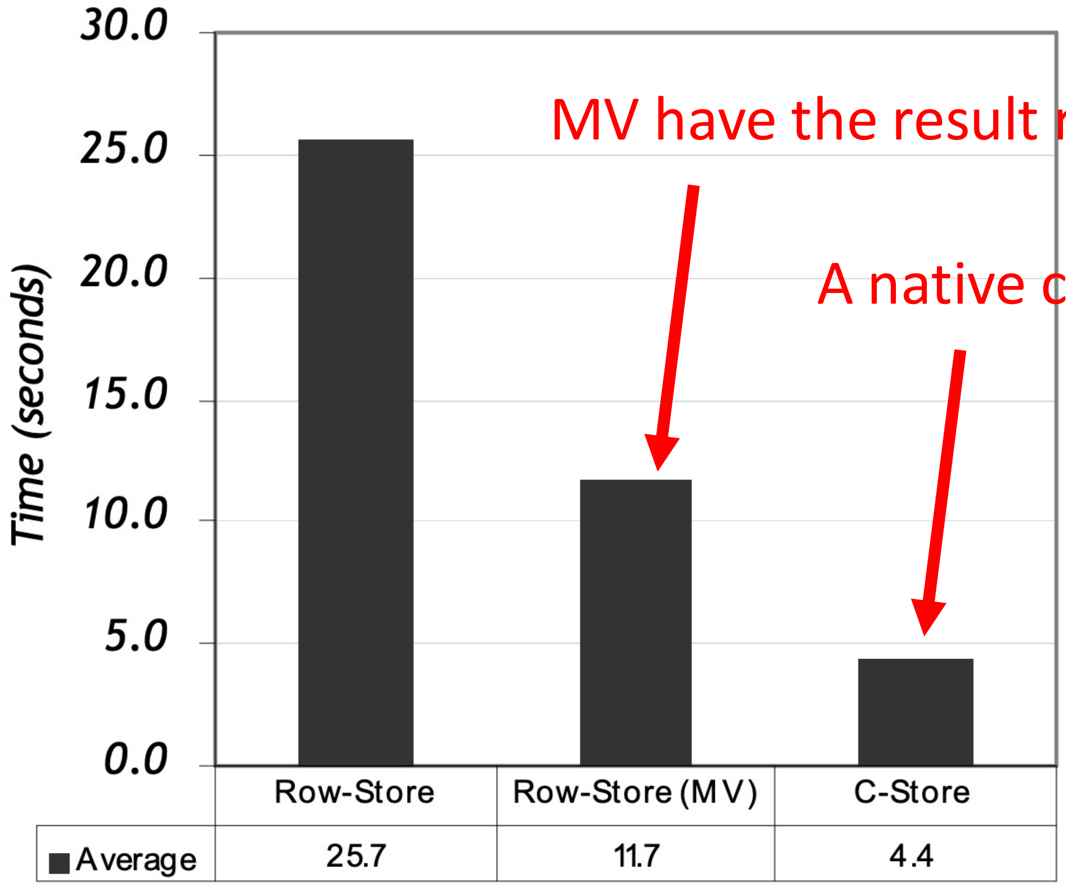
(a) *All Indexes* is a poor way to do it

(b) *Vertical Partitioning's problem are NOT fundamental*

   i. *tuple header can be removed*
   ii. *TIDs can be virtual*
   iii. *horizontal partitioning can be based on the values of a different VP*

But still, column-stores and row-stores are apples and oranges!!

Row-Stores vs. Column-Stores (SSB average)



MV have the result readily available

A native col-store is faster

Why?

Methodology

Start from a native column-store

Remove column-store-specific performance optimizations

End with a column-store with a row-oriented query engine

A. Compression

Alternative: Dictionary Compression

- Replace variable size with minimal fixed length e.g., integer

Q1
Q1
Q1
…
Q2
Q2
…

→

Run-length Encoding

Q1, 1, 300
Q2, 301, 500
Q3, 501, 550
Q4, 551, 800

Benefits of col-store compression

Reduces I/O

Can operate directly on compressed data

How?

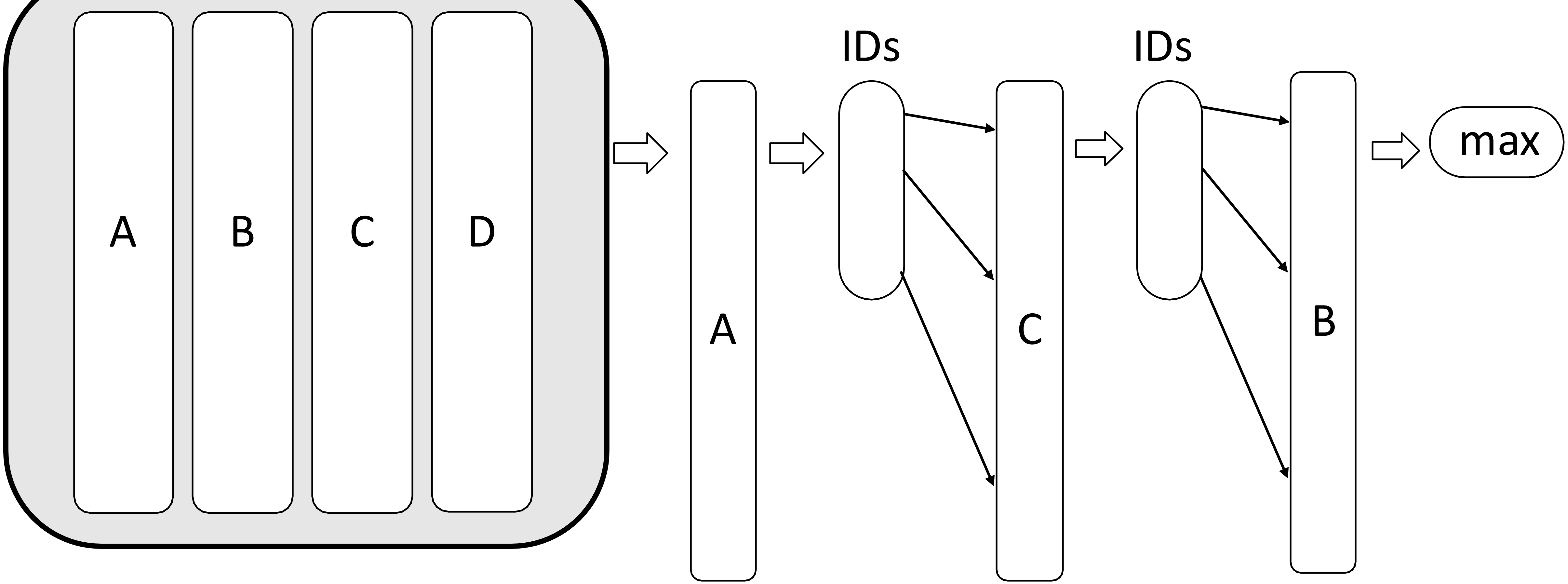
Are the same benefits applicable for row-store compression?

Reduces I/O ➔ yes, but with lower ratio (less data value locality)

No! Requires decompression before processing

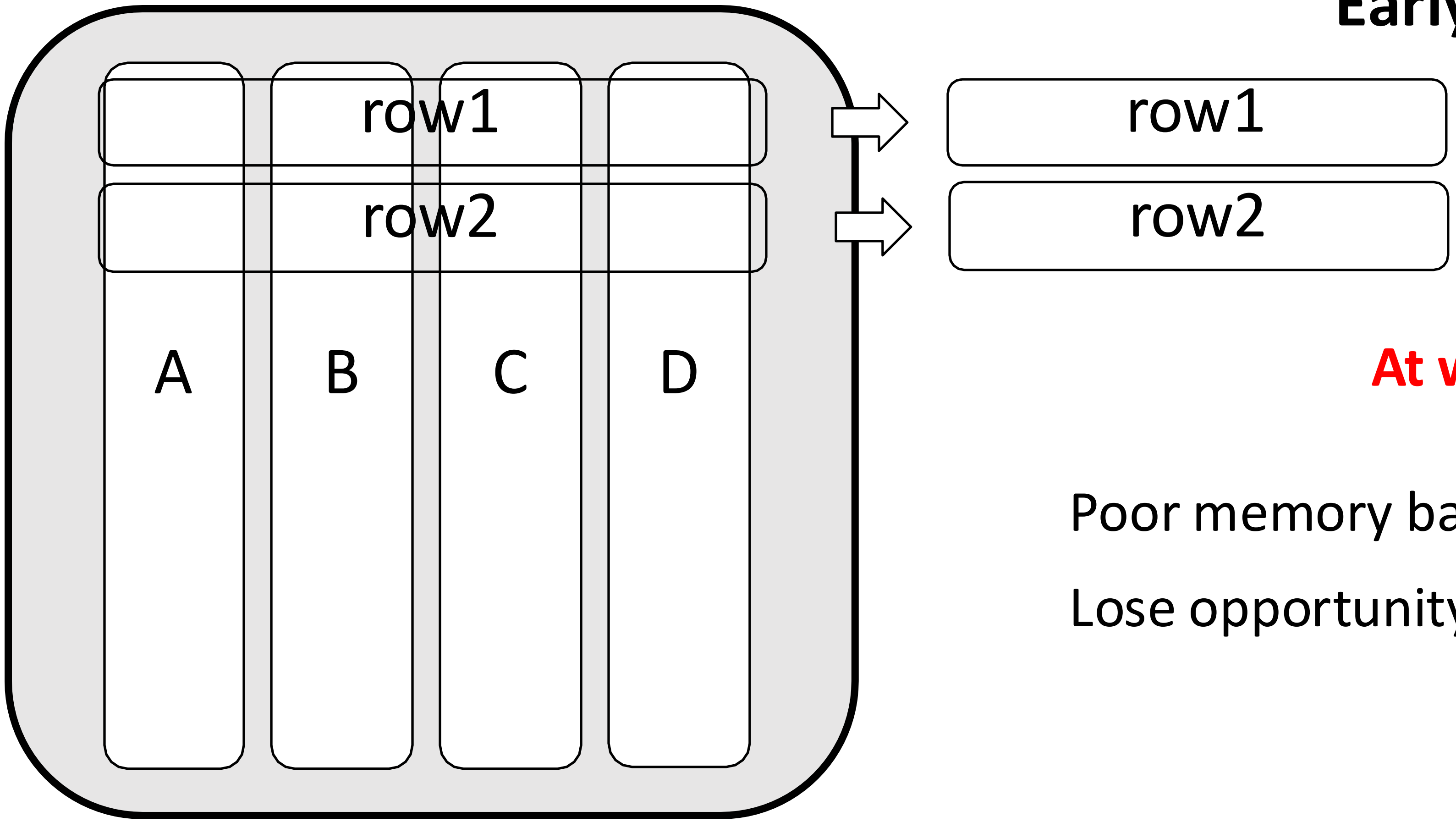
B. Early vs. Late Materialization

Late: Column-at-a-time



B. Early vs. Late Materialization

Early: Row-at-a-time



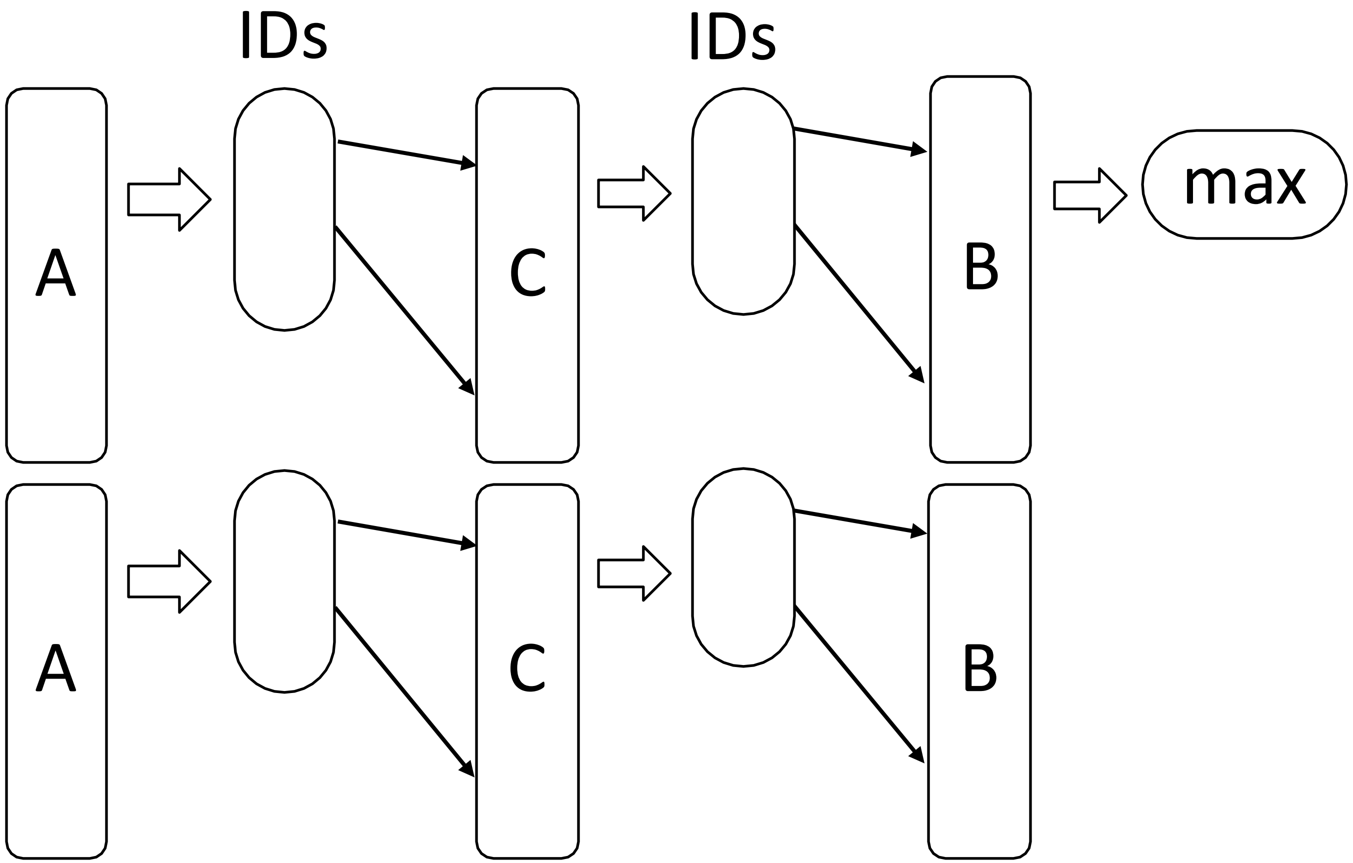
At what cost?

Poor memory bandwidth utilization

Lose opportunity for vectorized execution

C. Block Iteration

select max(B) **from** R **where** A>5 **and** C<10



whole column?

column at a time

block/vector at a time

D. Invisible Joins

Idea: rewrite joins as predicates on foreign keys in fact table

Algorithm:

1. apply each predicate to the appropriate dimension table
2. build a hash table on matching keys
3. compute bitvector with bits set for qualifying positions (tuples)
4. intersect bitvectors (positions) via bitwise AND
5. for each resulting position reconstruct the resulting tuple

1. apply each predicate to the appropriate dimension table
2. build a hash table on matching keys

Apply `region = 'Asia'` on Customer table

custkey	region	nation	...
1	Asia	China	...
2	Europe	France	...
3	Asia	India	...

→ Hash table with keys 1 and 3

Apply `region = 'Asia'` on Supplier table

suppkey	region	nation	...
1	Asia	Russia	...
2	Europe	Spain	...

→ Hash table with key 1

Apply `year in [1992,1997]` on Date table

dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

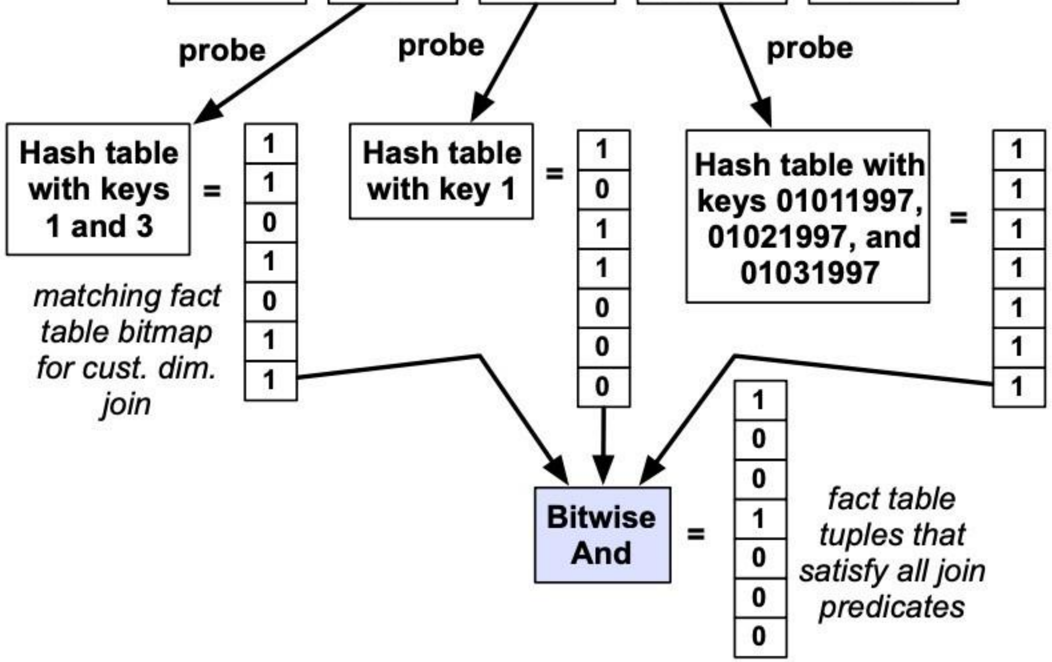
→ Hash table with keys 01011997, 01021997, and 01031997

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey AND
      lo.suppkey = s.suppkey AND
      lo.orderdate = d.datekey AND
      c.region = 'ASIA' AND s.region = 'ASIA' AND
      d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

3. compute bitvector with bits set for qualifying positions (tuples)
4. intersect bitvectors (positions) via bitwise AND

Fact Table

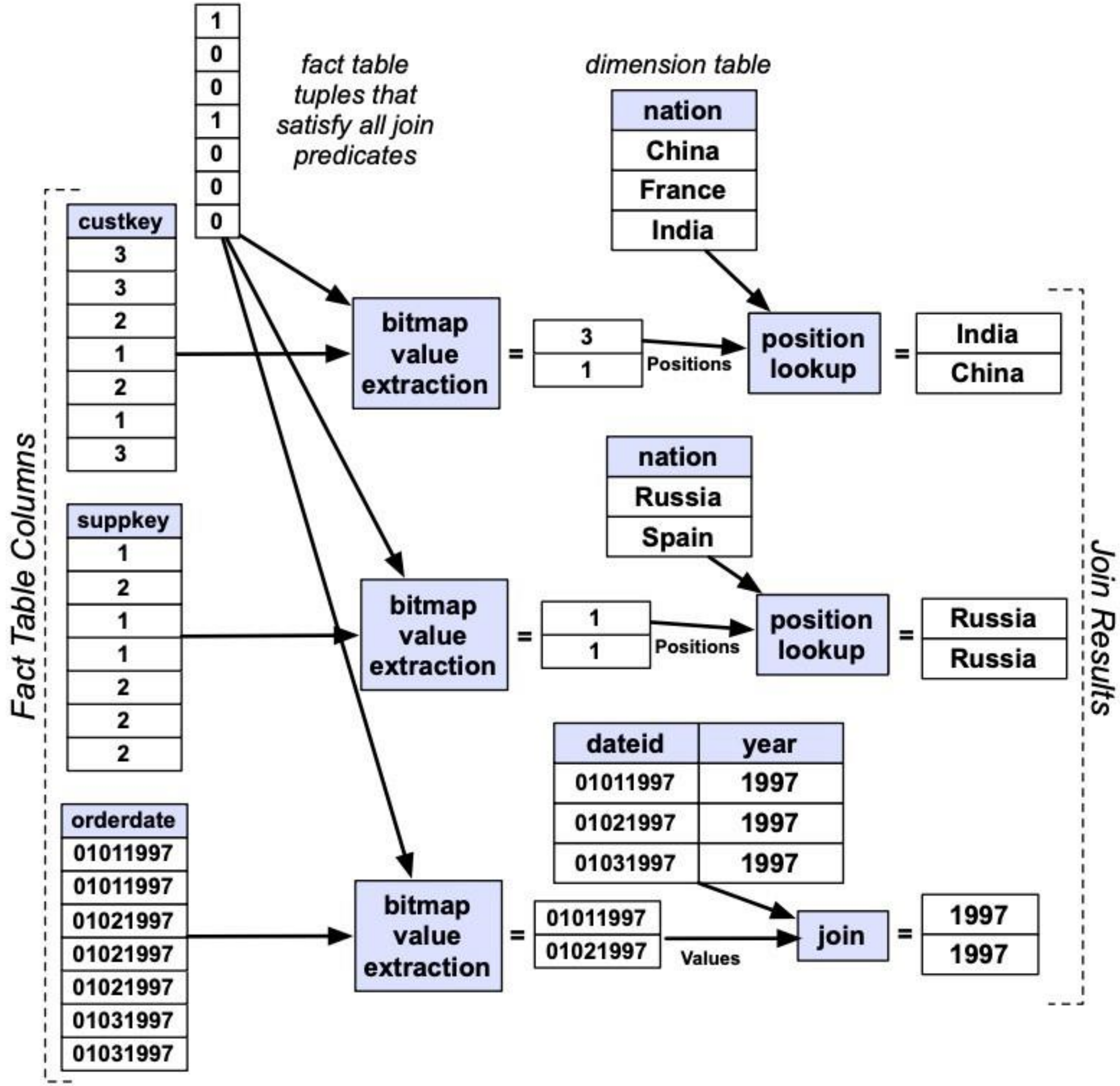
orderkey	custkey	suppkey	orderdate	revenue
1	3	1	01011997	43256
2	3	2	01011997	33333
3	2	1	01021997	12121
4	1	1	01021997	23233
5	2	2	01021997	45456
6	1	2	01031997	43251
7	3	2	01031997	34235



5. For each resulting position, extract the values from the columns that are in the result

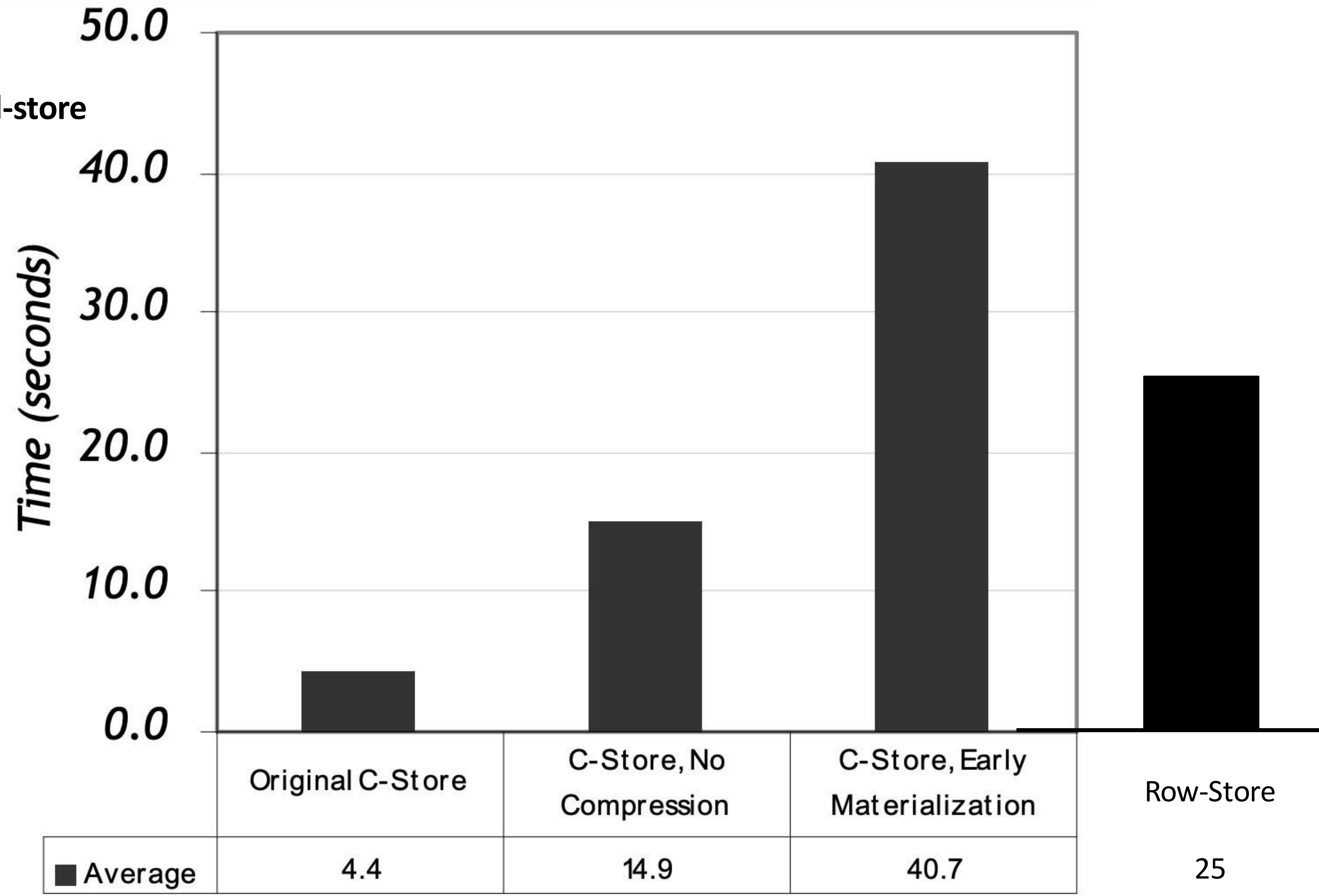
Are invisible joins a general join algorithm?

No! It works only for Star Schemas



To make the most of a col-store

1. Efficient support of vertical partitioning (compression)
2. Column-specific execution (late materialization)



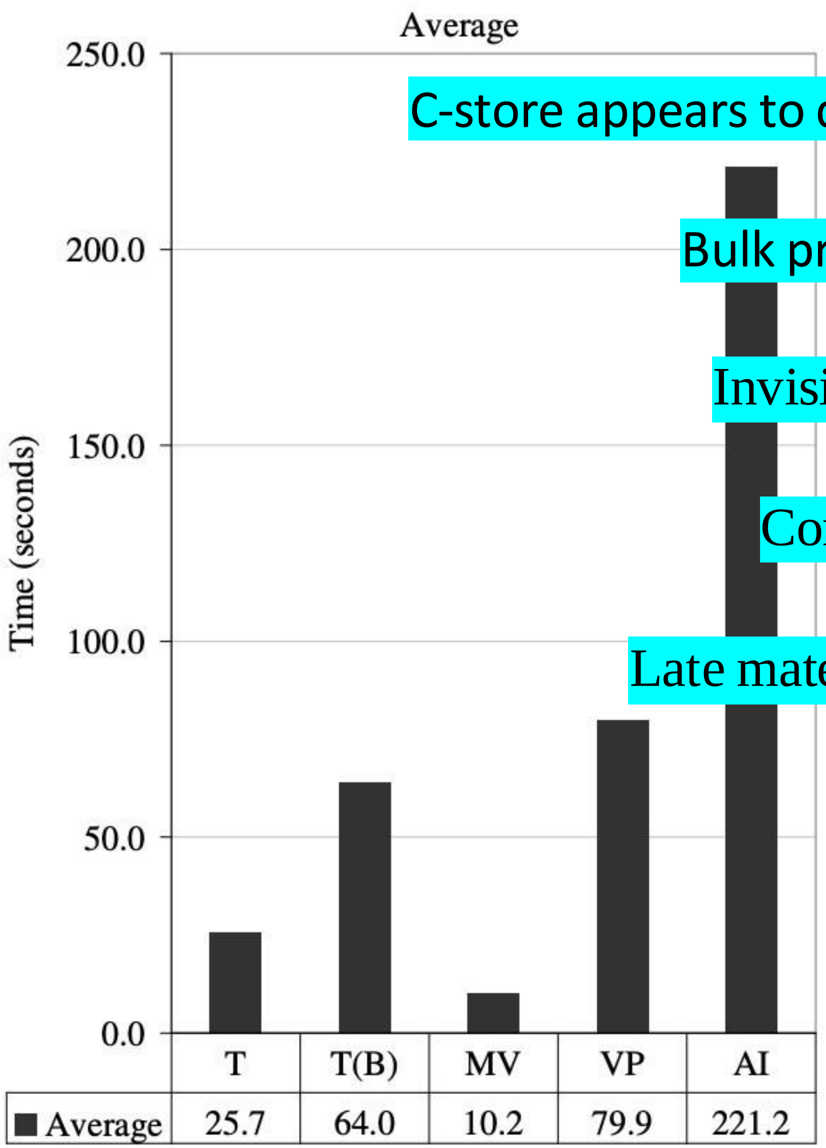
C-store appears to do even better than fully materialized joins

Bulk processing buys you 5 to 50%

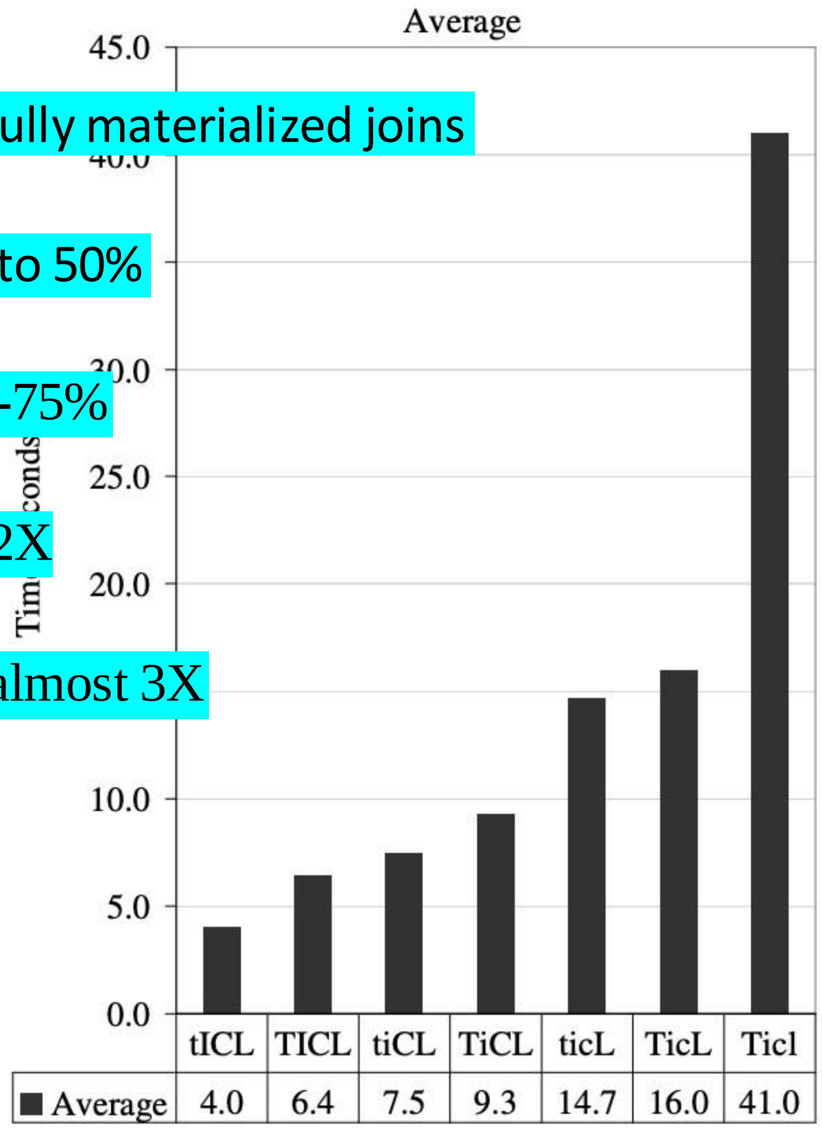
Invisible join buys you 50-75%

Compression buys you 2X

Late materialization gets you almost 3X



T is traditional, T(B) is traditional (bitmap),
MV is materialized views, VP is vertical partitioning,
and AI is all indexes



T=tuple-at-a-time processing, t=block processing;
I=invisible join enabled, i=disabled;
C=compression enabled, c=disabled;
L=late materialization enabled, l=disabled

Things to remember

Row-stores vs. Col-stores: fundamental differences

✓ Compression

✓ Late Materialization

✓ Block Iteration

✓ Column-store-specific join optimizaitons