COSI 167A
Advanced Data Systems

Class 4

Row-stores vs. Column-stores

Prof. Subhadeep Sarkar

https://ssd-brandeis.github.io/COSI-167A/

Brandeis University

Class **logistics**

and administrivia

Project 1 (C++/Java) has been **released** (due on **Sep 20**).

C/C++ learning resources at: https://ssd-brandeis.github.io/COSI-167A/assignments/

The **second technical question** is now available on the class website (due **before the class** on **Sep 17**).

Today in COSI 127B

What's on the cards?

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

Daniel J. Abadi
Yale University
New Haven, CT, USA
dna@cs.yale.edu

Samuel R. Madden
MIT
Cambridge, MA, USA
madden@csail.mit.edu

Nabil Hachem
AvantGarde Consulting, LLC
Shrewsbury, MA, USA
nhachem@agdba.com

Column-Stores vs. **Row-Stores**

How Different Are They Really?

Discussion points:

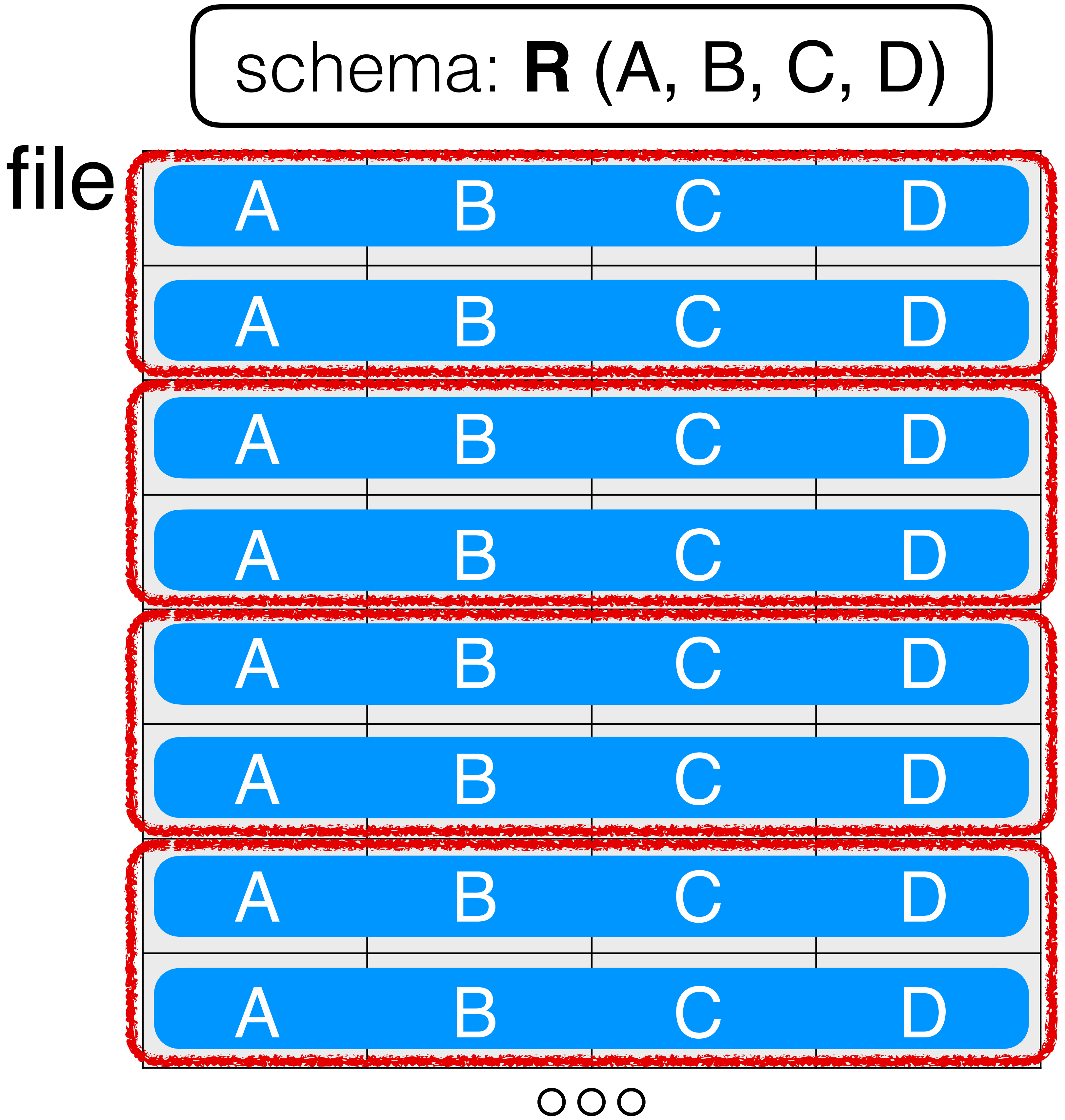
Are **column-stores** *really* **novel** implementation-wise?

Can **row-stores** be **made to act** like column-stores?

What **factors** make column-stores **special**?

Row-stores

Storing row by row!



Thought Experiment 1

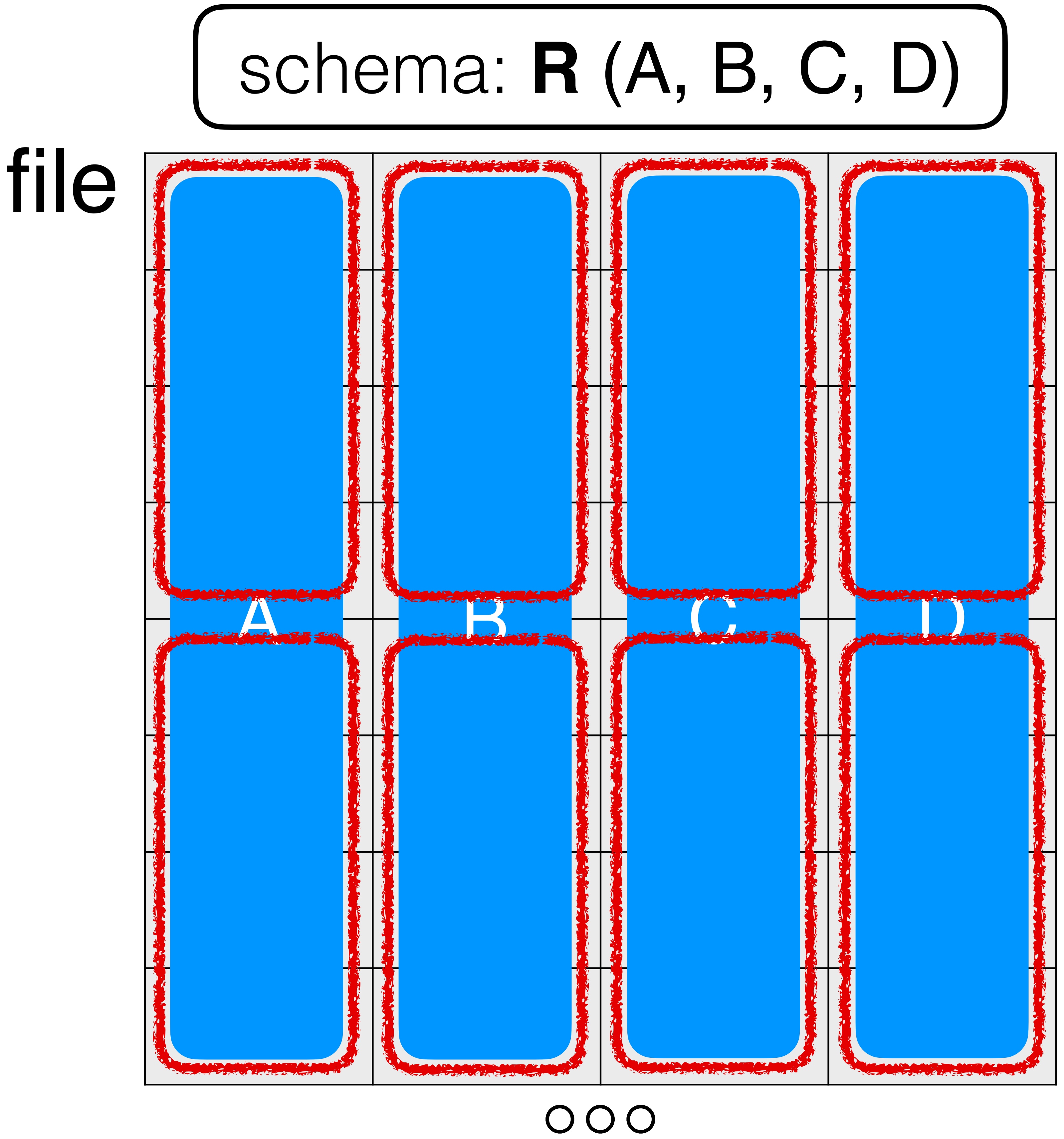
Pros & cons of **row-stores**?

- **good** for **inserts/updates**
- **good** for queries accessing **most/all columns**
- **read amplification**

Row-stores are great for transactional workloads (OLTP).

Column-stores

Storing column-wise!



Thought Experiment 2

Pros & cons of **column-stores**?

- **read necessary data** only
- **good** for **partial updates**

- **inserts** are **costly**
- **tuple reconstruction** overhead

Column-stores are great for analytical workloads (OLAP).

Goal of the paper

Dissecting row-stores and column-stores

Motivation: Prior to this paper, several studies highlighted column-stores performing ~5x better than row-stores

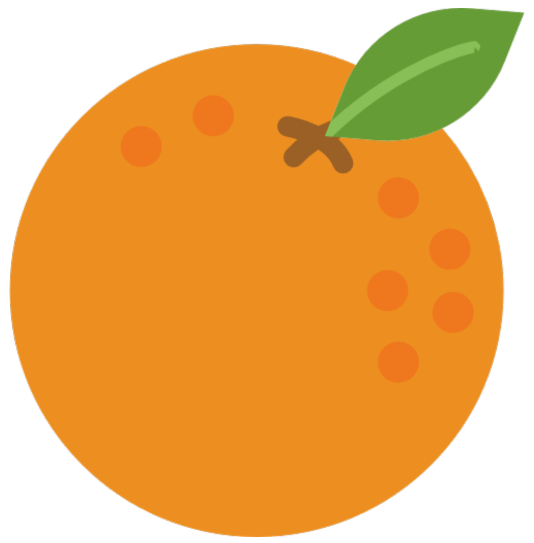
Goal: Compare row-stores and column-stores

Goal of the paper

Dissecting row-stores and column-stores

Motivation: Prior to this paper, several studies highlighted column-stores performing ~5x better than row-stores

Goal: ~~Compare row-stores and column-stores~~

 vs. 

Goal of the paper

Dissecting row-stores and column-stores

Motivation: Prior to this paper, several studies highlighted column-stores performing ~5x better than row-stores

Goal: Can a column-store be simulated using a row-store?

Are there benefits inherent to the column-store design?

Methodology of the paper

Dissecting row-stores and column-stores

Can a column-store be simulated using a row-store?

identify the **key design differences**

modify a row-store to behave like a column-store

Are there benefits inherent to the column-store design?

identify the **key optimizations** in a column-store

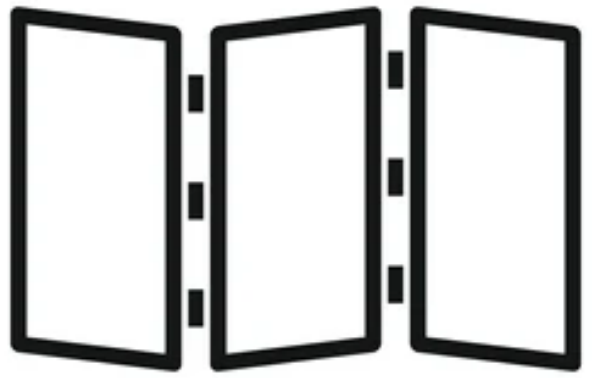
relax the optimizations one at a time

Simulating **column-store in a row-store**

Specialized modifications

Simulating **column-store in a row-store**

Specialized modifications

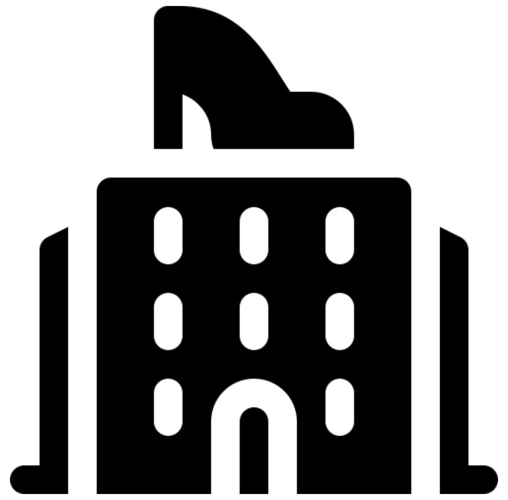


Vertical partitioning

physically partition the data **per column**

Index-only plans

use **only indexes in query plans** that contain only relevant **columns**

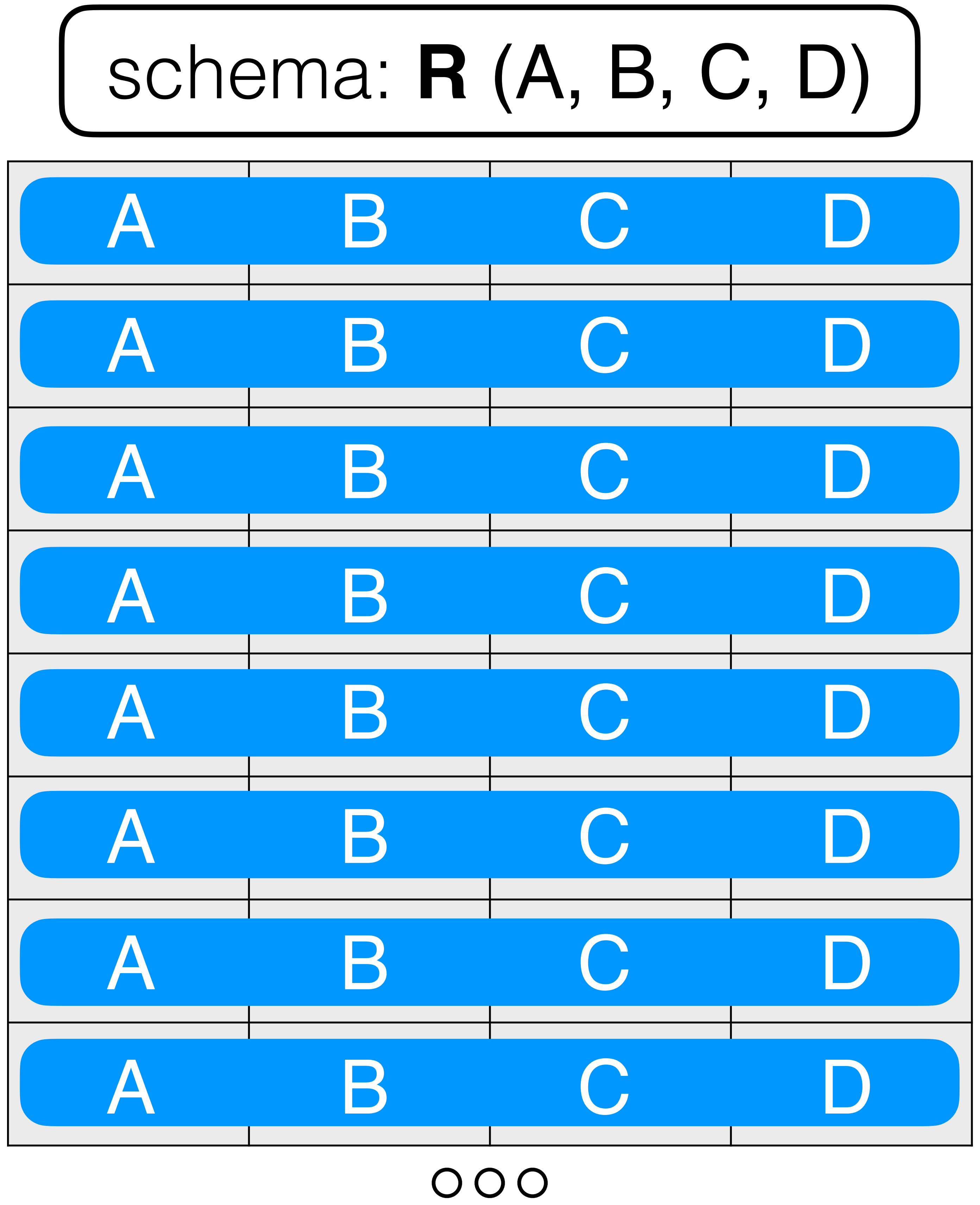


Materialized views

temporary tables that contain **exactly the answer to a query**

Vertical partitioning

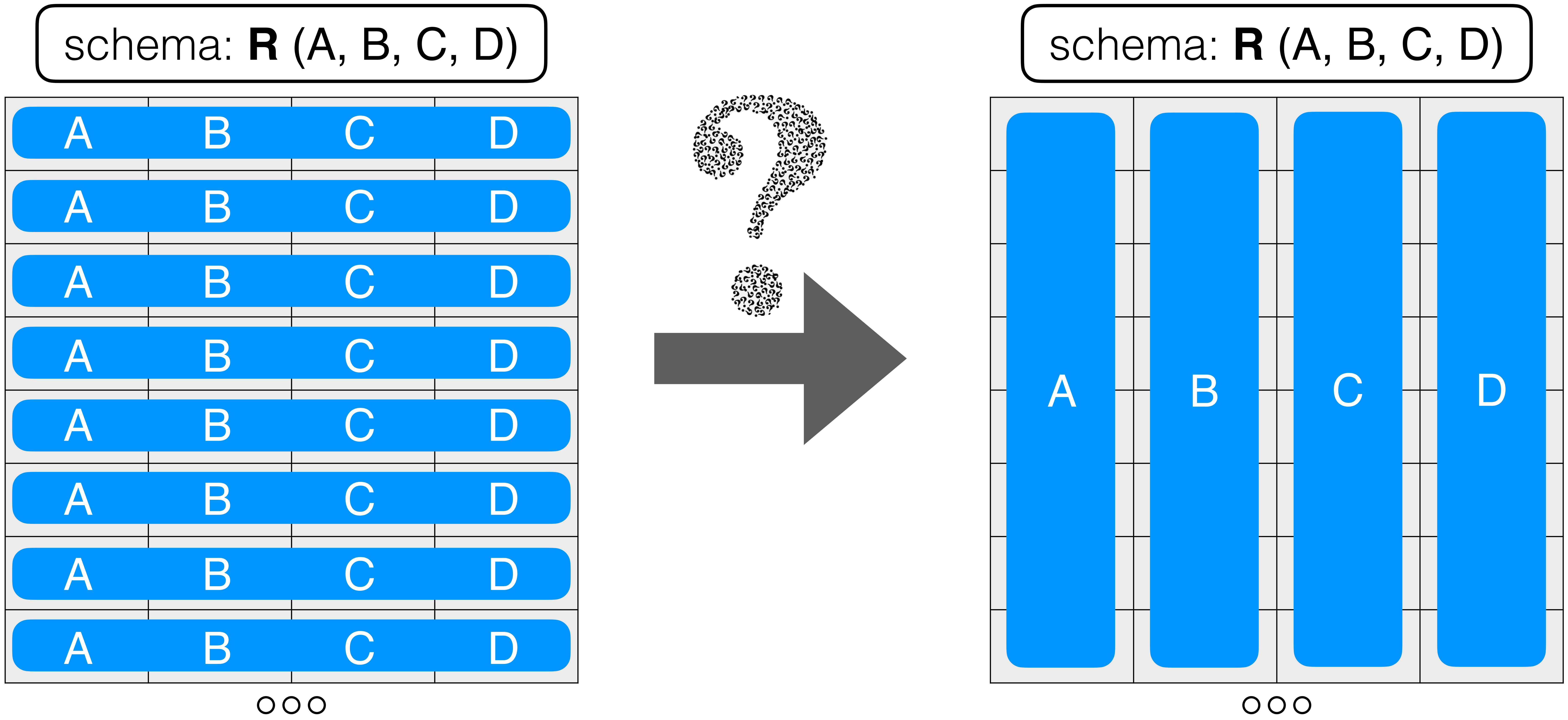
Physically partition the data per column



Brandeis UNIVERSITY

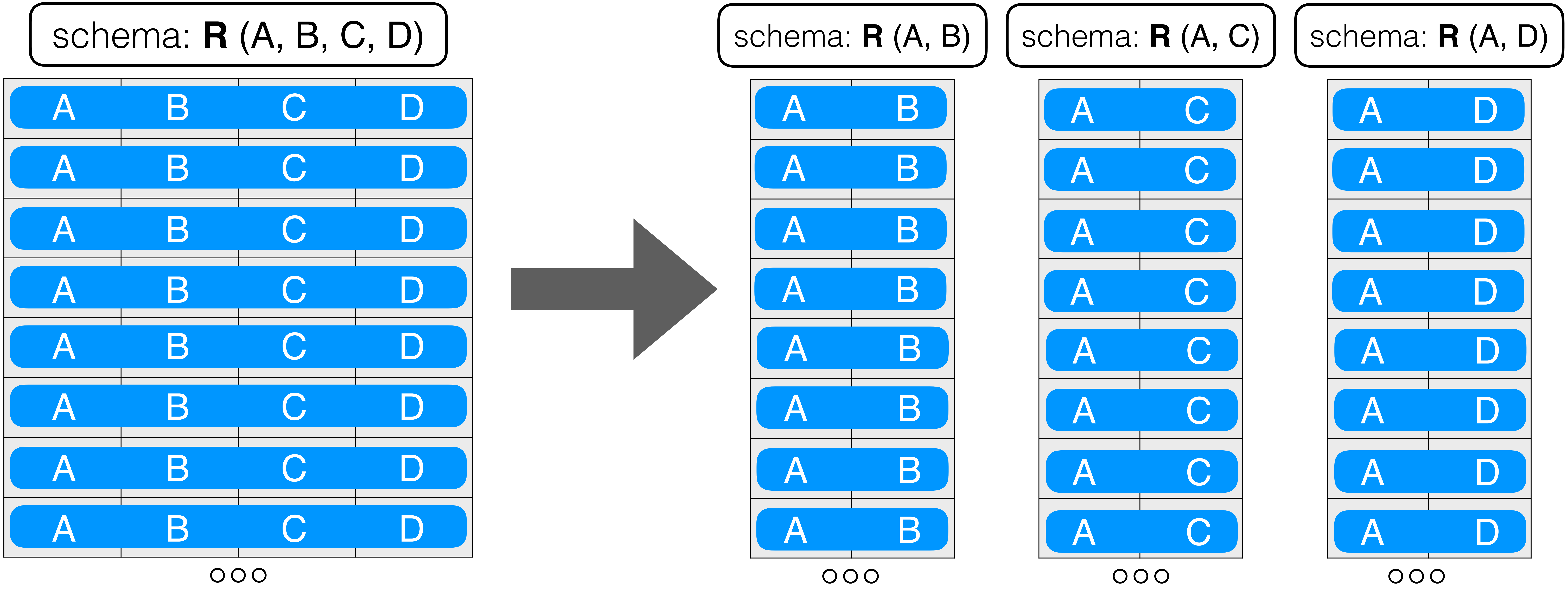
Vertical partitioning

Physically partition the data per column



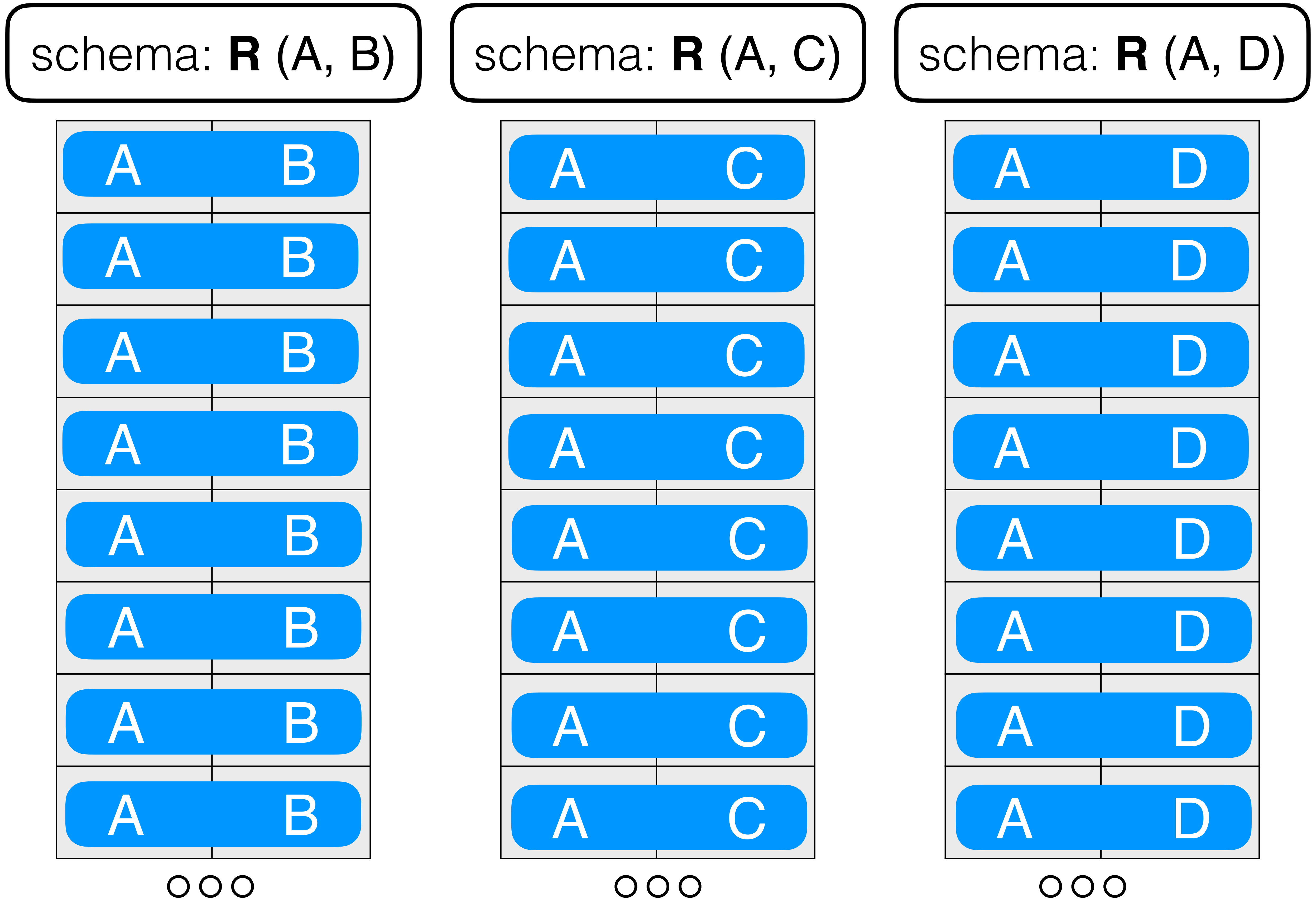
Vertical partitioning

Physically partition the data per column



Vertical partitioning

Physically partition the data per column

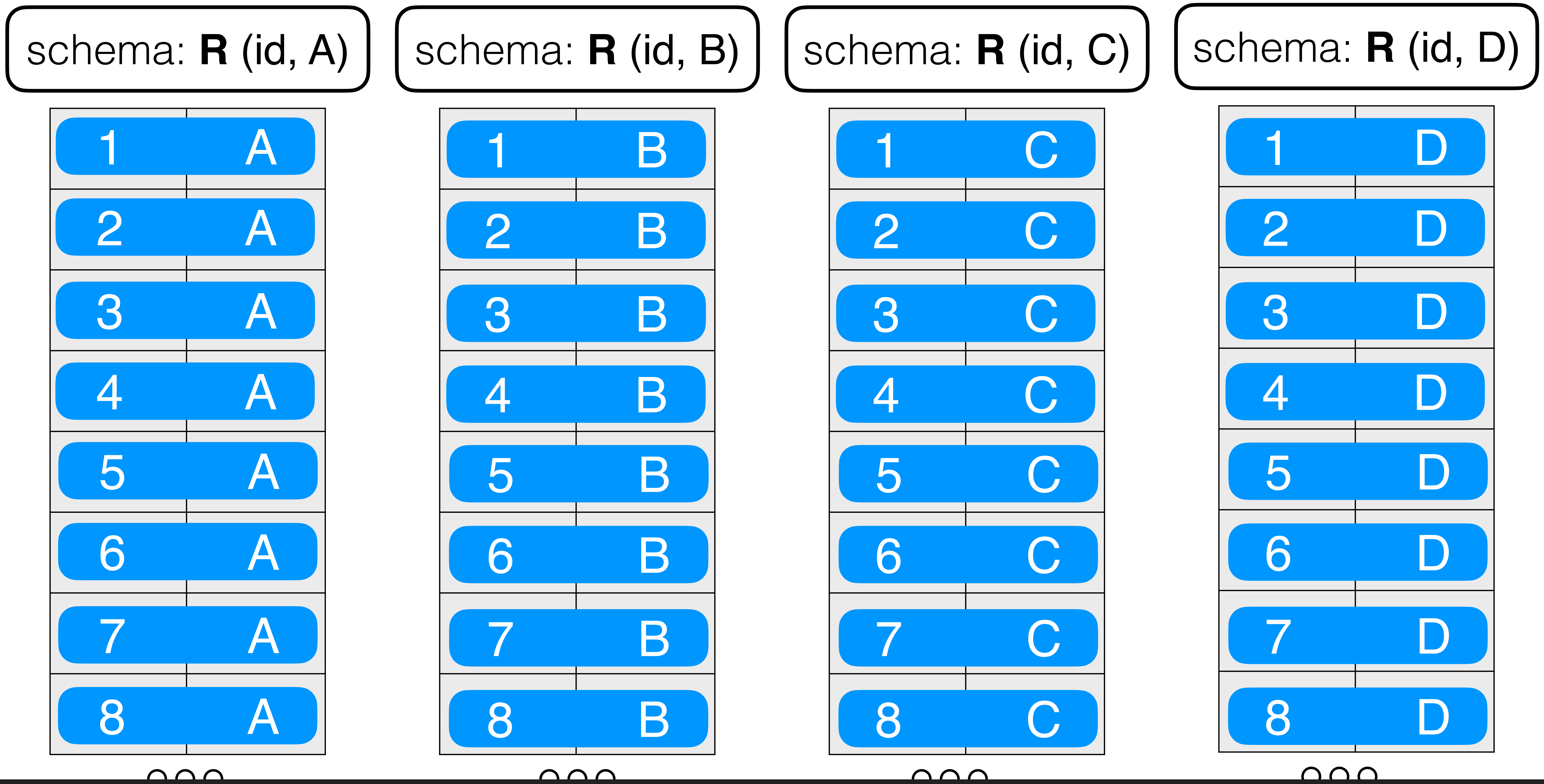


any **problem**?

- **duplicated** attribute
- what if A is **large**

Vertical partitioning

Physically partition the data per column



any **problem**?

- **duplicated** attribute
- **tuple header**

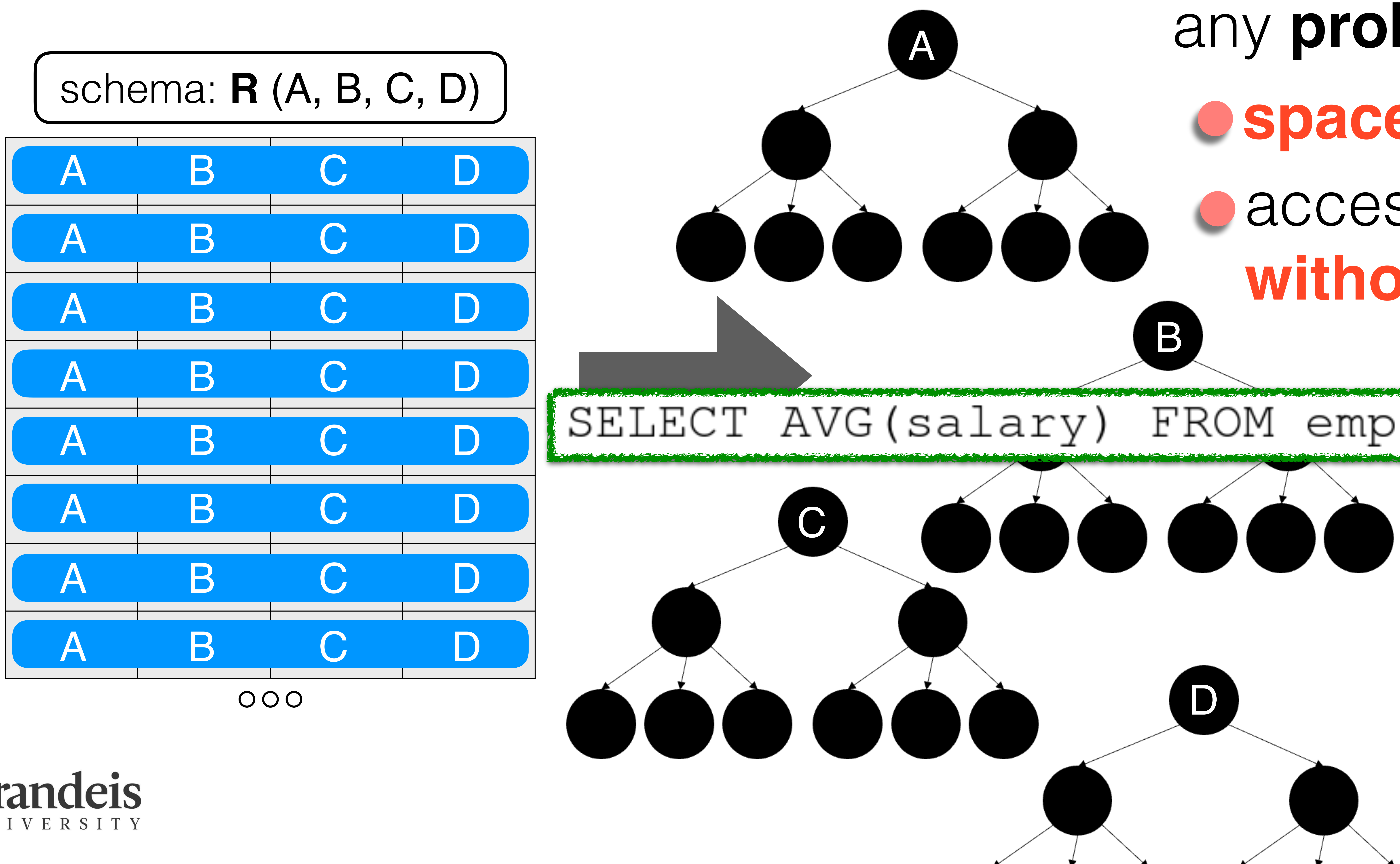
Native column-stores only store **raw values** as an array.

Index-only plans

Only indexes in query plans

any **problem**?

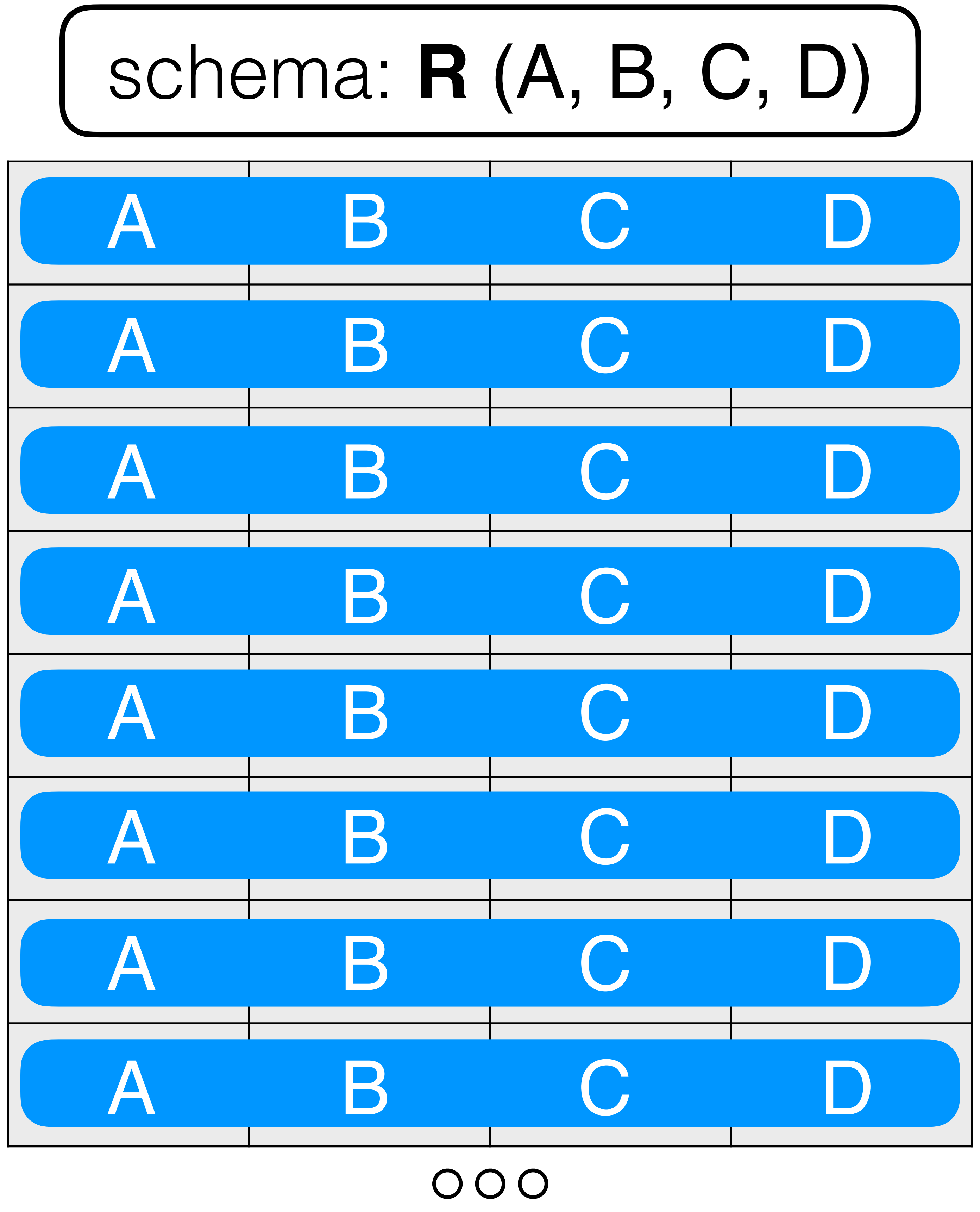
- **space** amplification
- accessing **attributes without predicates**

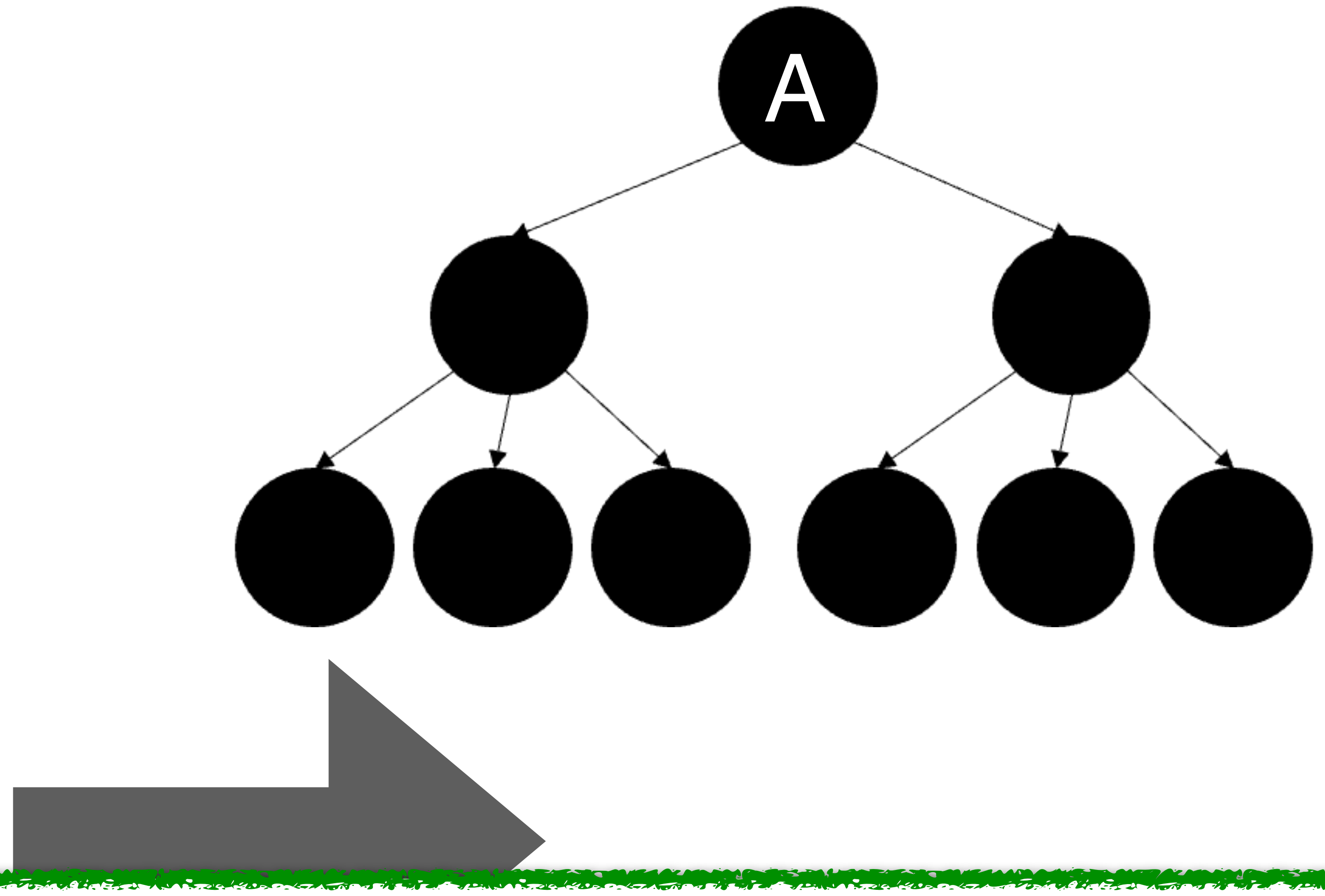


`SELECT AVG(salary) FROM emp WHERE age>40`

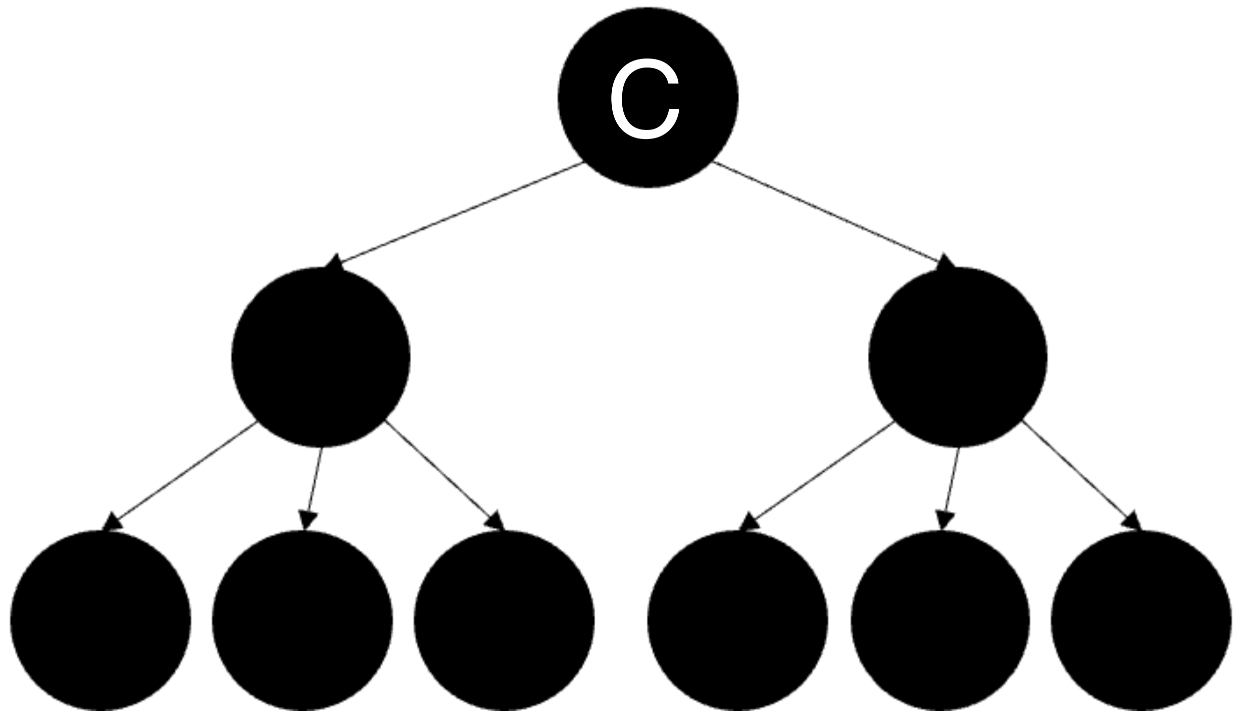
Index-only plans

Only indexes in query plans






```
SELECT AVG(salary) FROM emp WHERE age>40
```



any **problem**?

- **space** amplification
- accessing **attributes without a predicate**

Composite index

- needs **more space**
- **workload** knowledge

Brandeis UNIVERSITY

Materialized views

Tables with exact answers to queries

schema: **R** (A, B, C, D)

A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

ooo

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

schema: **R** (A, B, C)

A	B	C
A	B	C
A	B	C
A	B	C

any **problem**?

- **space** amplification
- **workload** knowledge

Methodology of the paper

Dissecting row-stores and column-stores

Can a column-store be simulated using a row-store?

- identify the **key design differences**
- **modify a row-store** to behave like a column-store

Are there benefits inherent to the column-store design?

- identify the **key optimizations** in a column-store
- **relax the optimizations** one at a time

Methodology of the paper

Dissecting row-stores and column-stores

Can a column-store be simulated using a row-store?

identify the **key design differences**

modify a row-store to behave like a column-store

Are there benefits inherent to the column-store design?

identify the **key optimizations** in a column-store

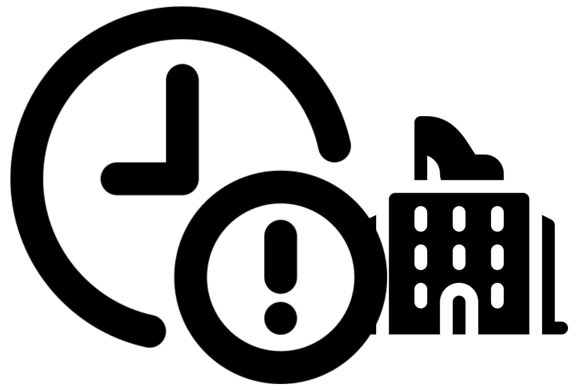
relax the optimizations one at a time

State-of-the-art **column-store designs**

Identifying the optimizations

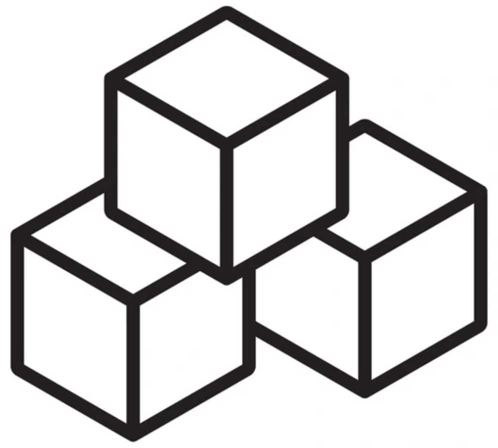
State-of-the-art **column-store designs**

Identifying the optimizations



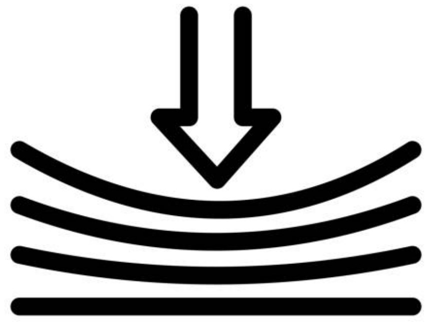
Late materialization

stitch the columns together **as late as possible**



Block iteration

execute columnar operations over a **block of values**



Compression

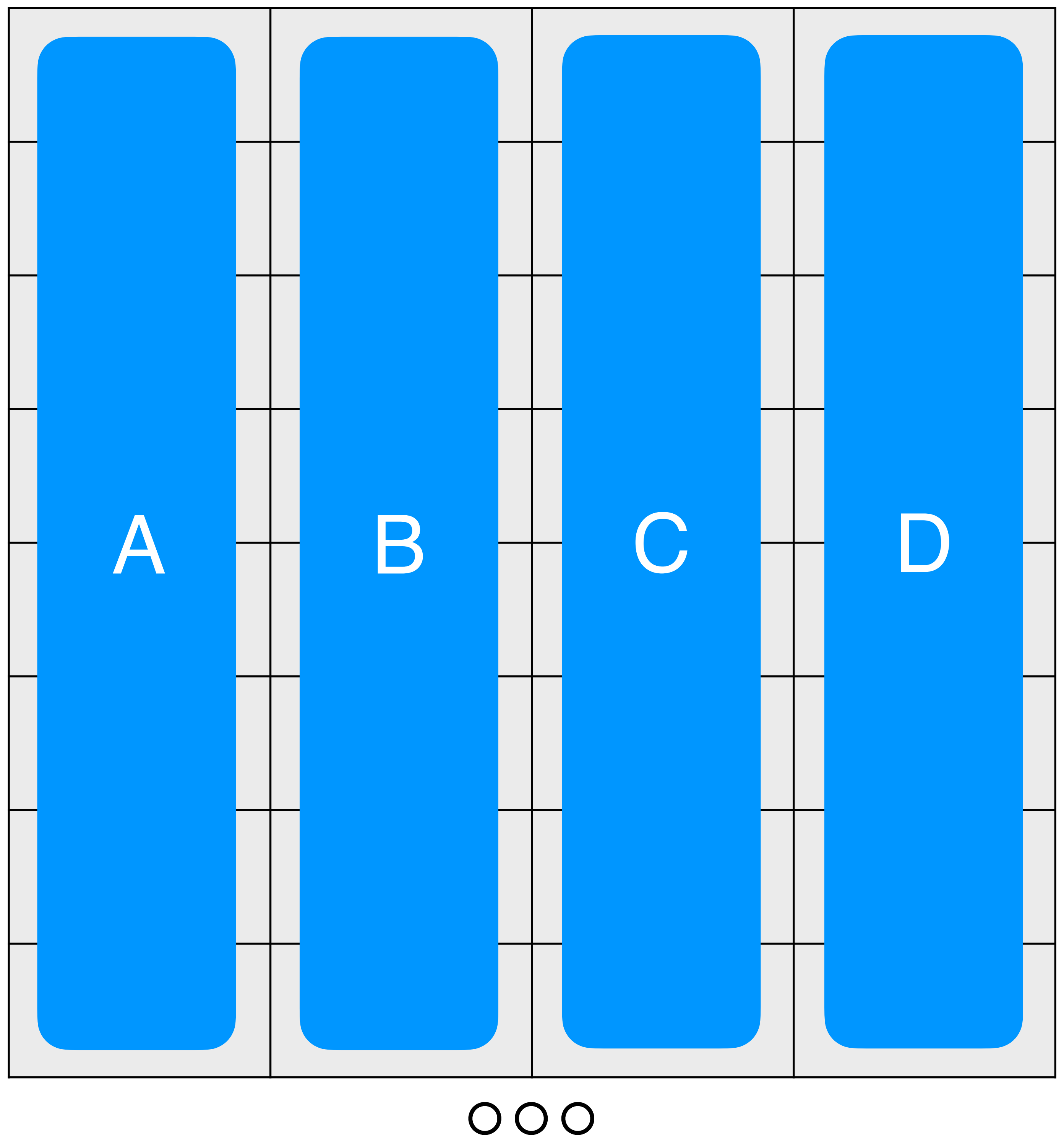
column-specific compression

Invisible join

Querying in a **column-store**

Understanding the schema

schema: **R** (A, B, C, D)



Thought Experiment

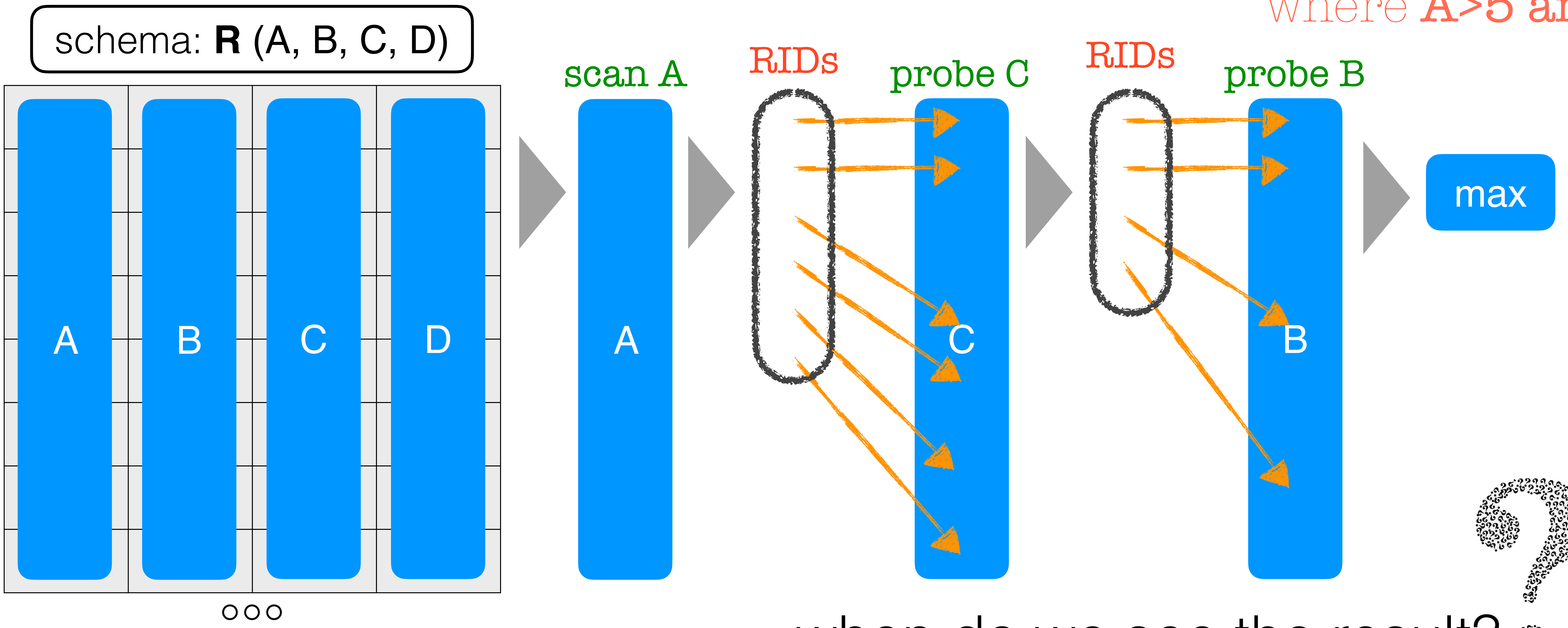
select **max(B)** from **R**
where **A>5 and C<10**

Home work!

Querying in a **column-store**

Understanding the schema

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



when do we see the result?

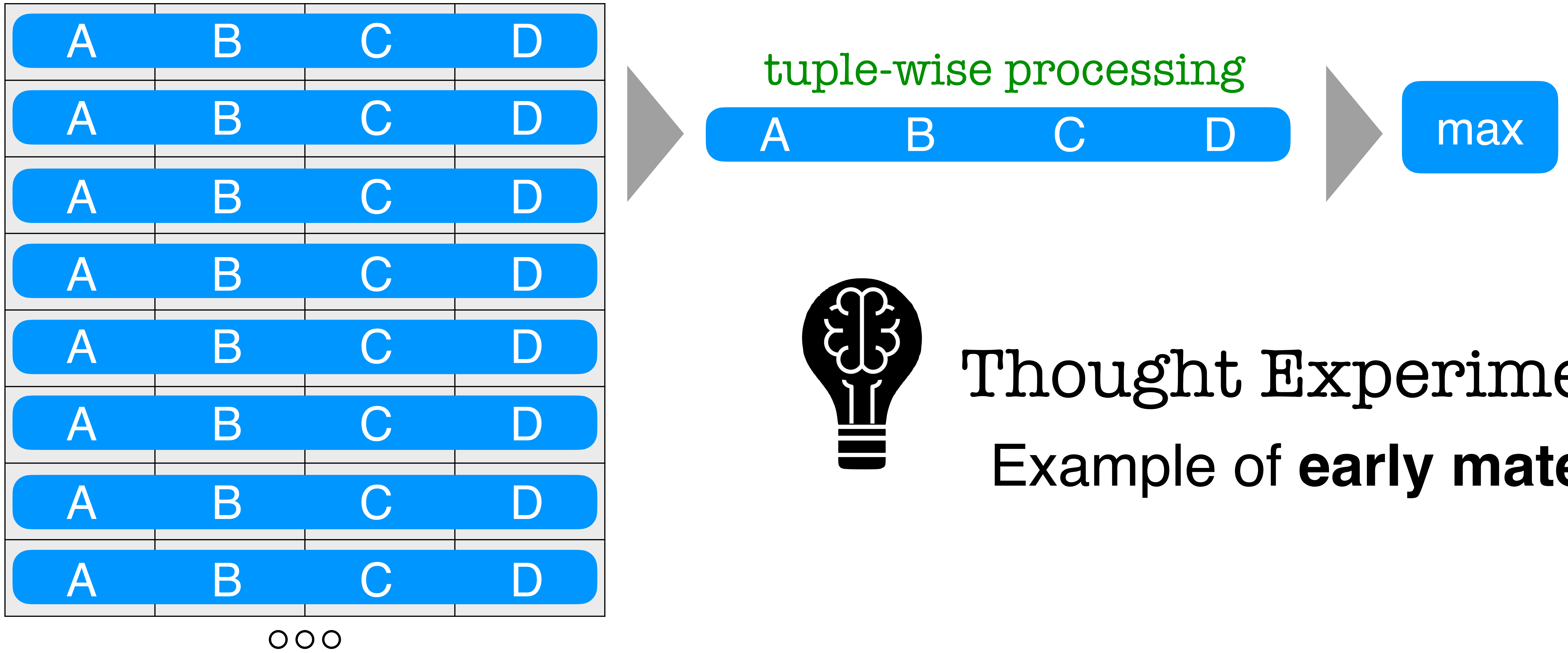
Late materialization

Querying in a **row-store**

Understanding the schema

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

schema: **R** (A, B, C, D)

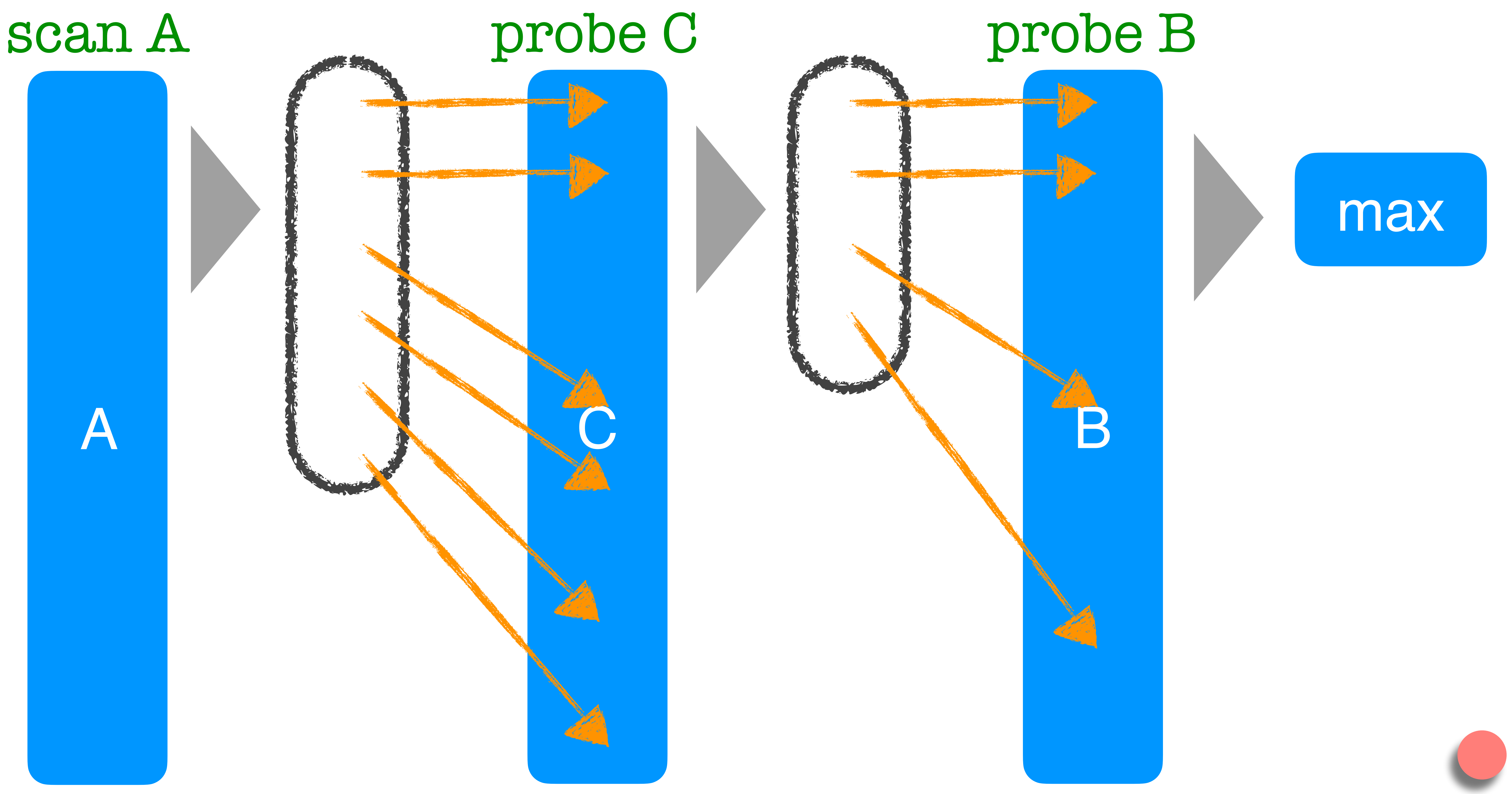


Thought Experiment 3

Example of **early materialization**?

Late materialization

stitch the columns together as late as possible



advantages?

- **cache friendly** (seq. access)
- **minimal reconstruction**
- operate efficiently on **compressed data**

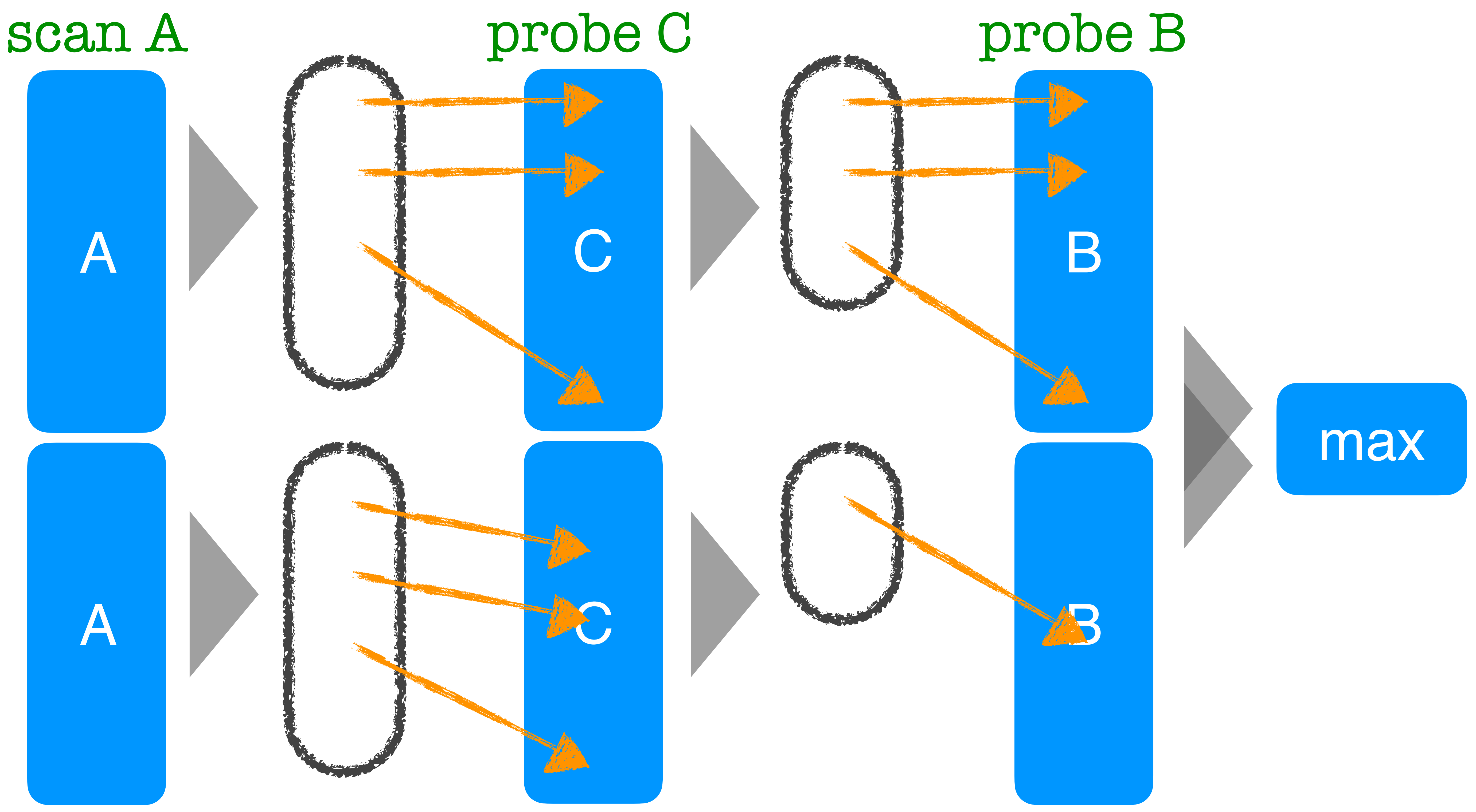
any **problem**?

- **poor resource utilization**
- may require **more I/Os**

Block iteration

execute columnar operations over a **block of values**

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



advantages?

- **good resource utilization**
- **low** query **latency**

Compression

column-specific strategies

row-store

schema: **Billing** (org, quarter, date, state)

Alphabet	Q1	Jan 1, 2024	San Fransicco
Apple	Q1	Jan 11, 2024	Massachusetts
Netflix	Q1	Jan 12, 2024	San Fransicco
Cloudflare	Q1	Jan 12, 2024	Washington
Alphabet	Q2	Jun 17, 2024	San Fransicco
Microsoft	Q2	Jul 17, 2024	Washington
Apple	Q2	Jul 27, 2024	Massachusetts
Alphabet	Q3	Sep 10, 2024	San Fransicco

○○○

Compression

row-store

column-specific strategies

Homogeneous data

schema: **Billing** (org, quarter, date, state)

Alphabet	Q1	Jan 1, 2024	San Fransicco
Apple	Q1	Jan 11, 2024	Massachusetts
Netflix	Q1	Jan 12, 2024	San Fransicco
Cloudflare	Q1	Jan 12, 2024	Washington
Alphabet	Q2	Jun 17, 2024	San Fransicco
Microsoft	Q2	Jul 17, 2024	Washington
Apple	Q2	Jul 27, 2024	Massachusetts
Alphabet	Q3	Sep 10, 2024	San Fransicco

ooo

vs.

column-stores

Alphabet
Apple
Netflix
Cloudflare
Alphabet
Microsoft
Apple
Alphabet

ooo

Q1
Q1
Q1
Q1
Q2
Q2
Q2
Q3

ooo

Jan 1, 2024
Jan 11, 2024
Jan 12, 2024
Jan 12, 2024
Jun 17, 2024
Jul 17, 2024
Jul 27, 2024
Sep 10, 2024

ooo

San Fransicco
Massachusetts
San Fransicco
Washington
San Fransicco
Washington
Massachusetts
San Fransicco

ooo

which one is **easily compressible**?

Compression

column-specific strategies

column-stores

Alphabet	Q1	Jan 1, 2024	San Fransicco
Apple	Q1	Jan 11, 2024	Massachusetts
Netflix	Q1	Jan 12, 2024	San Fransicco
Cloudflare	Q1	Jan 12, 2024	Washington
Alphabet	Q2	Jun 17, 2024	San Fransicco
Microsoft	Q2	Jul 17, 2024	Washington
Apple	Q2	Jul 27, 2024	Massachusetts
Alphabet	Q3	Sep 10, 2024	San Fransicco

Thought Experiment 4

How do column-stores **compress data** efficiently?

Compression

column-specific strategies

column-stores

Alphabet	Q1	Jan 1, 2024	San Fransicco
Apple	Q1	Jan 11, 2024	Massachusetts
Netflix	Q1	Jan 12, 2024	San Fransicco
Cloudflare	Q1	Jan 12, 2024	Washington
Alphabet	Q2	Jun 17, 2024	San Fransicco
Microsoft	Q2	Jul 17, 2024	Washington
Apple	Q2	Jul 27, 2024	Massachusetts
Alphabet	Q3	Sep 10, 2024	San Fransicco
ooo	ooo	ooo	ooo

Thought Experiment 4

How do column-stores **compress data** efficiently?

100M entries; 100K+ unique organizations

Dictionary compression

Replace **variable-length strings** with **fixed-sized integers**

Compression

column-specific strategies

column-stores

Alphabet	Q1	Jan 1, 2024	San Fransicco
Apple	Q1	Jan 11, 2024	Massachusetts
Netflix	Q1	Jan 12, 2024	San Fransicco
Cloudflare	Q1	Jan 12, 2024	Washington
Alphabet	Q2	Jun 17, 2024	San Fransicco
Microsoft	Q2	Jul 17, 2024	Washington
Apple	Q2	Jul 27, 2024	Massachusetts
Alphabet	Q3	Sep 10, 2024	San Fransicco
ooo	ooo	ooo	ooo

Thought Experiment 4

How do column-stores **compress data** efficiently?

100M entries; 100K+ unique organizations

Dictionary compression

Replace **variable-length strings** with **fixed-sized integers**

Use a **constant number of bits** if the **domain is fixed**

Compression

column-specific strategies

column-stores

Alphabet	Q1	Jan 1, 2024	San Fransicco
Apple	Q1	Jan 11, 2024	Massachusetts
Netflix	Q1	Jan 12, 2024	San Fransicco
Cloudflare	Q1	Jan 12, 2024	Washington
Alphabet	Q2	Jun 17, 2024	San Fransicco
Microsoft	Q2	Jul 17, 2024	Washington
Apple	Q2	Jul 27, 2024	Massachusetts
Alphabet	Q3	Sep 10, 2024	San Fransicco
ooo	ooo	ooo	ooo

Thought Experiment 4

How do column-stores **compress data** efficiently?

100M entries; 50 states

Delta compression

Store **only** the **deltas** (**differences**)

Compression

column-specific strategies

column-stores

Alphabet	Q1	Jan 1, 2024	San Fransicco
Apple	Q1	Jan 11, 2024	Massachusetts
Netflix	Q1	Jan 12, 2024	San Fransicco
Cloudflare	Q1	Jan 12, 2024	Washington
Alphabet	Q2	Jun 17, 2024	San Fransicco
Microsoft	Q2	Jul 17, 2024	Washington
Apple	Q2	Jul 27, 2024	Massachusetts
Alphabet	Q3	Sep 10, 2024	San Fransicco
○○○	○○○	○○○	○○○

Thought Experiment 4

How do column-stores **compress data** efficiently?

200 Q1's, 300 Q2's, 1000 Q3's, ...

Compression

column-specific strategies

Q1
Q1
Q1
Q1
Q2
Q2
Q2
Q3
ooo

Q1	1	200
Q2	201	300
Q3	301	1300
Q4	1301	2500

Thought Experiment 4

How do column-stores **compress data** efficiently?

200 Q1's, 300 Q2's, 1000 Q3's, 1200 Q4's, …

Run-length encoding

Store **only** the **start index** & **frequency**

Can operate on **compressed data**

Needs to be **sorted**

Invisible join

Star-schema specific optimization

Benchmarking

The set up!

When comparing database systems we need a **common "language"**

standardization is key for future **comparison**

Benchmarks from the **Transaction Performance Council**

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

Also, a benchmark for **data warehousing**

Star Schema Benchmark

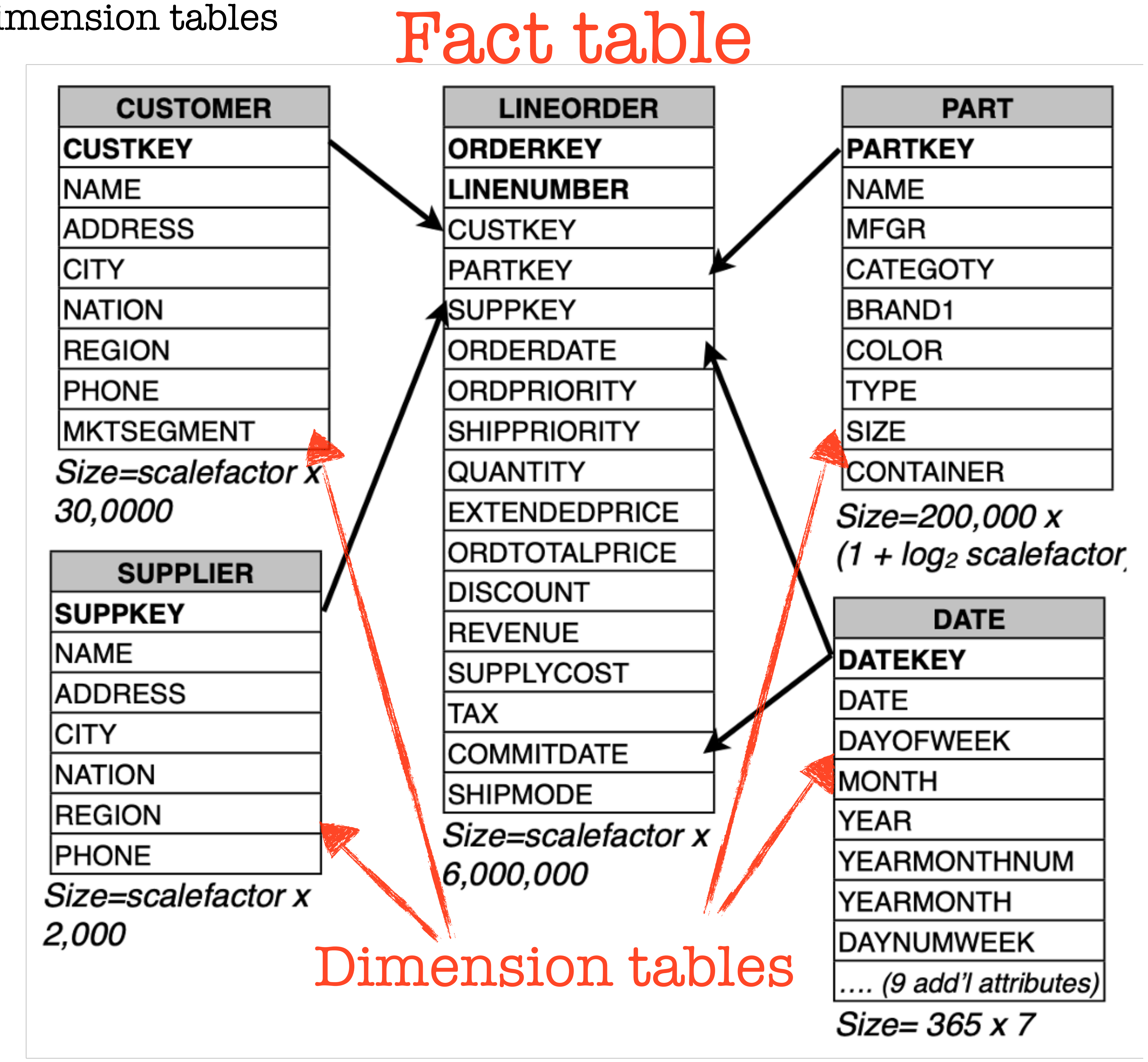
Star Schema Benchmark

Fact table and Dimension tables

Comes with **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 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;
```



Invisible join

Star-schema specific optimization

Motivation: 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

```sql
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;
```

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

```sql
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;
```

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**

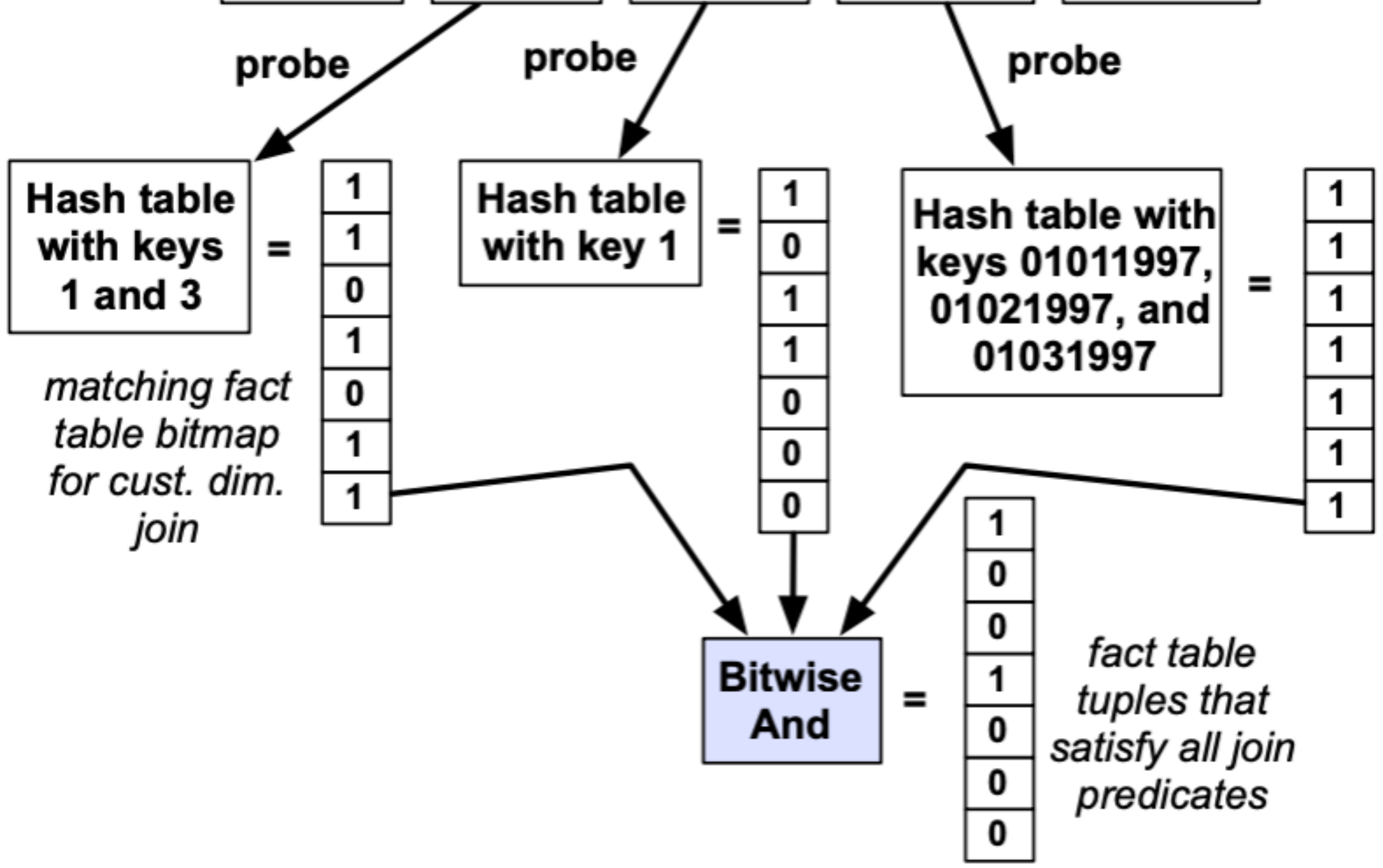
dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

→ **Hash table with keys 01011997, 01021997, and 01031997**

3. compute bitvector with bits set for qualifying positions (tuples)

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

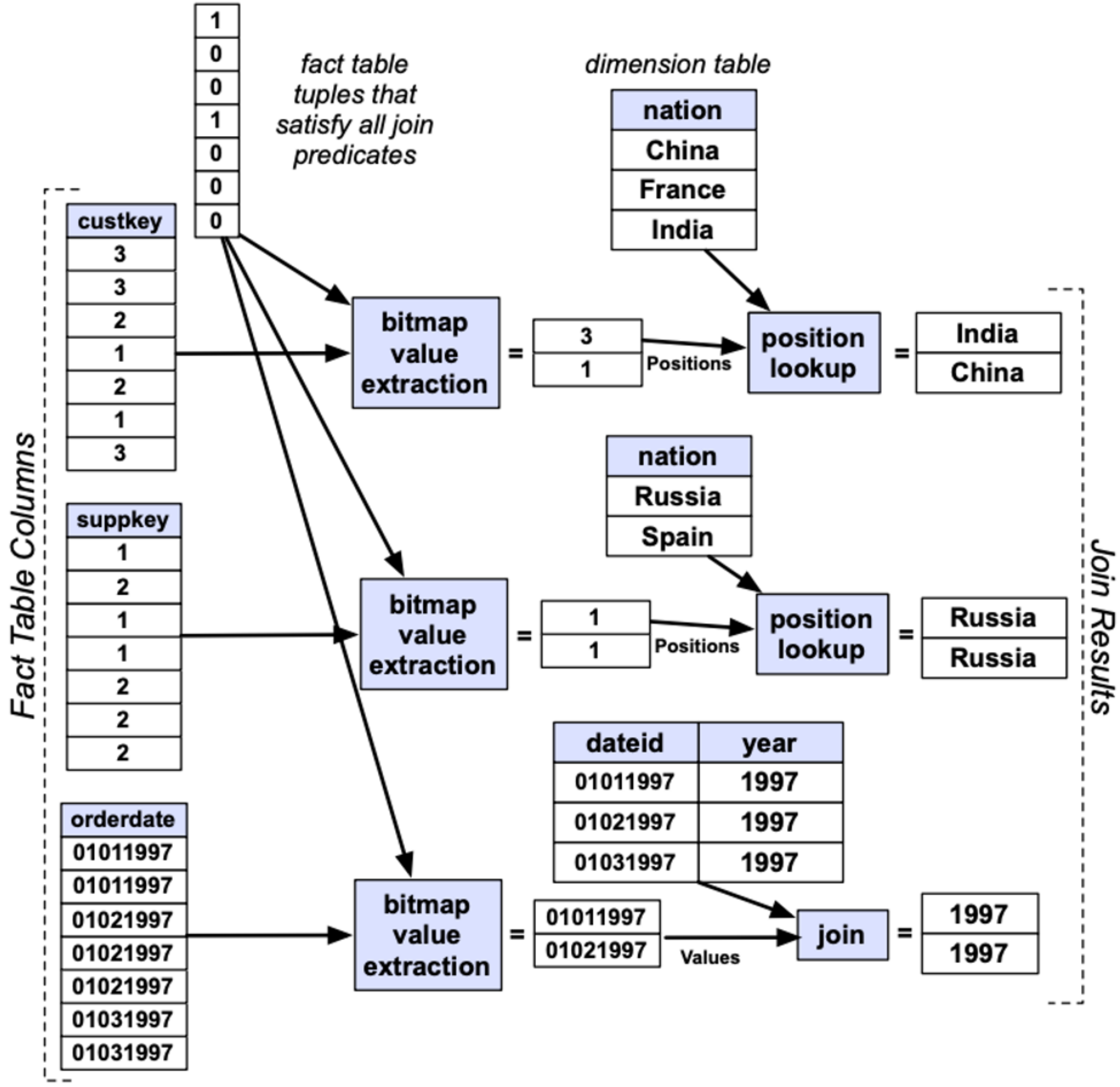


4. intersect bitvectors (positions) via bitwise AND

```sql
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;
```

5. for each resulting position reconstruct the resulting tuple

- works only for **star schemas**
- **not** a general join algorithm



Brandeis UNIVERSITY

Experiments

Comparing the results

Experiments

Comparing the results

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)

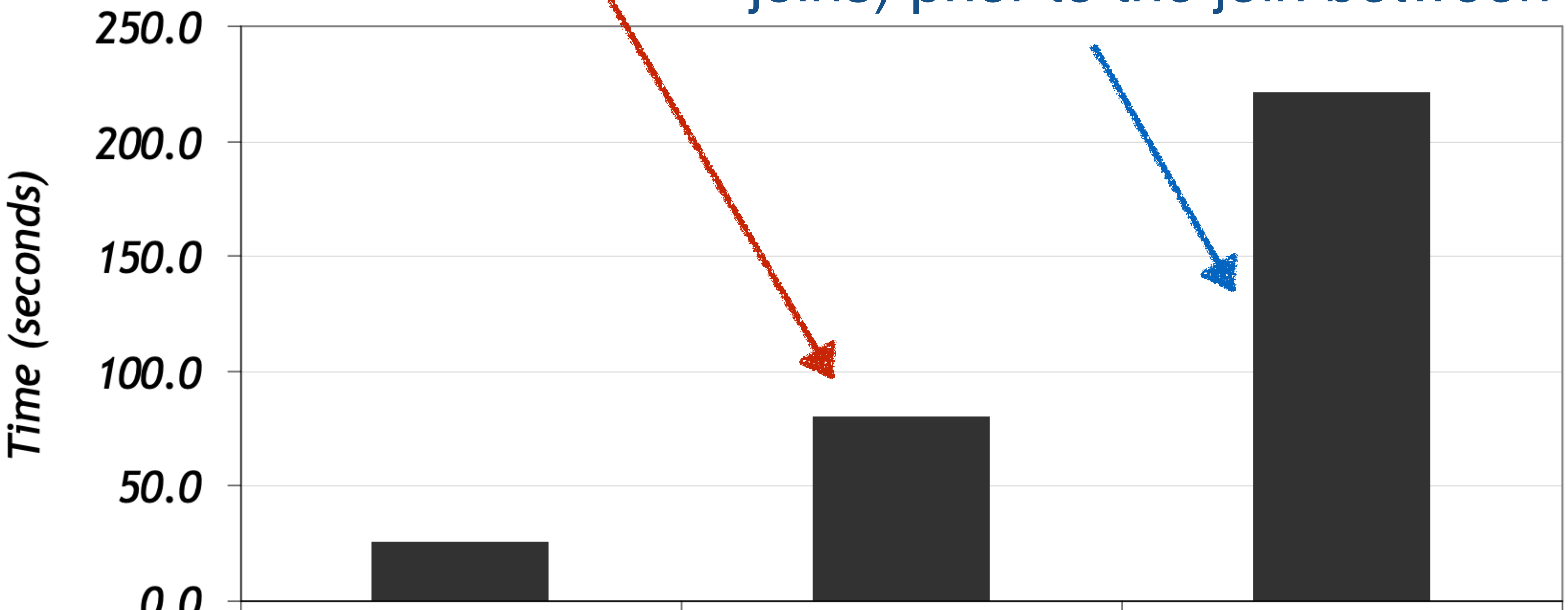
Focus on SSB averages (the paper has more detailed graphs)

Experiments with **row-stores**

Comparing the results

tuple overheads
(additional record IDS)

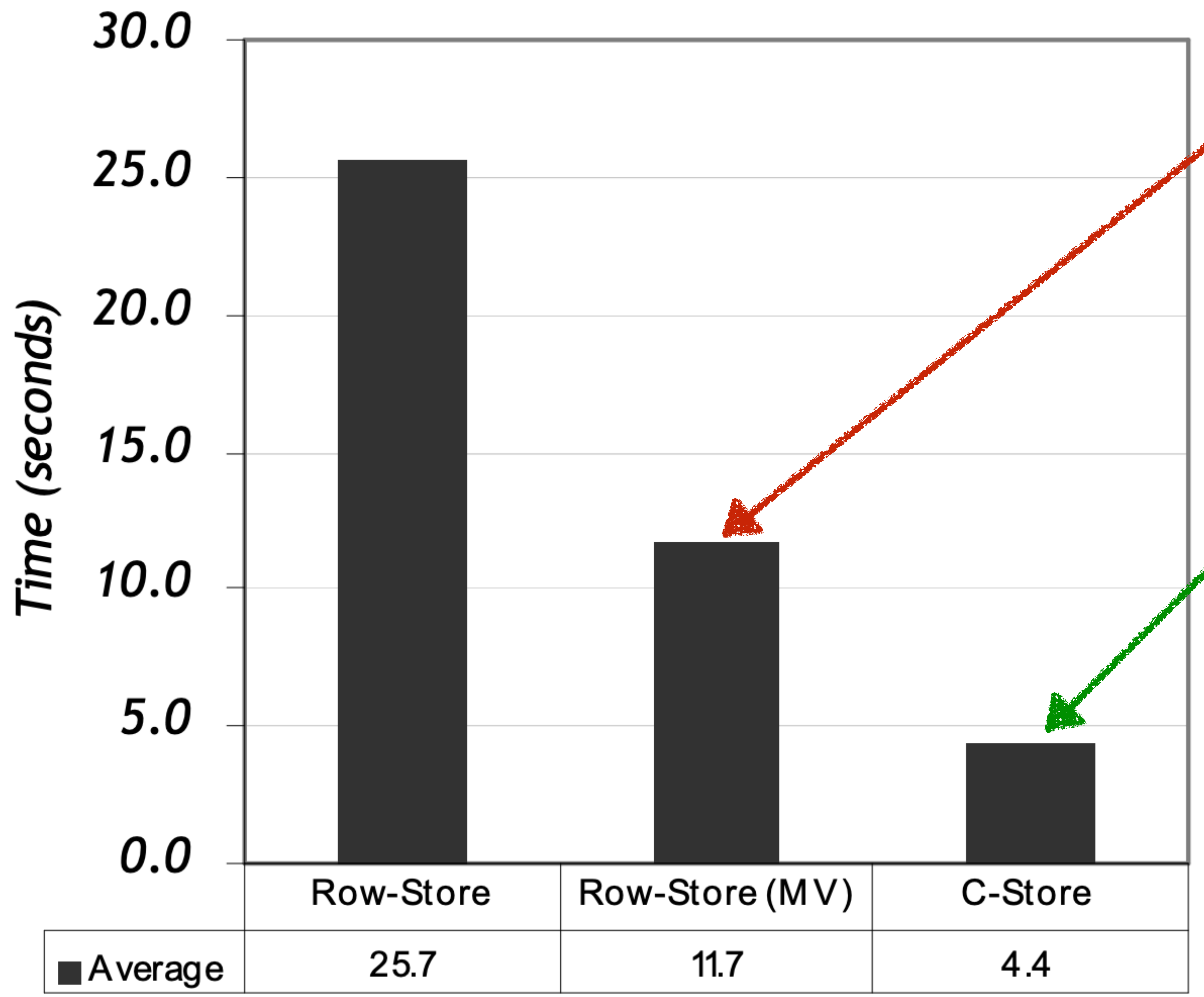
tuple reconstruction (via expensive joins) prior to the join between tables



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

Row-stores vs. **column-stores**

Comparing the results



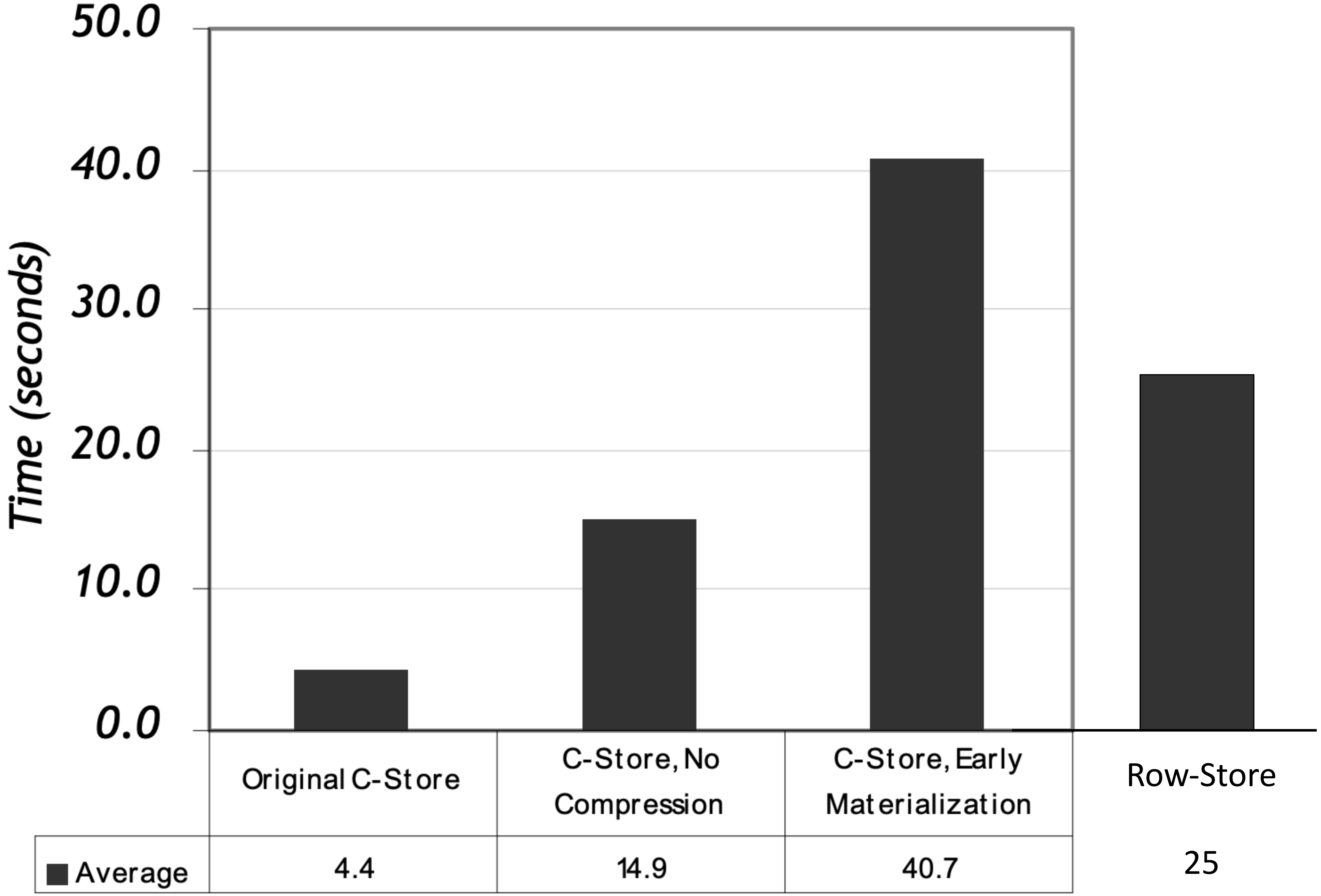
MV have the result readily available

A native column-store is faster

Brandeis
UNIVERSITY

Row-stores vs. **column-stores**

Comparing the results



To make the most of a column-store:

1. efficient **compression**
2. **column-specific execution** (late materialization)

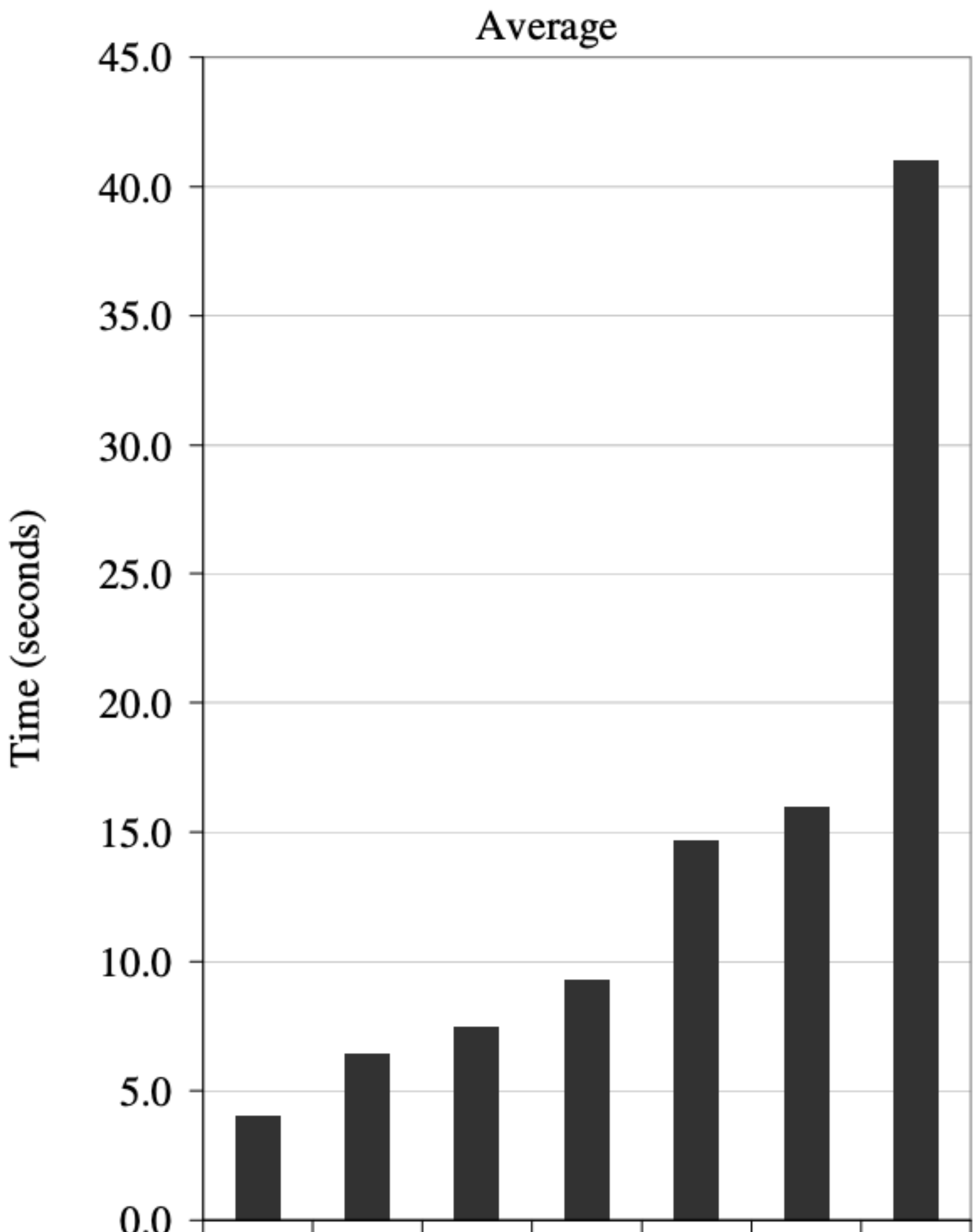
C-store appears to do even better than fully materialized joins

Block processing buys you 5 to 50%

Invisible join buys you 50-75%

Compression buys you 2X

Late materialization gets you almost 3X



	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
Average	4.0	6.4	7.5	9.3	14.7	16.0	41.0

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

Summary

The key takeaways

Row-stores & Column-stores are fundamentally different!

- Compression
- Late materialization
- Block iteration
- Column-store-specific join optimizations

Next time in COSI 167A

Intro. + Administrivia

Introduction to **LSM-trees**

[P] "LSM-based Storage Techniques: A Survey", *VLDB Journal*, 2019

[B] "Dissecting, Designing, and Optimizing LSM-based Data Stores", *SIGMOD*, 2022

COSI 167A
Advanced Data Systems

Class 4

Row-stores vs. Column-stores

Prof. Subhadeep Sarkar

https://ssd-brandeis.github.io/COSI-167A/

Brandeis
UNIVERSITY