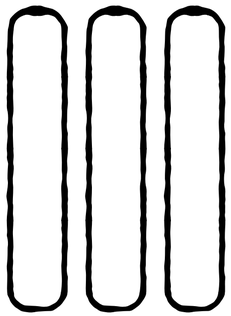
Column-Stores



CSC443/CSC2234 Database System Technology

Niv Dayan

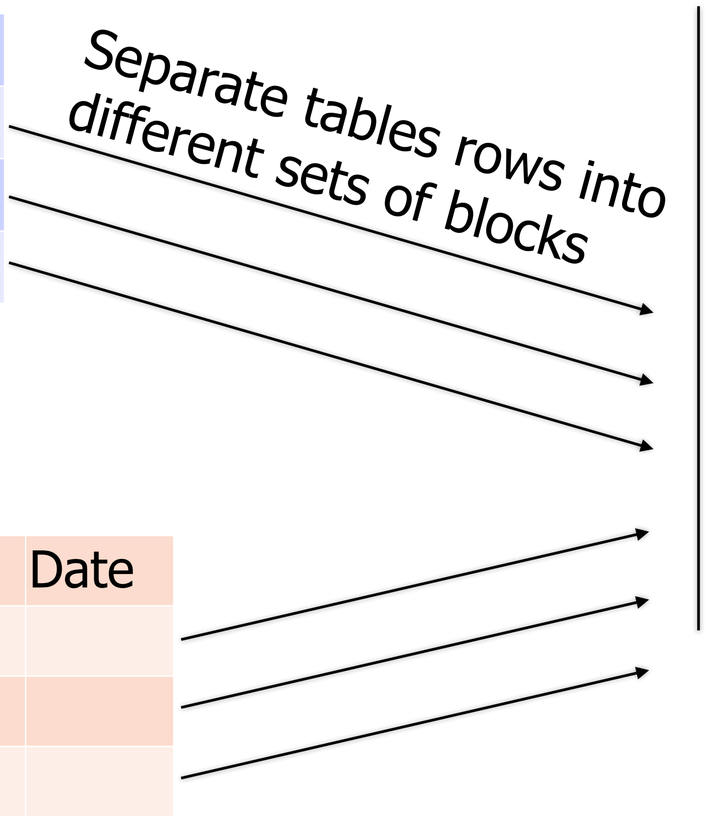
Efficient Scans

Customers

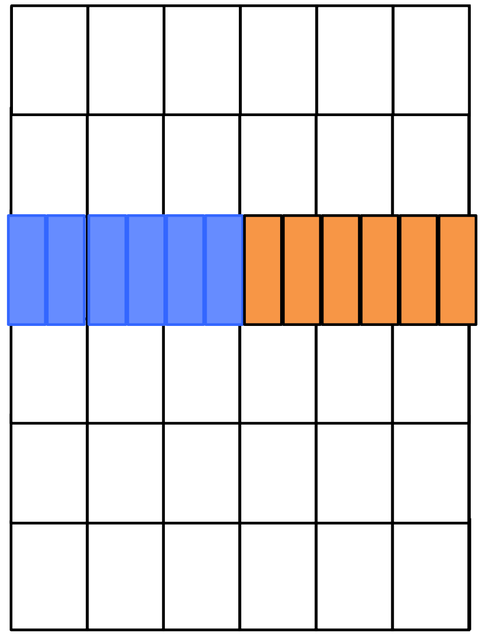
ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date



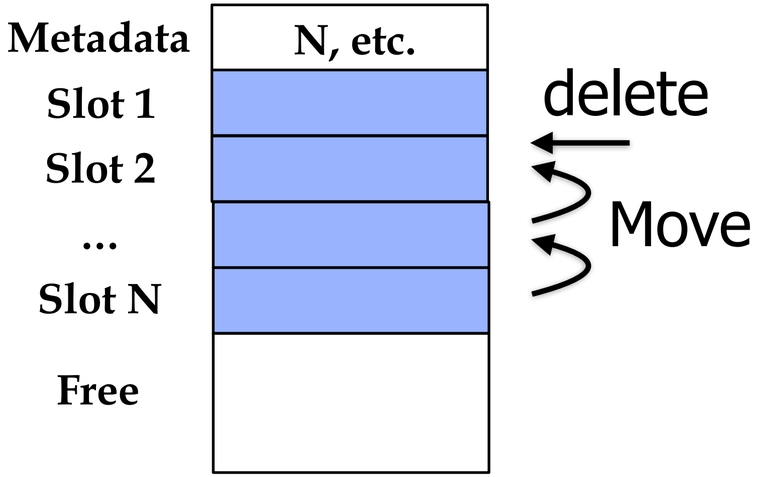
Address Space



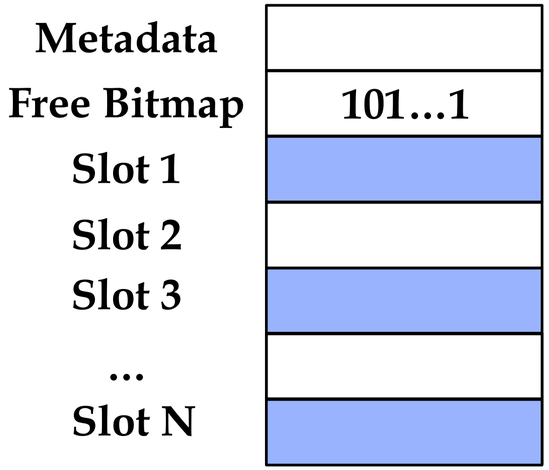
Scan cost: O(N/B) I/Os

Internal Page Organization for Fixed-Sized Rows

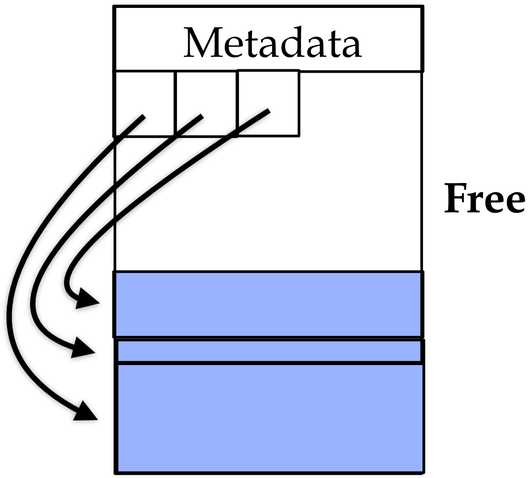
Entries compactly packed at front of page



Can also use a bitmap to mark occupied slots



Internal Page Organization for Variable-Sized Rows

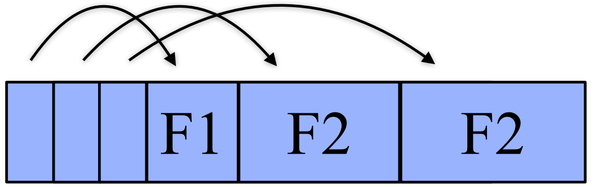


Variable-Sized Row Organization

Delimiters

Smaller
No random access

Pointers



More space
Random access (faster)

Variable-Sized Row Organization

Fetching a particular row entails significant traversal & pointer chasing overheads

So far we have assumed rows are stored contiguously

ID	Name	email	Addr	Salary
ID1	name1	email1	addr1	salary1
ID2	name2	email2	addr2	salary2
ID3	name3	email3	addr3	salary3

So far we have assumed rows are stored contiguously

ID	Name	email	Addr	Salary

So far we have assumed rows are stored contiguously

ID	Name	email	Addr	Salary

Great for queries that examine most columns

So far we have assumed rows are stored contiguously

ID	Name	email	Addr	Salary

Great for queries that examine most columns

e.g., Select * from Customers

e.g., Select **Id, Name, Email, Addr** from Customers where salary > 10000

So far we have assumed rows are stored contiguously

ID	Name	email	Addr	Salary

But how about queries that examine few columns?

e.g., Select **avg(salary)** from Customers

e.g., Select **sum(email)** from Customers where email like '%@gmail.com';

Problem?

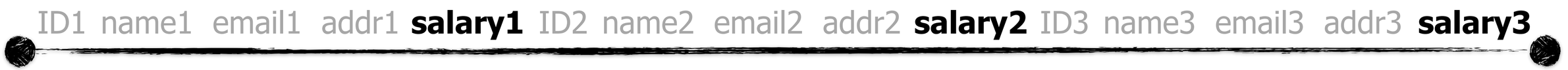
So far we have assumed rows are stored contiguously

ID	Name	email	Addr	Salary

Select avg(**salary**) from Customers

Problem? We only need a little of each row

So far we have assumed rows are stored contiguously



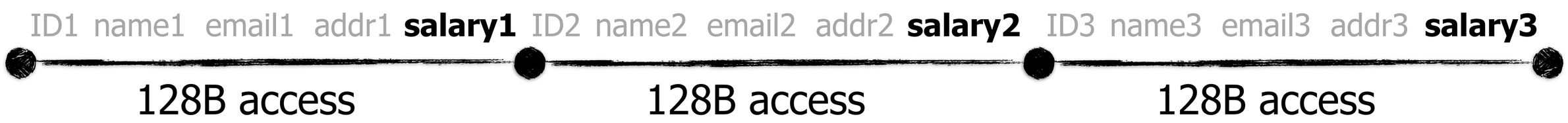
4 KB access

Select avg(**salary**) from Customers

Problem? We only need a little of each row

But storage access granularity is coarse

So far we have assumed rows are stored contiguously



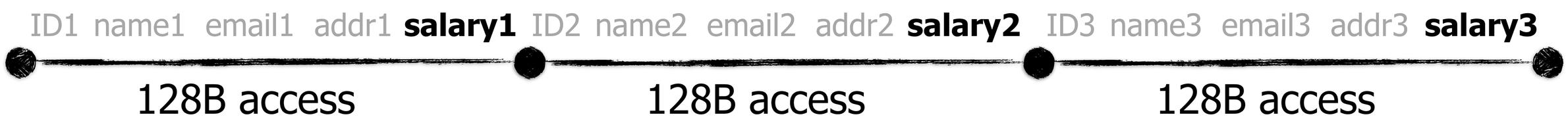
Select avg(**salary**) from Customers

Problem? We only need a little of each row

But storage access granularity is coarse

Memory can also only be accessed in cache lines

So far we have assumed rows are stored contiguously



Select avg(**salary**) from Customers

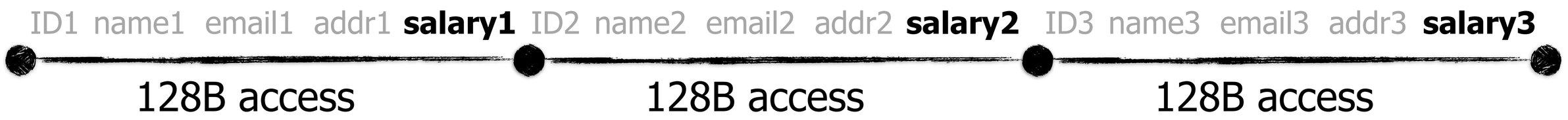
Problem? We only need a little of each row

But storage access granularity is coarse

Memory can also only be accessed in cache lines

Reading more than we are interested in wastes bandwidth

So far we have assumed rows are stored contiguously



Select avg(**salary**) from Customers

Problem? We only need a little of each row

But storage access granularity is coarse

Memory can also only be accessed in cache lines

Reading more than we are interested in wastes bandwidth

Solution?

What if we instead store the data one column at a time?

ID	Name	email	Addr	Salary
ID1	name1	email1	addr1	salary1
ID2	name2	email2	addr2	salary2
ID3	name3	email3	addr3	salary3

What if we instead store the data one column at a time?

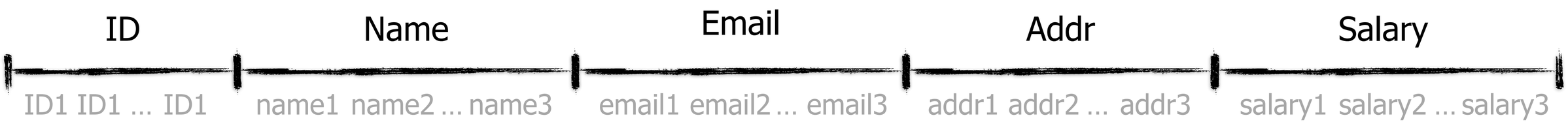
ID	Name	email	Addr	Salary

ID1 ID1 … ID1 name1 name2 … name3 email1 email2 … email3 addr1 addr2 … addr3 salary1 salary2 … salary3

Address Space

What if we instead store the data one column at a time?

ID	Name	email	Addr	Salary



What if we instead store the data one column at a time?

ID	Name	email	Addr	Salary

What if we instead store the data one column at a time?

ID	Name	email	Addr	Salary

Select avg(**salary**) from Customers

We can now skip irrelevant data!

What if we instead store the data one column at a time?

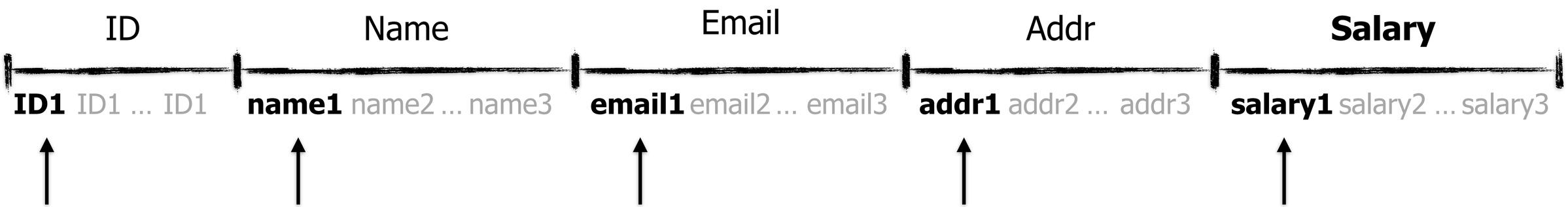
ID	Name	email	Addr	Salary

Select avg(salary) from Customers

Any concerns?

What if we instead store the data one column at a time?

ID	Name	email	Addr	Salary

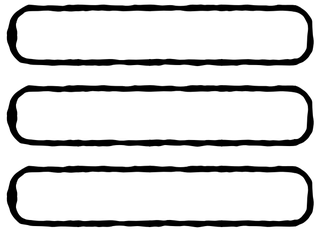


Select * from Customers where ID = 1

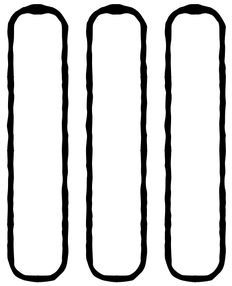
Trade-off: random I/Os for selective queries across many columns

The two relational database families

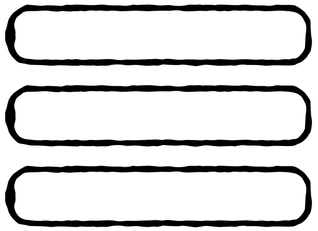
Row-Stores



Column-Stores

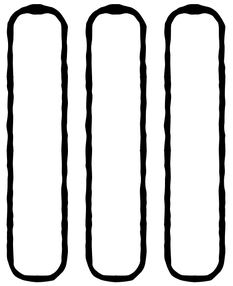


Row-Stores



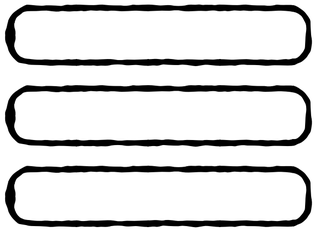
Selective queries/updates

Column-Stores



Large statistical calculations & batch updates

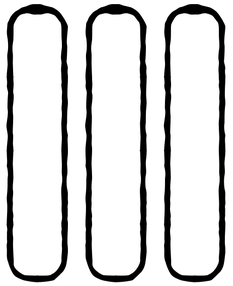
Row-Stores



Selective queries/updates

Online Transaction Processing (OLTP)

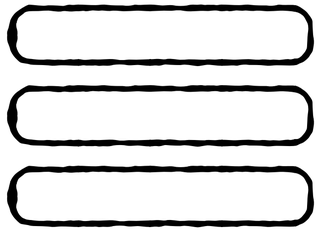
Column-Stores



Large statistical calculations & batch updates

Online Analytical Processing (OLAP)

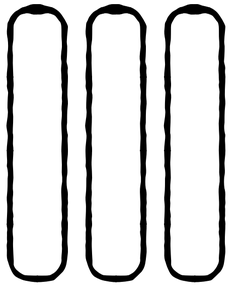
Row-Stores



OLTP examples

Banking, retail, social media

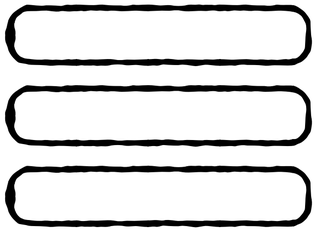
Column-Stores



OLAP examples

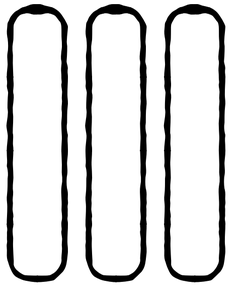
Analytics, machine learning

Row-Stores



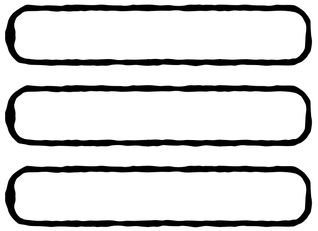
Postgres, MariaDB, etc.

Column-Stores



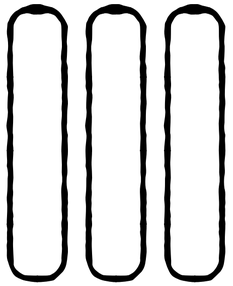
MonetDB, Vectorwise, C-Store, Vertica
(Today all major DBs offer this, e.g.,)

Row-Stores



Postgres, MariaDB, etc.

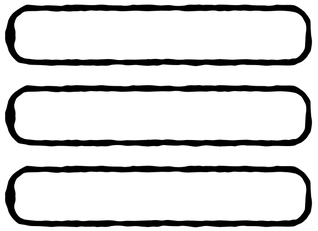
Column-Stores



MonetDB, Vectorwise, C-Store, Vertica
(Today all major DBs offer this, e.g.,)

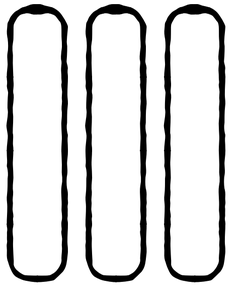
Oracle, IBM, Microsoft were here

Row-Stores



Postgres, MariaDB, etc.

Column-Stores



MonetDB, Vectorwise, C-Store, Vertica
(Today all major DBs offer this, e.g.,)

Oracle, IBM, Microsoft were here

But now they offer both

First row-store Developed and productized

First open-source Column-store developed

Initial industry adaptation of column-stores

Column-stores and row-stores are Both a norm

Timeline

First row-store Developed and productized

First open-source Column-store developed

Initial industry adaptation of column-stores

Column-stores and row-stores are Both a norm

Why the 30 year gap?

First row-store Developed and productized

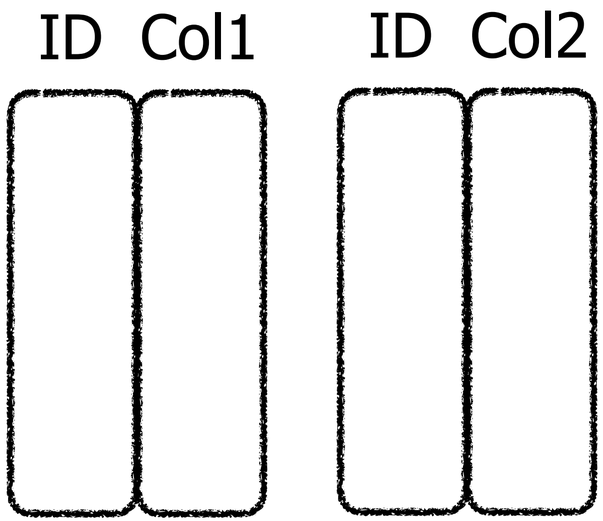
First open-source Column-store developed

Initial industry adaptation of column-stores

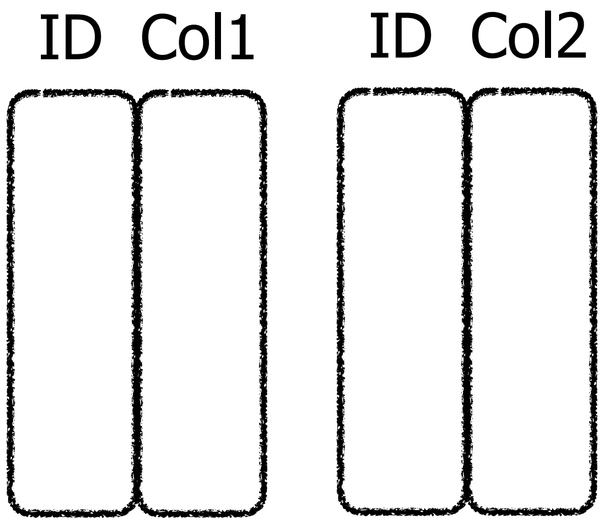
Column-stores and row-stores are Both a norm

Why the 30 year gap? **Storage density grew, so we store far more data now. Processing it efficiently is more critical. This drives a need for specialization.**

Should we store each column with a materialized ID?



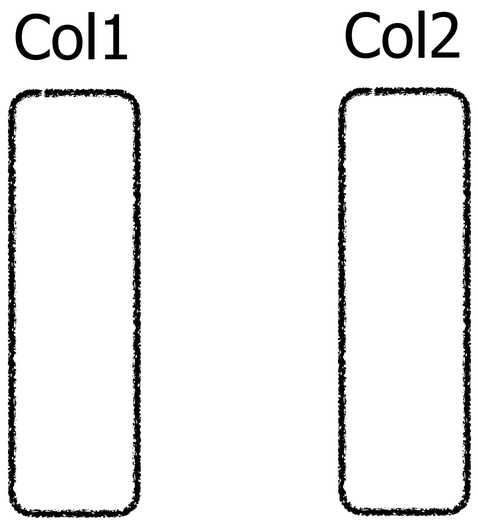
Should we store each column with a materialized ID?



No, this would slow down queries by reading more data.

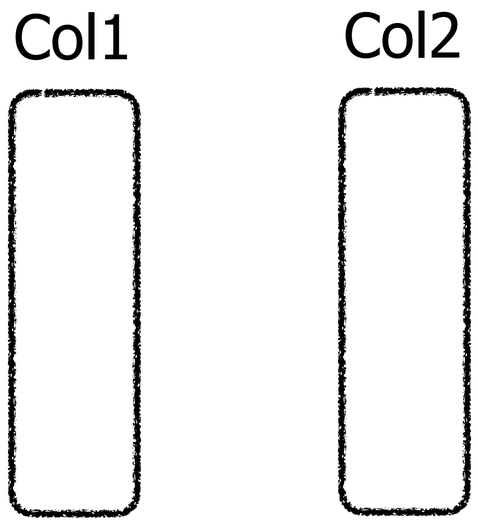
Instead use positional alignment

$$Col1(i) = i \cdot width(Col1)$$



Instead use positional alignment

Col1(i) = i · width(Col1)

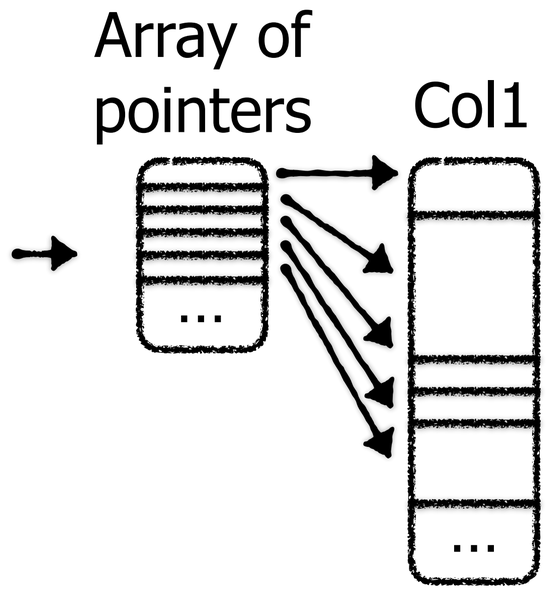


What is a column is var-length?

In case of variable-length data, we can employ indirection

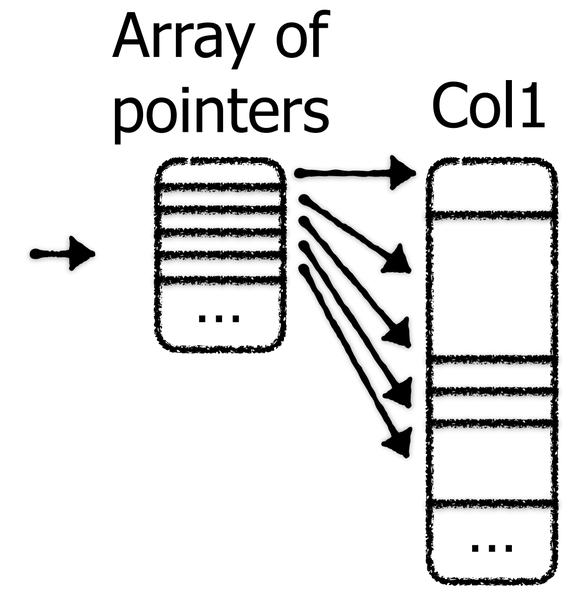
(unrecommended)

Col1(i) = i · width(ptr)



In case of variable-length data, we can employ indirection
(unrecommended)

Col1(i) = i · width(ptr)

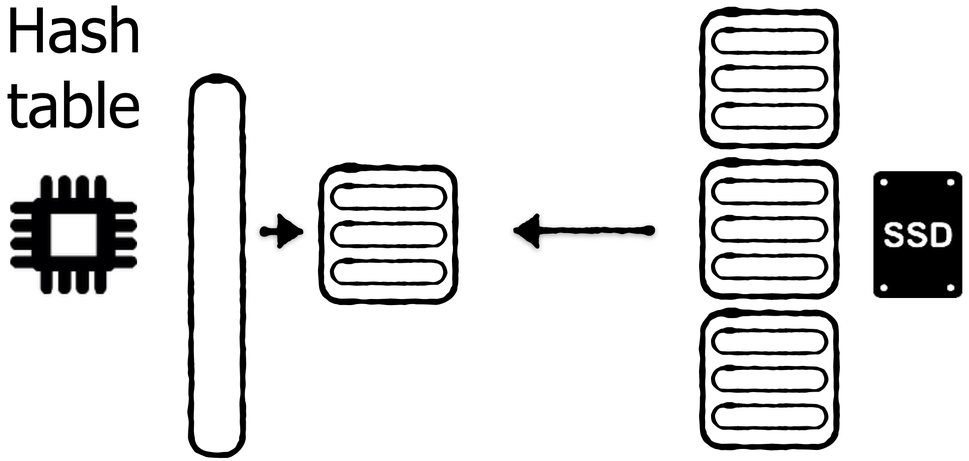


Best to define a schema with fixed-length fields if possible
prevent indirection overheads

How should we handle the buffer pool (BP)?

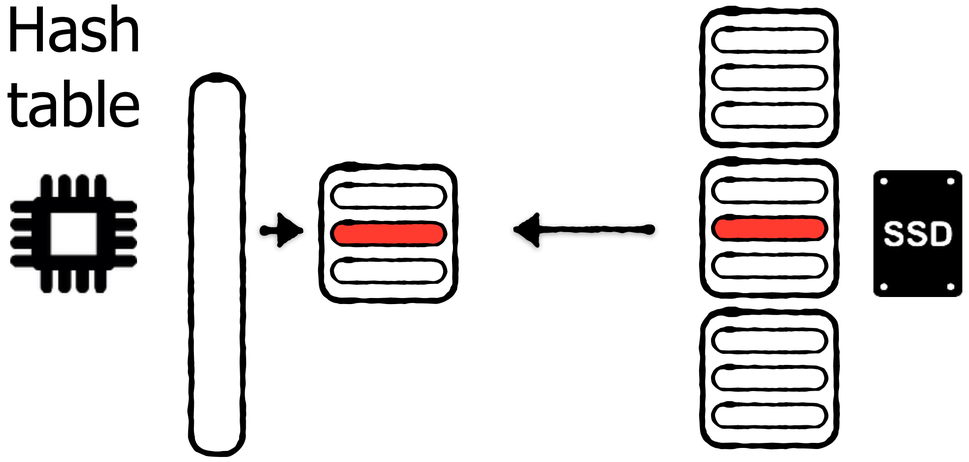
How should we handle the buffer pool (BP)?

A row-store BP maps 4KB pages.



How should we handle the buffer pool (BP)?

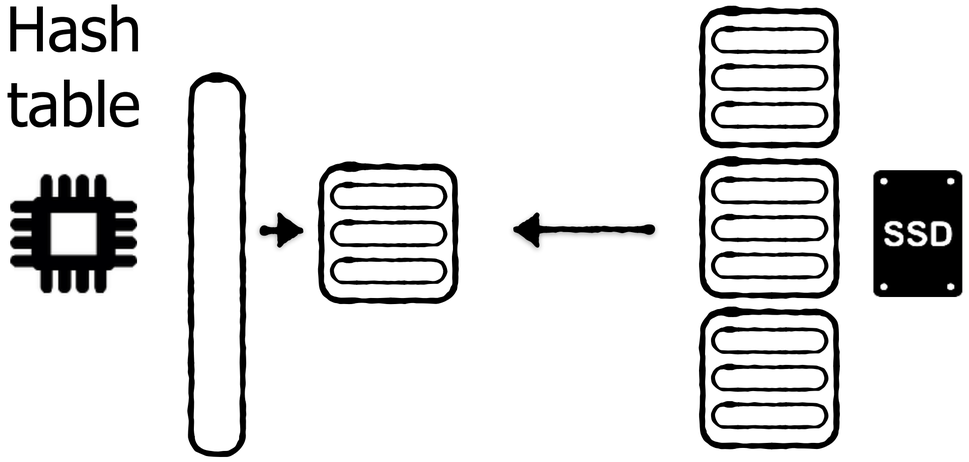
A row-store BP maps 4KB pages.



This makes sense each query is "selective" (accesses few rows)

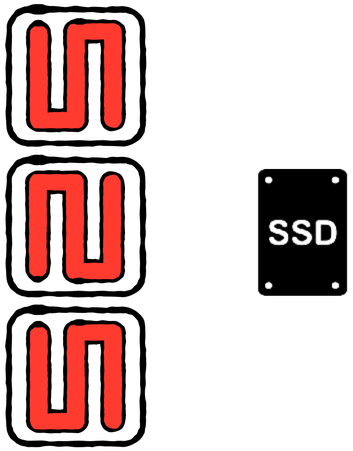
How should we handle the buffer pool (BP)?

A row-store BP maps 4KB pages.



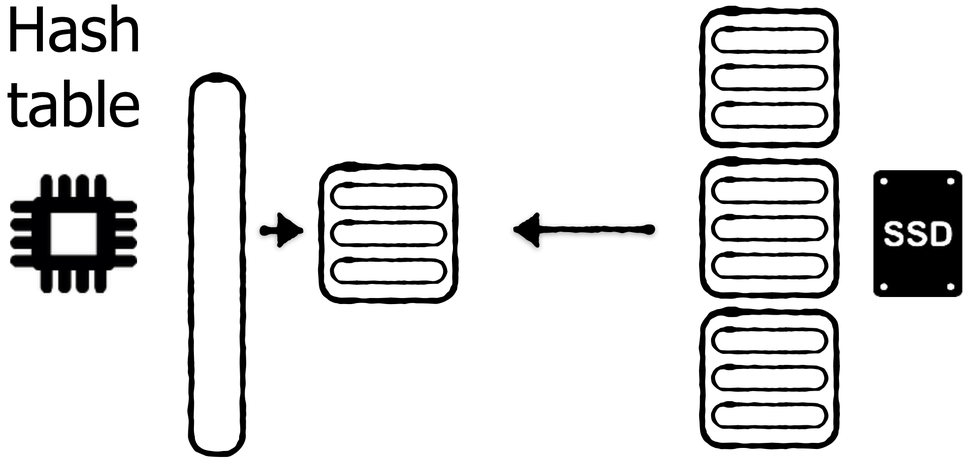
This makes sense each query is "selective" (accesses few rows)

In column-stores, each column spans multiple pages & queries are unselective



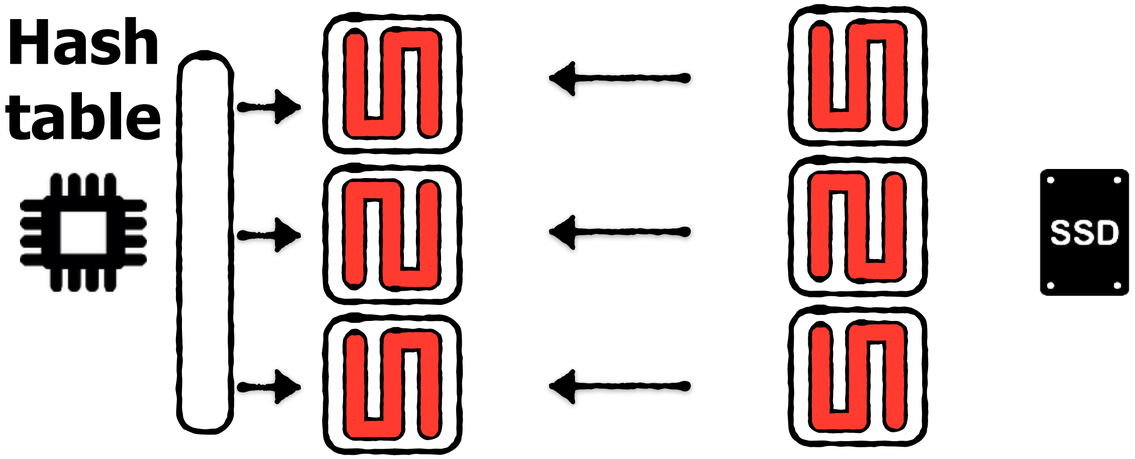
How should we handle the buffer pool (BP)?

A row-store BP maps 4KB pages.



This makes sense each query is "selective" (accesses few rows)

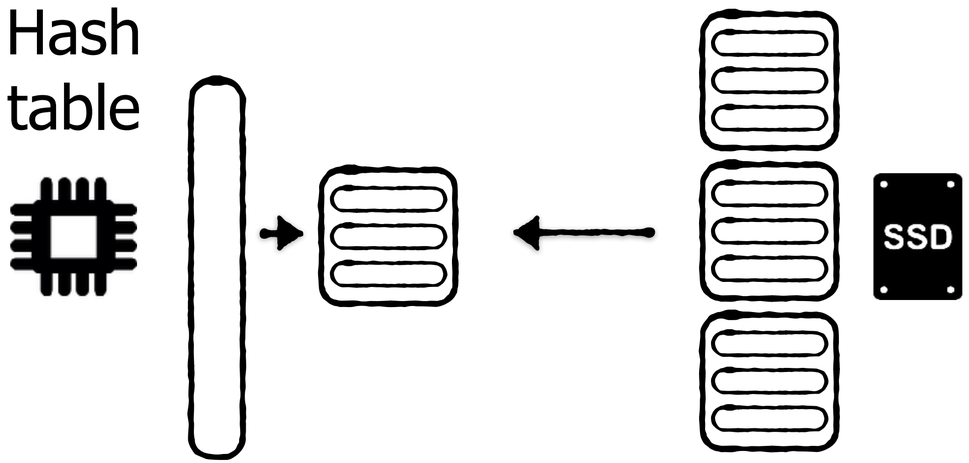
In column-stores, each column spans multiple pages & queries are unselective



Scanning a column in memory would require hashing overheads for each page.

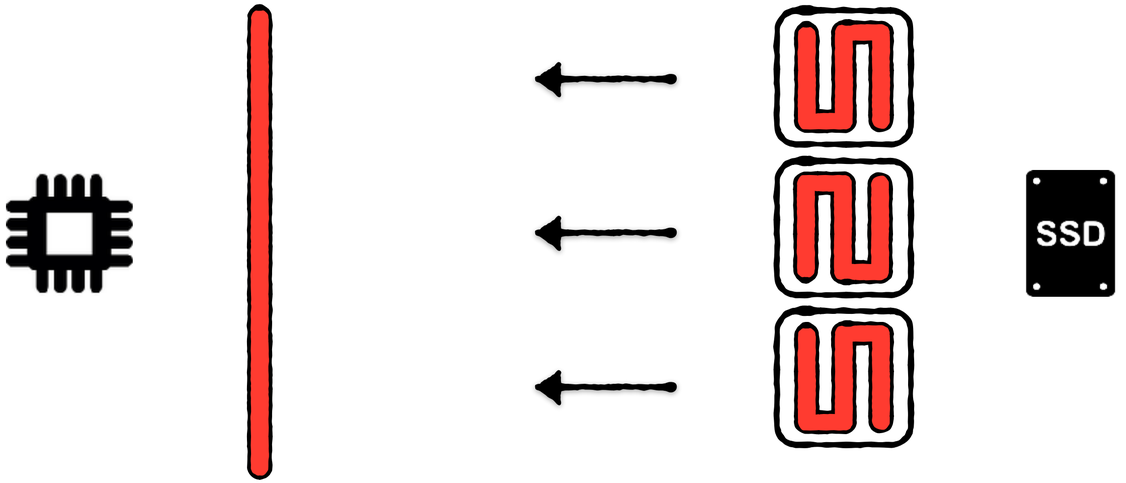
How should we handle the buffer pool (BP)?

A row-store BP maps 4KB pages.



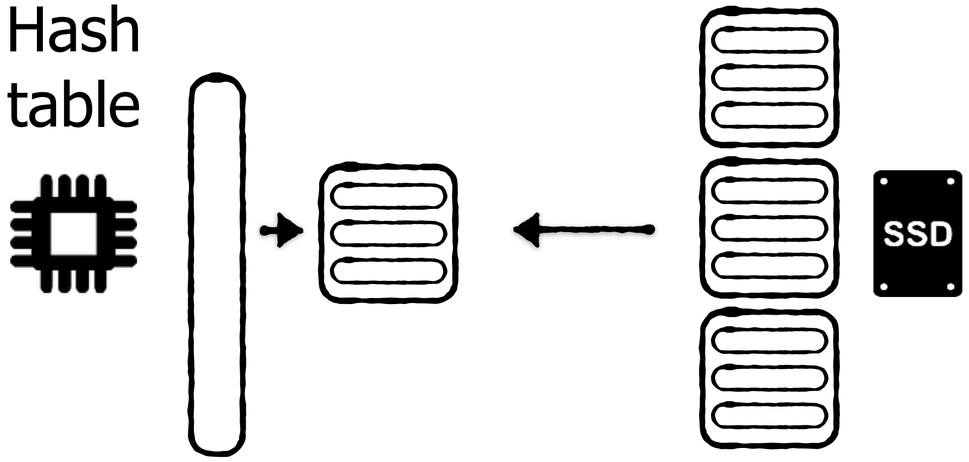
This makes sense each query is "selective" (accesses few rows)

Better to read and map at column-granularity (or multiple pages thereof)



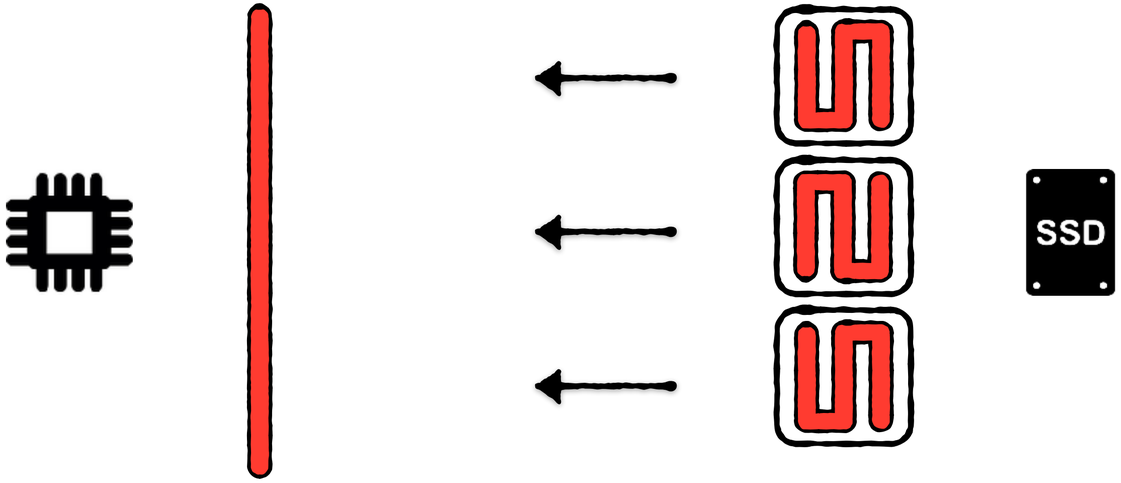
How should we handle the buffer pool (BP)?

A row-store BP maps 4KB pages.



This makes sense each query is "selective" (accesses few rows)

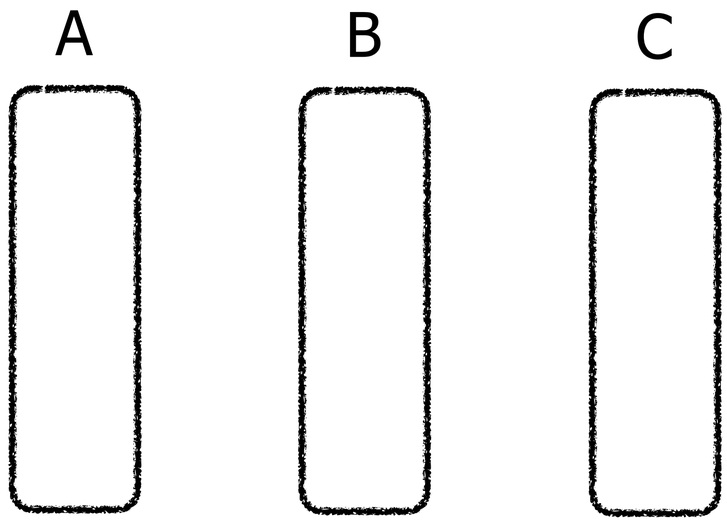
Better to read and map at column-granularity (or multiple pages thereof)



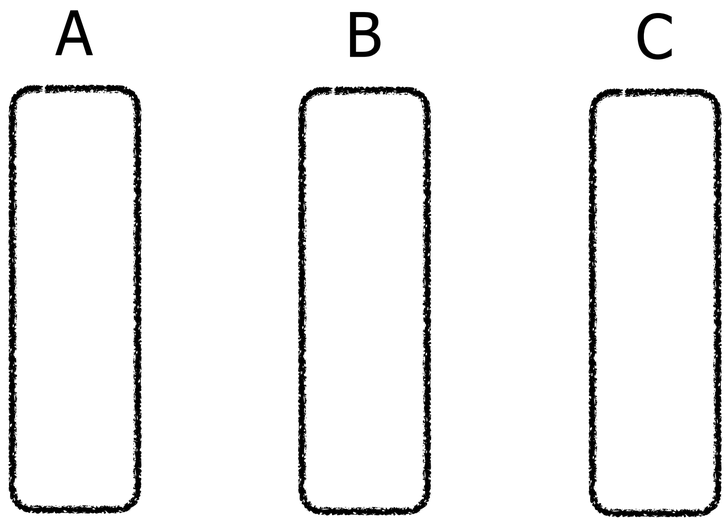
Can do this by mapping columns in virtual memory or in the BP

How to process queries over a single table?

Select min(C) from table where A > 10 and B < 20

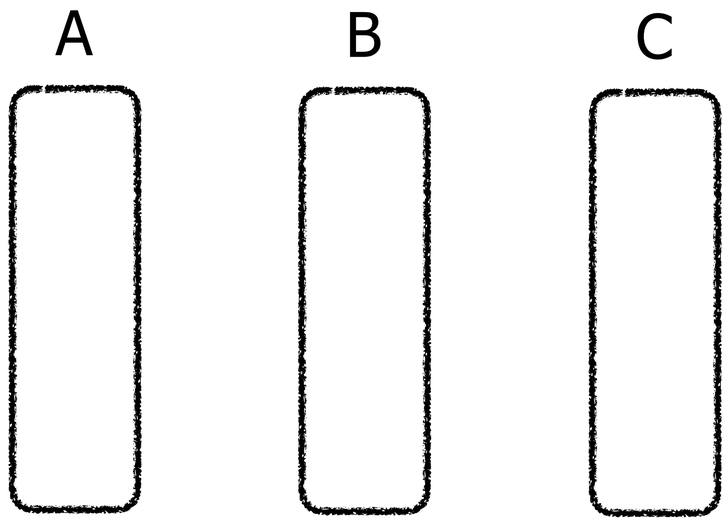


Select min(C) from table where A > 10 and B < 20



How shall we process this?

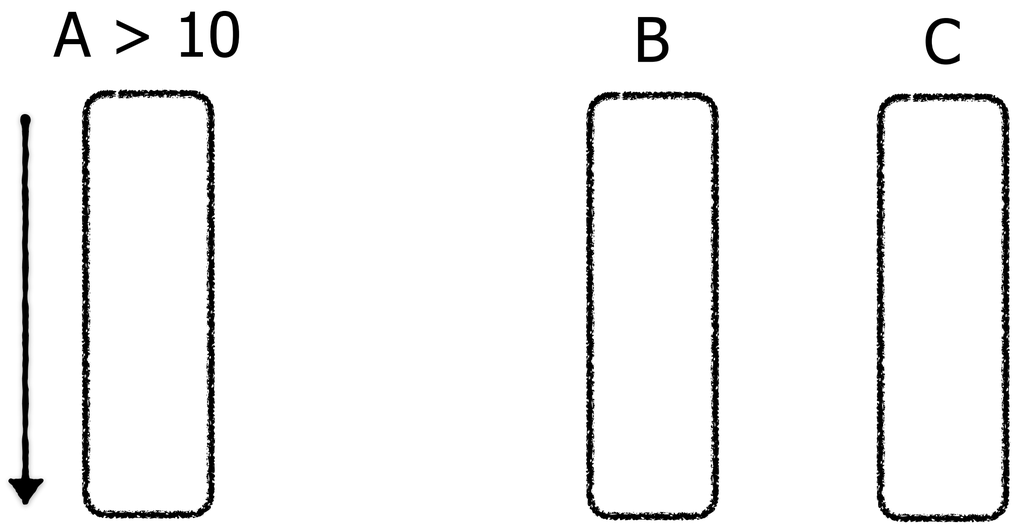
Select min(C) from table where A > 10 and B < 20



Early Materialization: return each qualifying row immediately

Select min(C) from table where A > 10 and B < 20

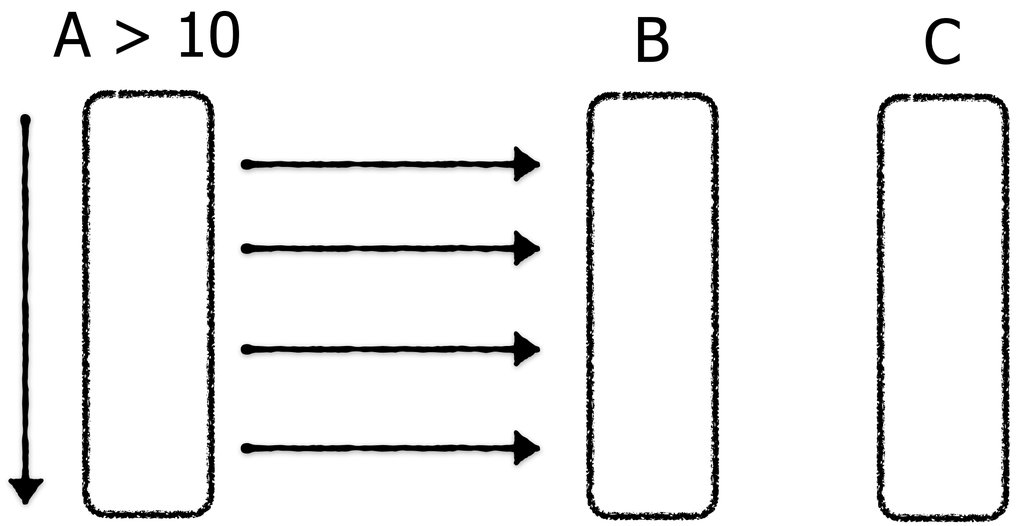
Scan & evaluate predicate



Early Materialization: return each qualifying row immediately

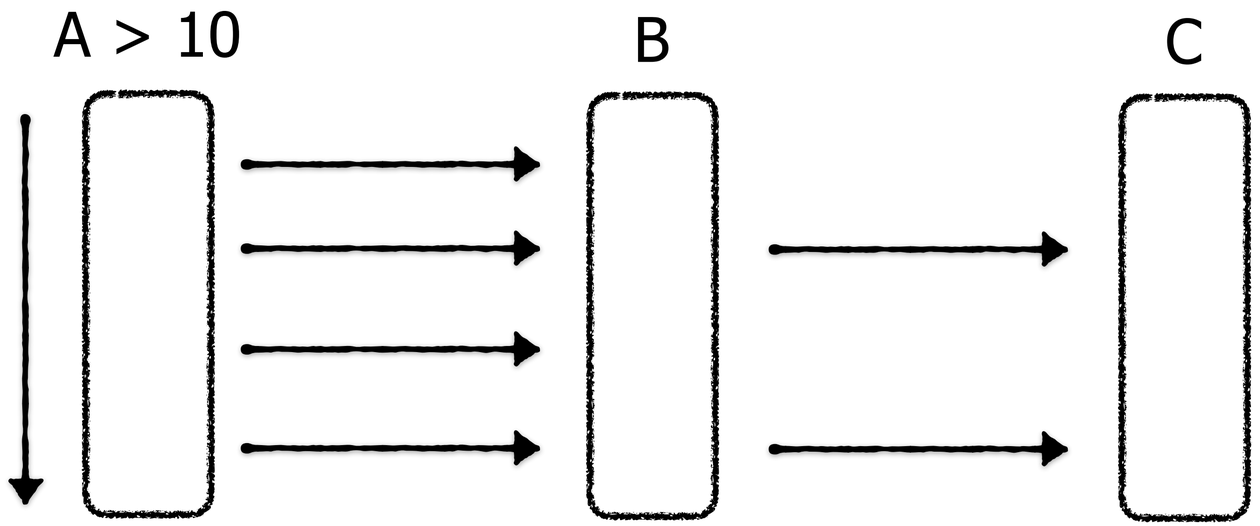
Select min(C) from table where A > 10 and B < 20

Scan & evaluate predicate



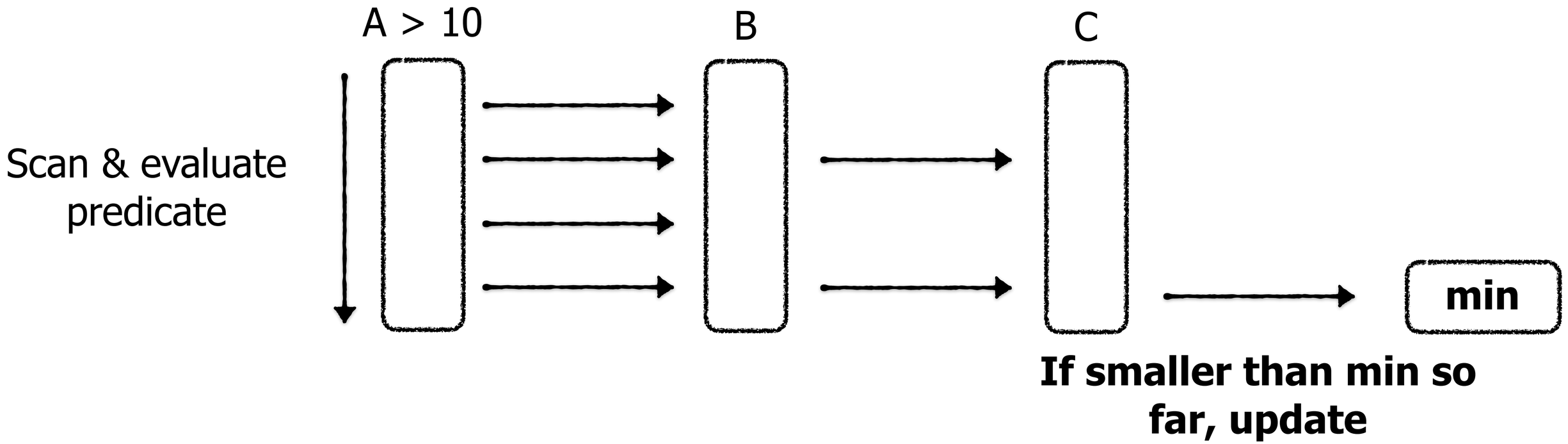
For each qualifying tuple on A, check if B also qualifies

Select min(C) from table where A > 10 and B < 20

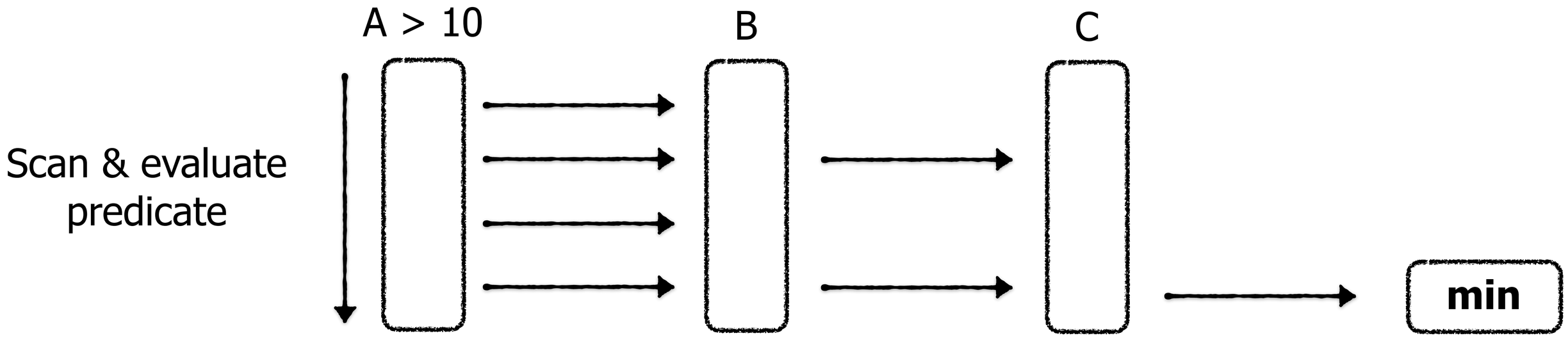


For each qualifying tuple on A and B, fetch C

Select min(C) from table where A > 10 and B < 20

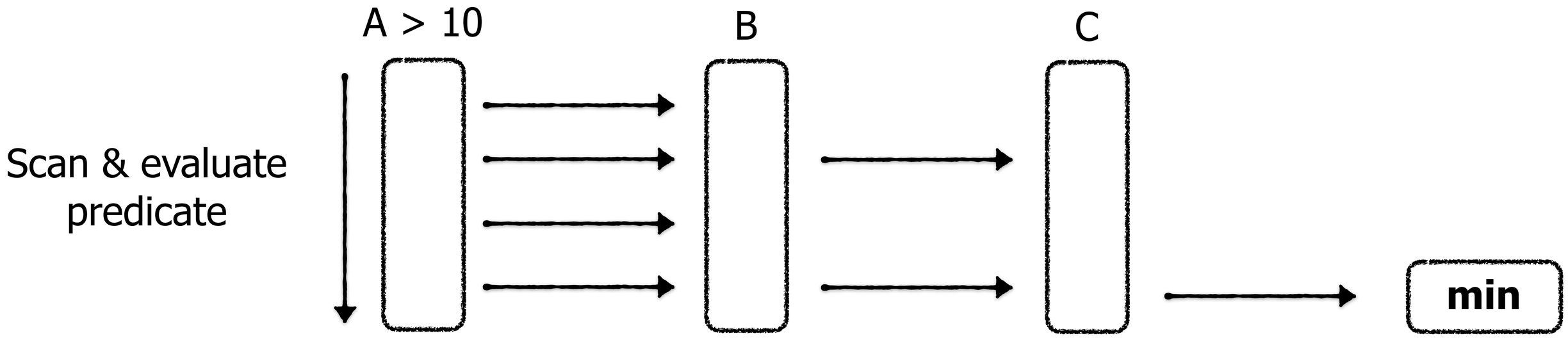


Select min(C) from table where A > 10 and B < 20



Problem?

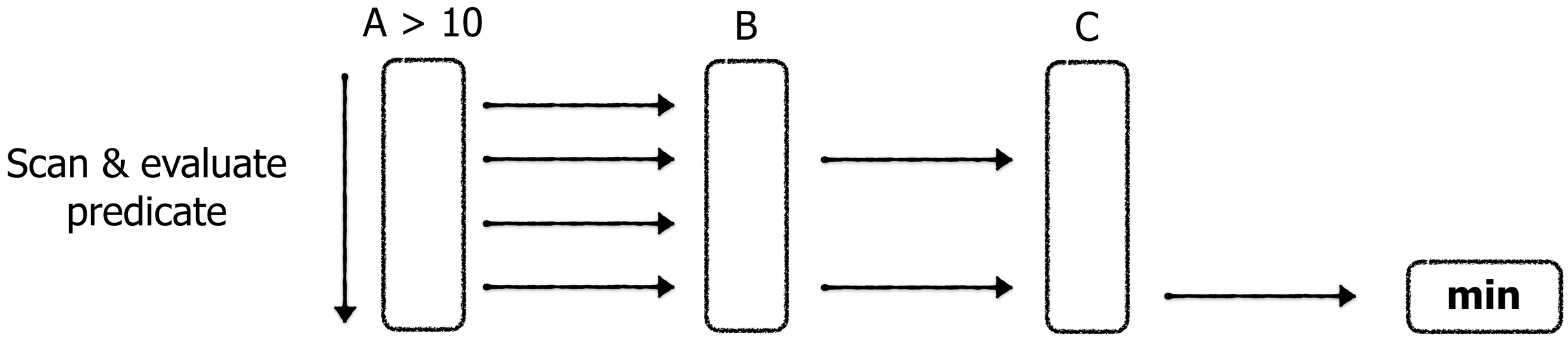
Select min(C) from table where A > 10 and B < 20



Problem? **Potentially more random I/Os**

More CPU branch mispredictions

Select min(C) from table where A > 10 and B < 20

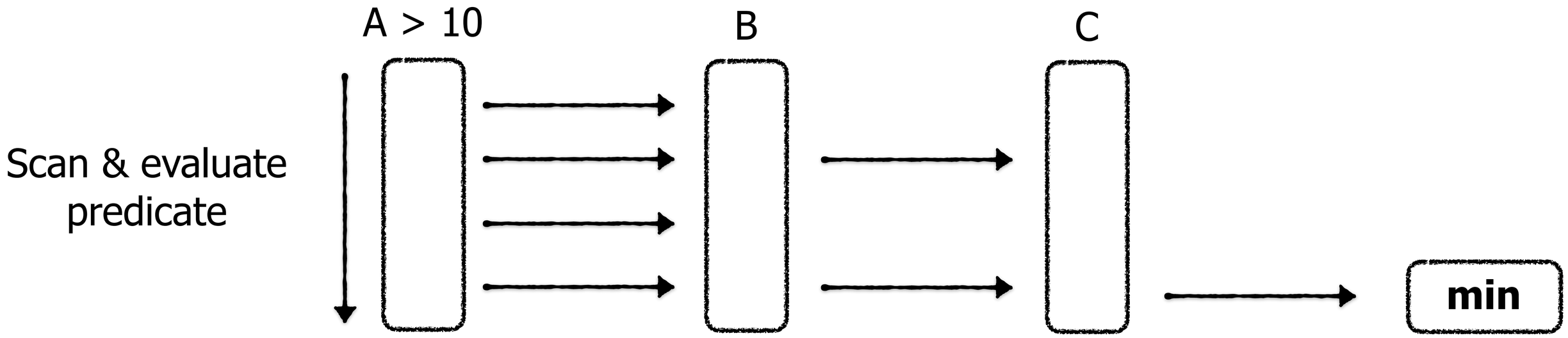


Problem? **Potentially more random I/Os**

More CPU branch mispredictions

Incompatible with SIMD (more later)

Select min(C) from table where A > 10 and B < 20



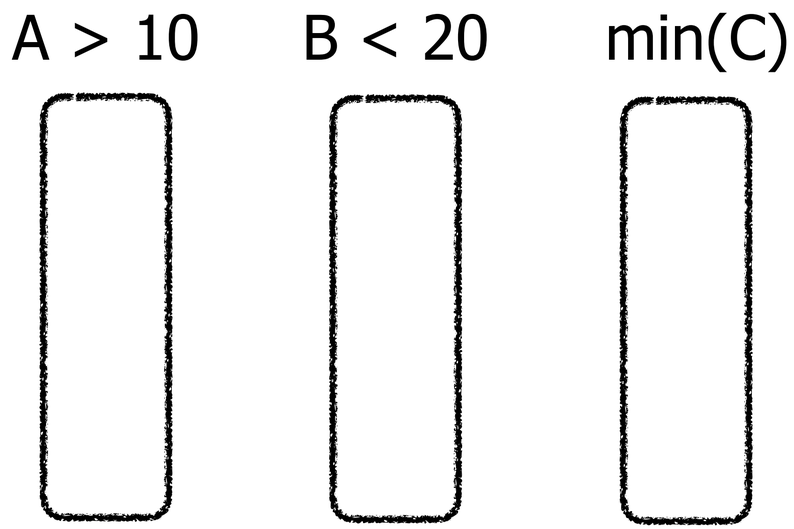
Problem? **Potentially more random I/Os**

More CPU branch mispredictions

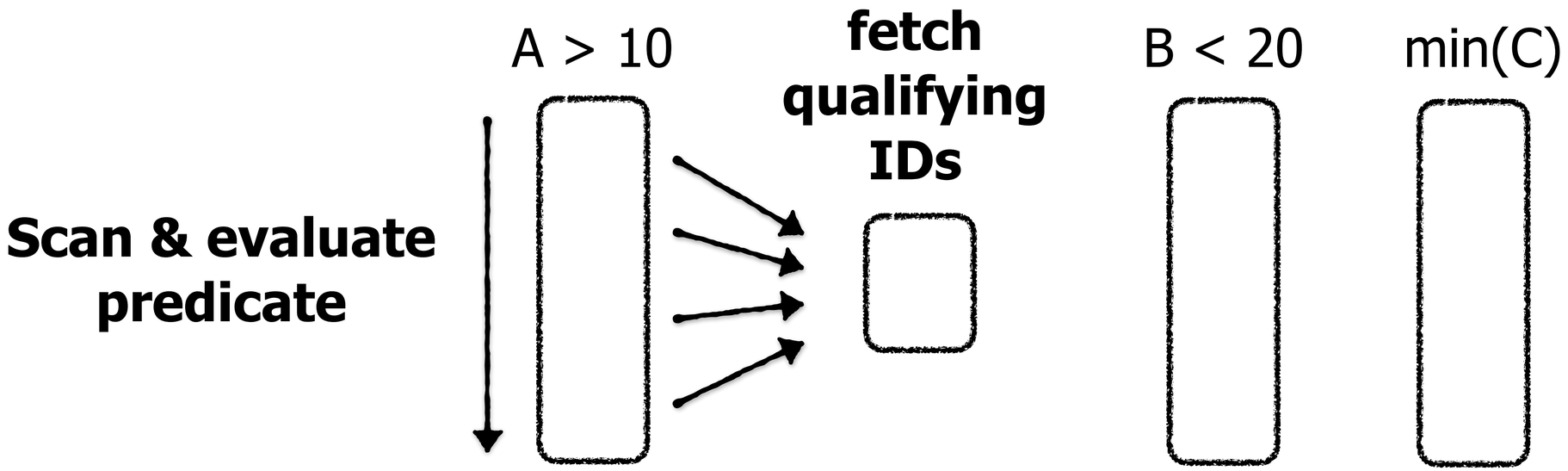
Incompatible with SIMD (more later)

Solutions?

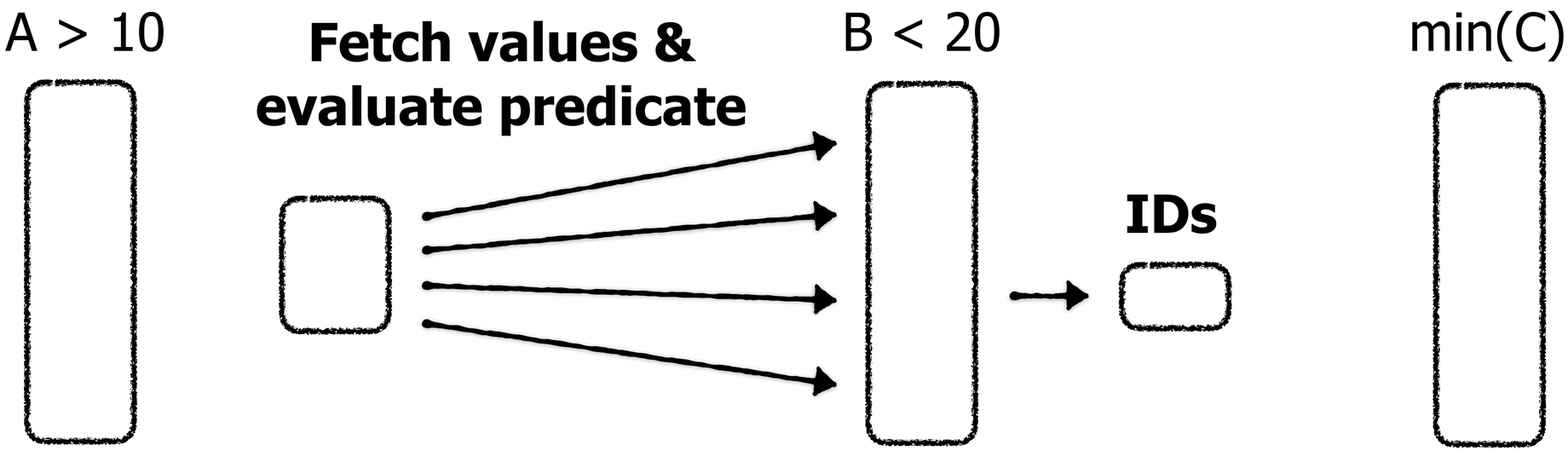
Late Materialization: process one column at a time



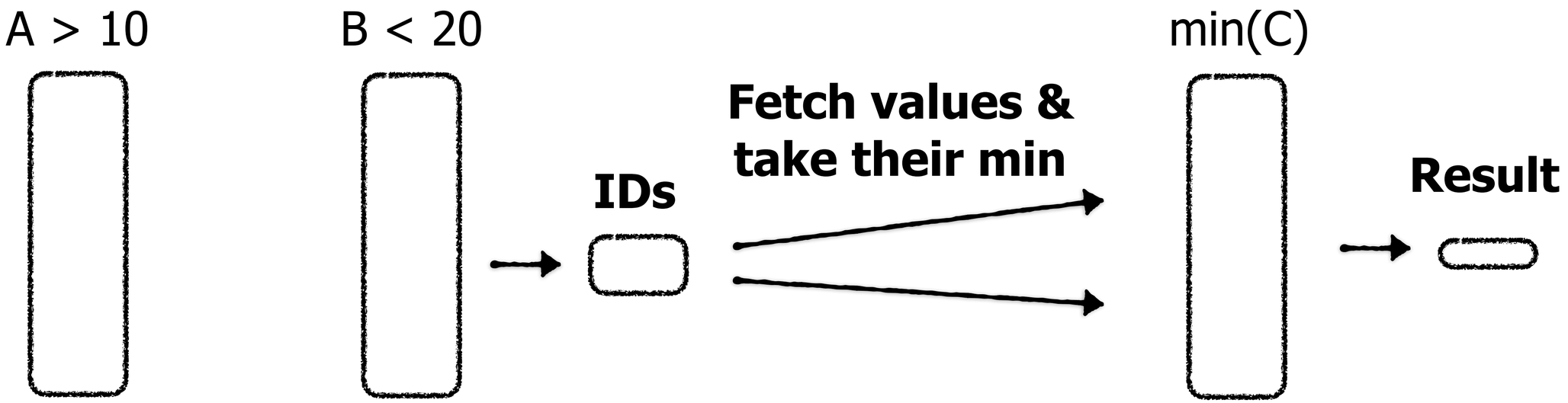
Late Materialization



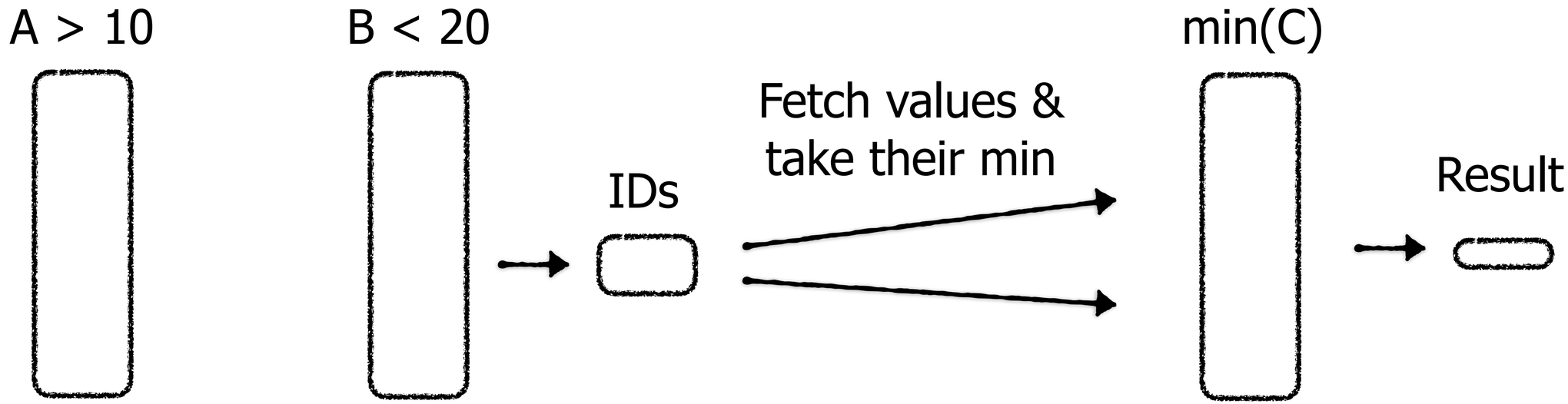
Late Materialization

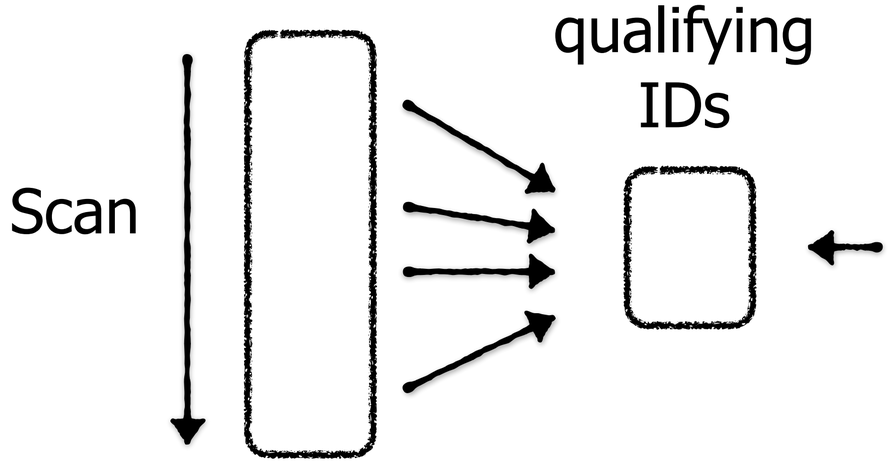


Late Materialization

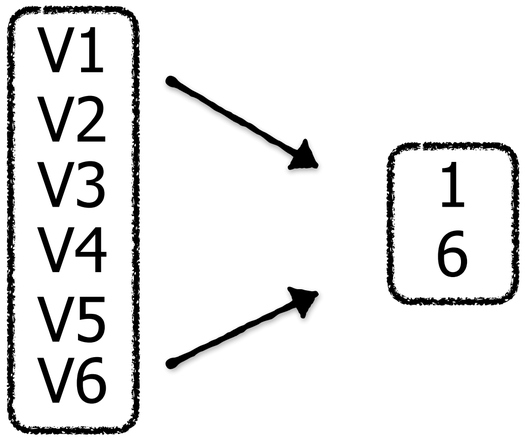


Late Materialization requires more memory to withhold intermediate results, but entails better cache behavior & CPU efficiency

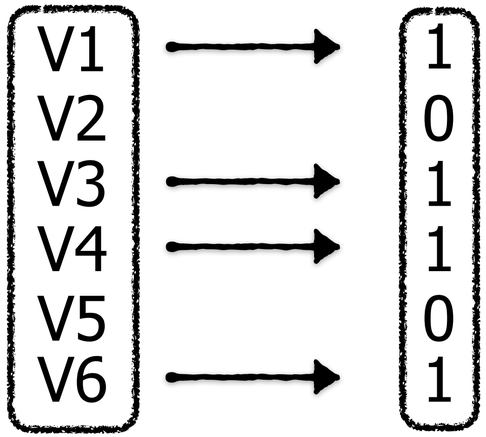




Options for structuring intermediate results ?

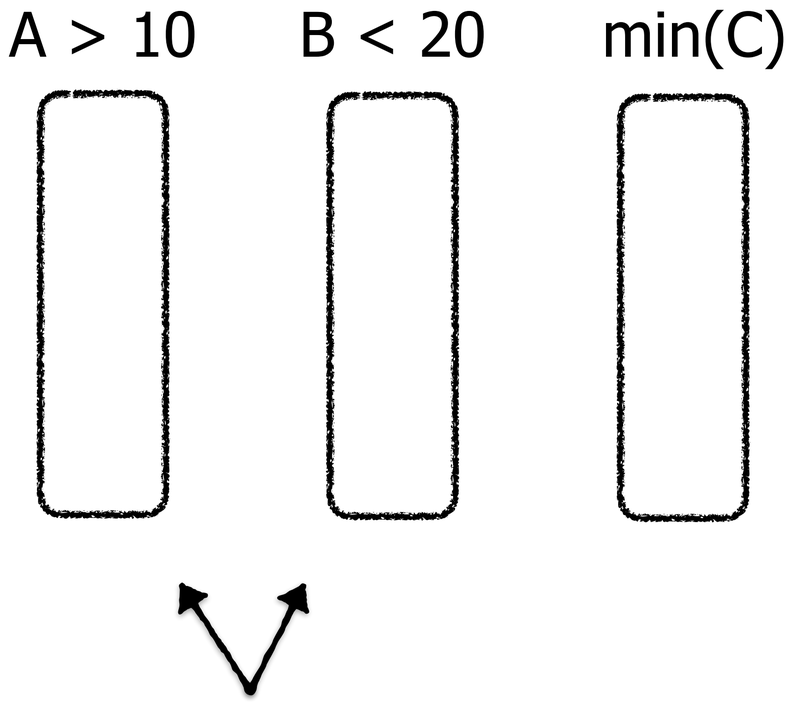


(1) Array of integer IDs
good when few results qualify



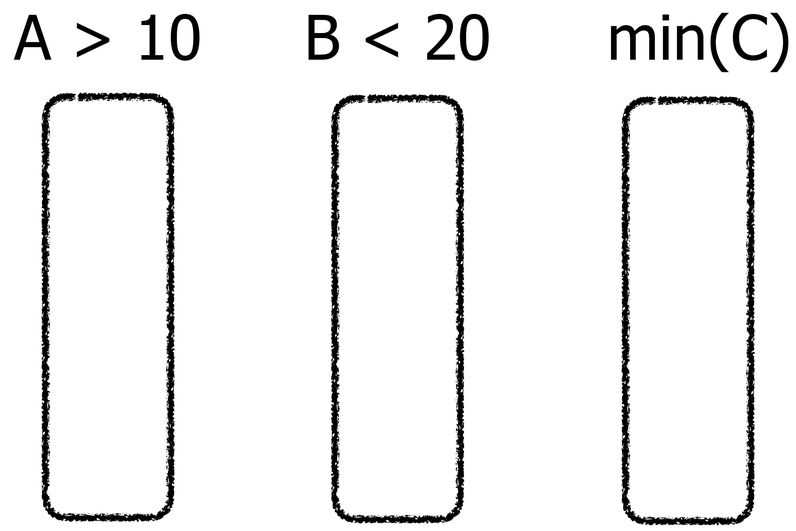
(2) Bitmap
good when many results qualify

Which column should we filter on first?



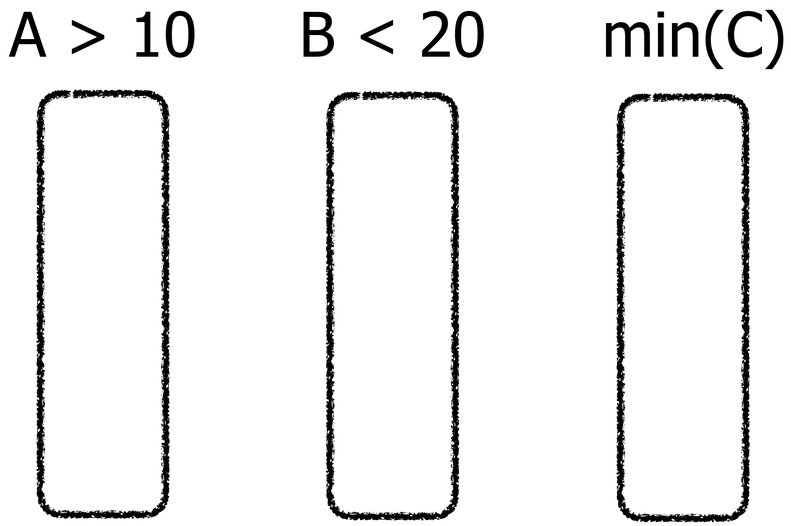
This one or this one?

Which column should we filter on first?



Filter on more selective columns first to reduce size of intermediate results and thus I/O

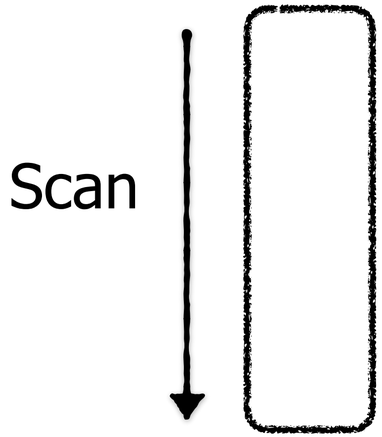
Which column should we filter on first?



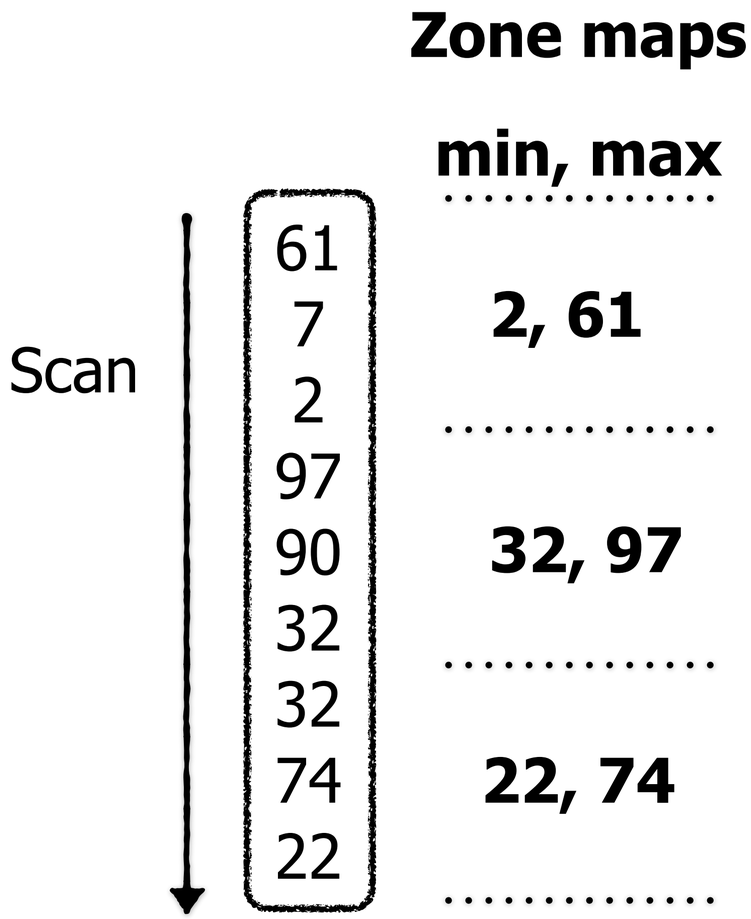
Filter on more selective columns first to reduce size of intermediate results and thus I/O

Can do this via cardinality estimation (e.g., histograms, count-min, etc)

How to speed up scans?

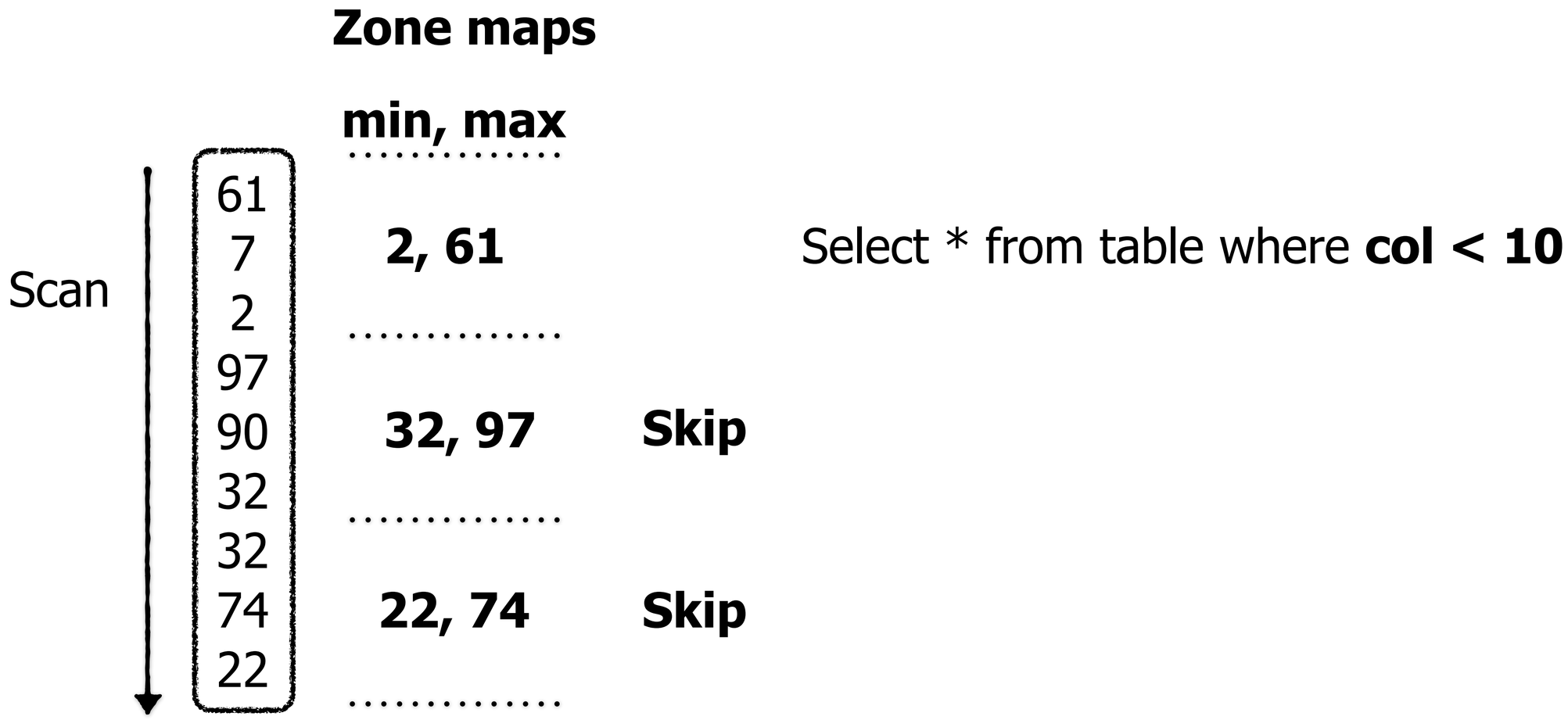


How to speed up scans?



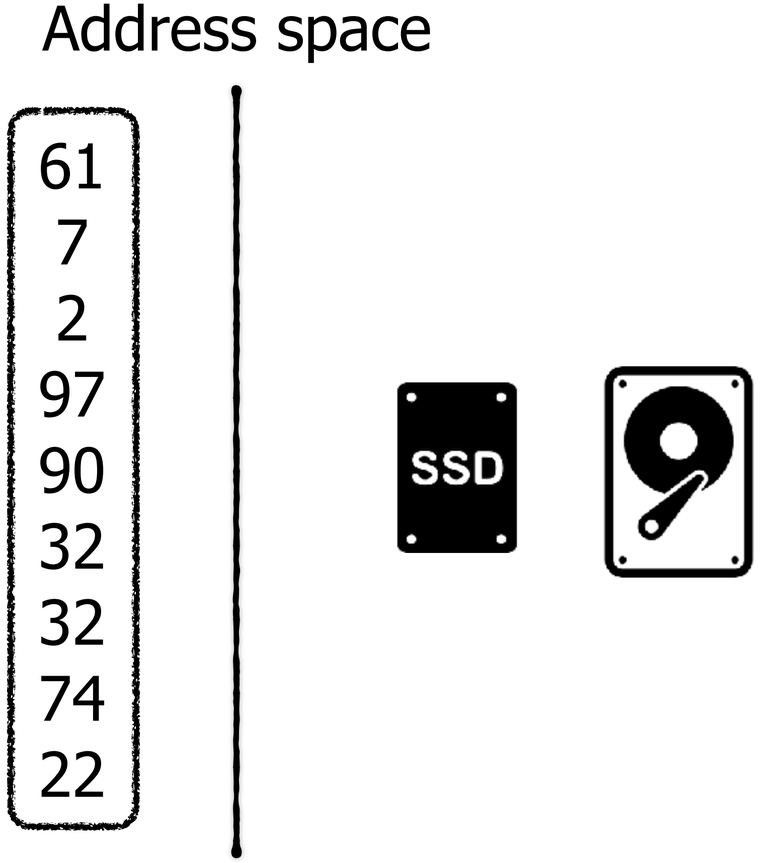
Equally sized partitions (MBs or GBs)

How to speed up scans?



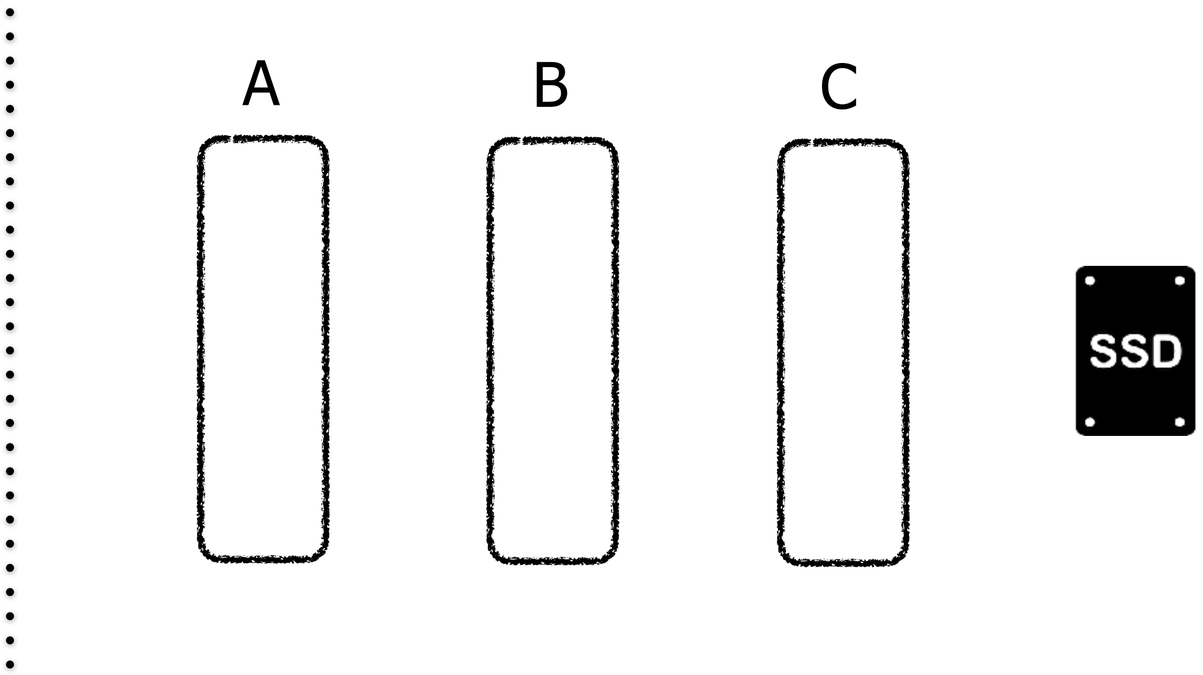
How to speed up scans?

Store columns sequentially to allow pre-fetching and I/O parallelism



Handling Insertions

Insert into table (a1,b1,c1)

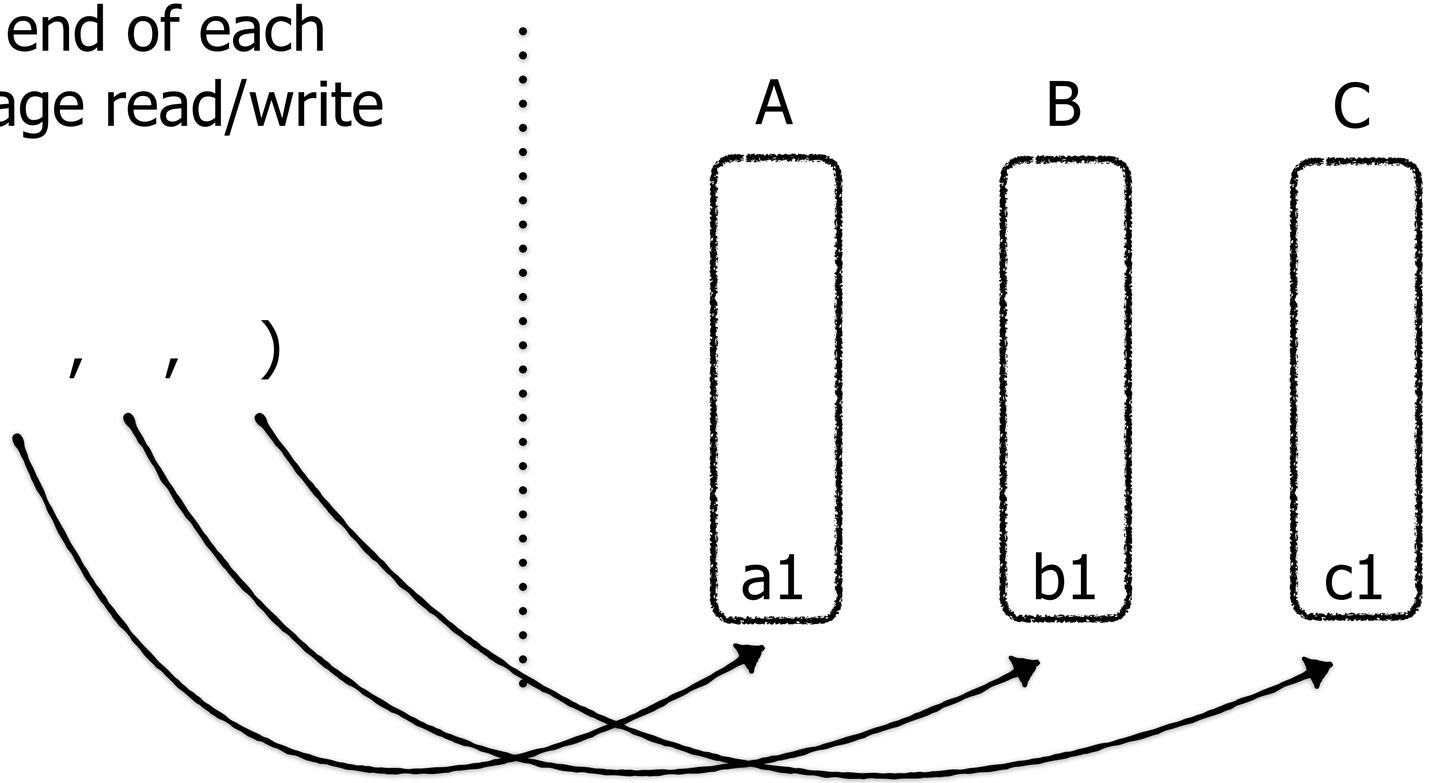


Handling Insertions

Option 1: In-Place Updates

Directly insert to end of each column with a storage read/write

Insert into table (, ,)



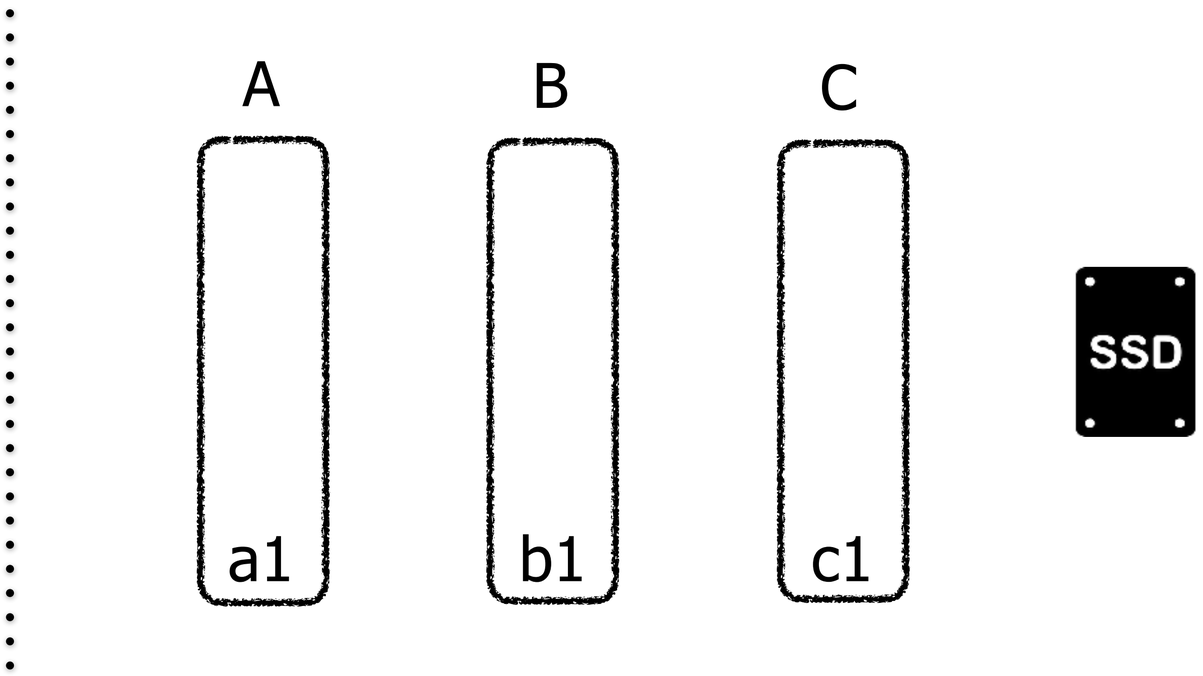
Handling Insertions

Option 1: In-Place Updates

Directly insert to end of each column with a storage read/write

Insert into table (, ,)

Cost model?



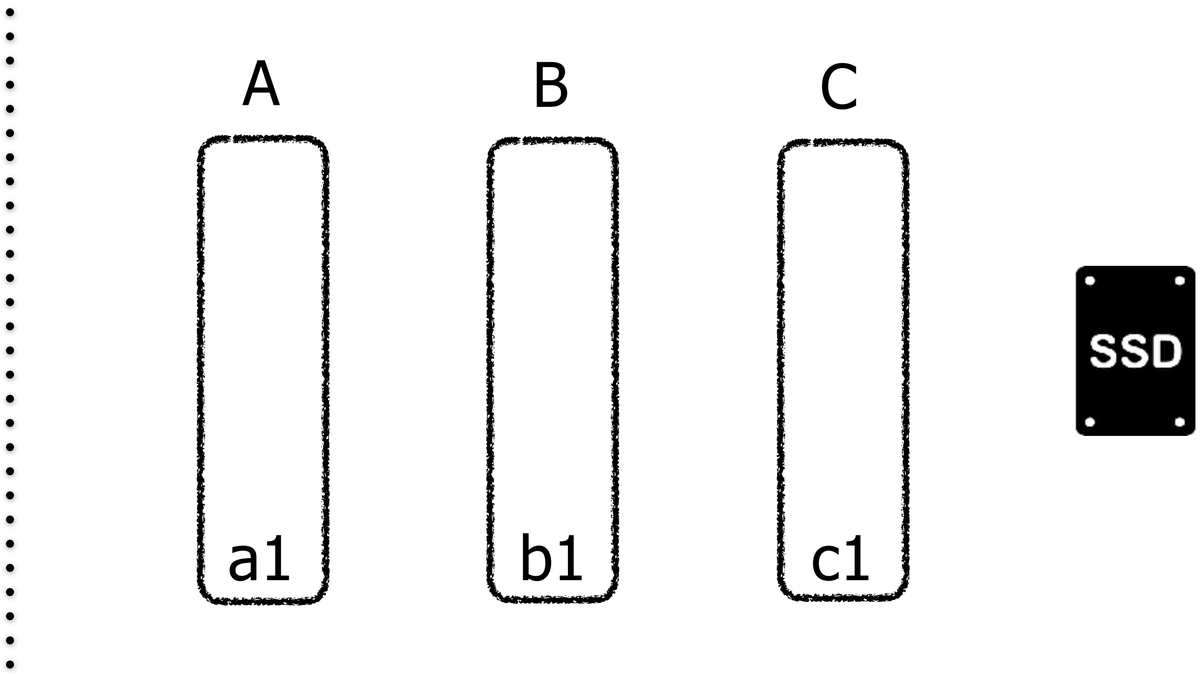
Handling Insertions

Option 1: In-Place Updates

Directly insert to end of each column with a storage read/write

Insert into table (, ,)

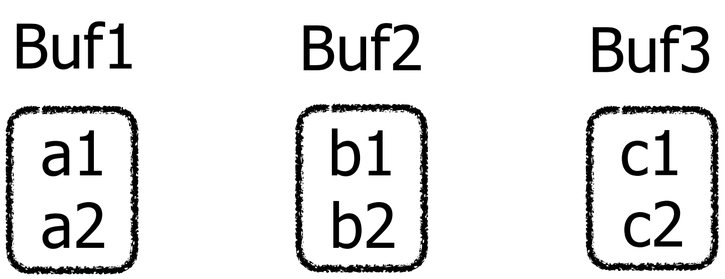
Cost model: O(#cols)

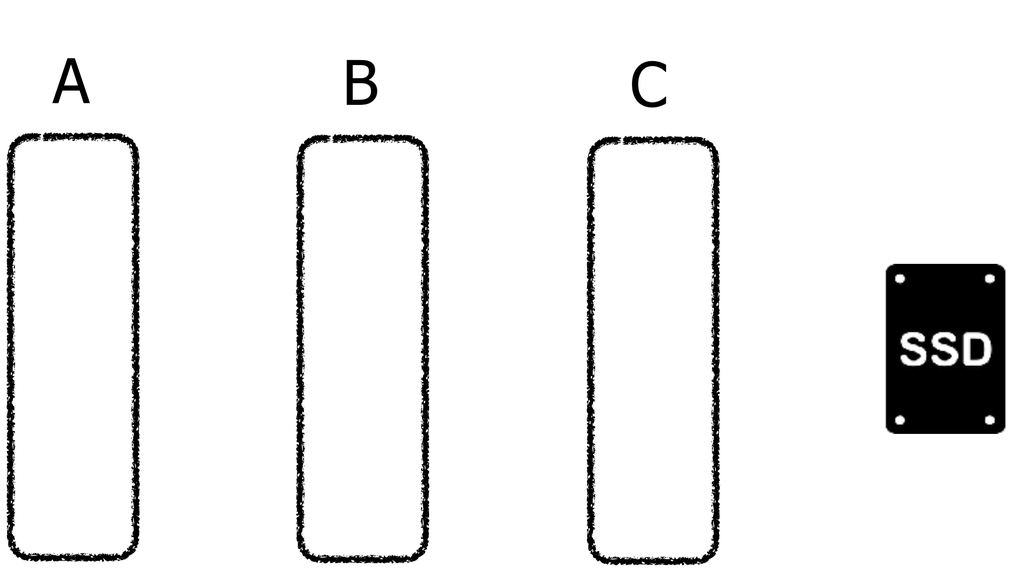


Handling Insertions

Option 2: In-memory Buffering

Insert into table (, ,)



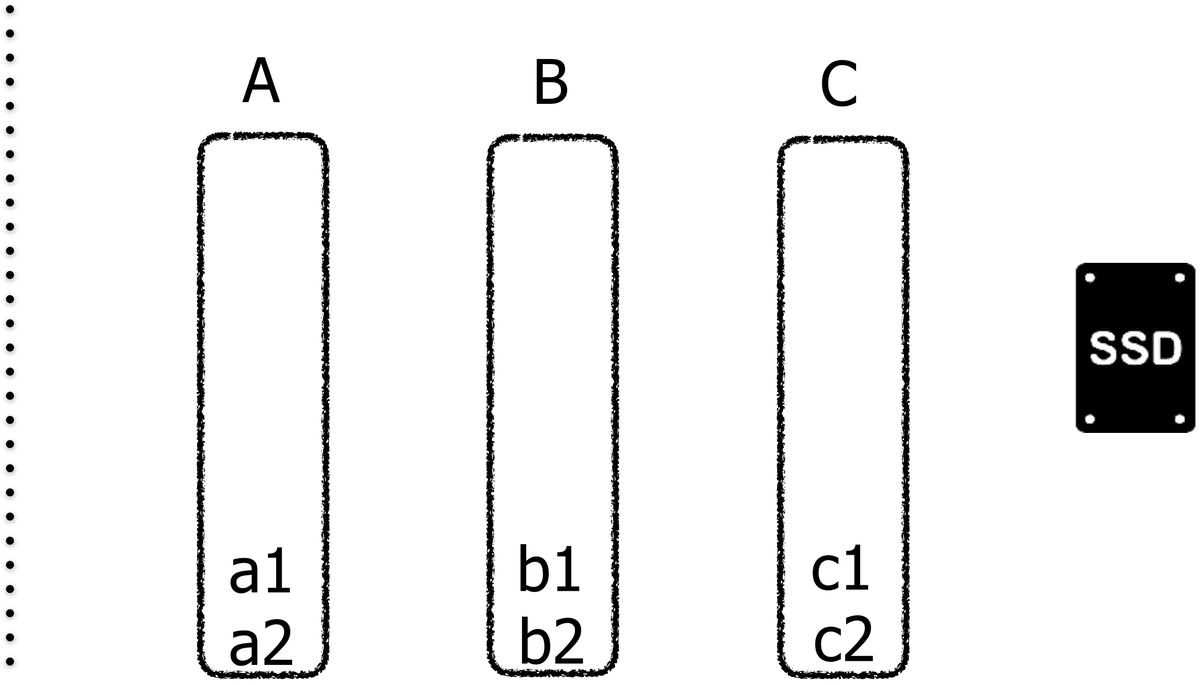


Handling Insertions

Option 2: In-memory Buffering

Insert into table (, ,)

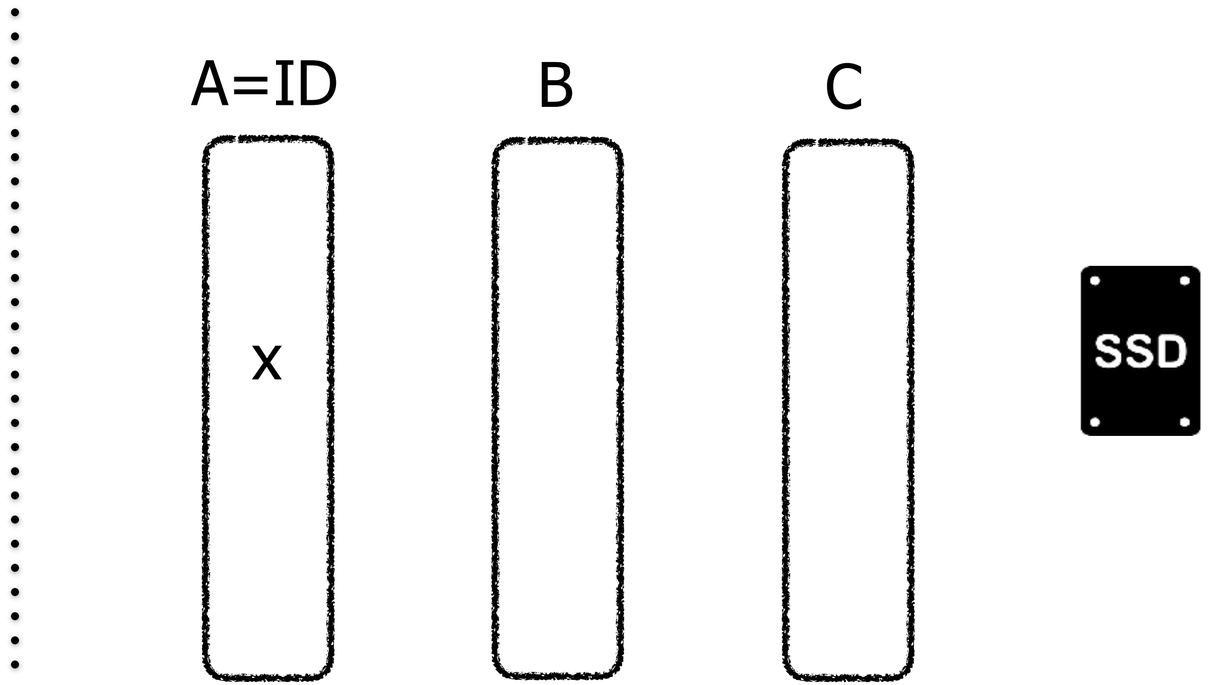
Cheap insertions are possible :)



Handling Deletes

delete from table where **A = “x”**

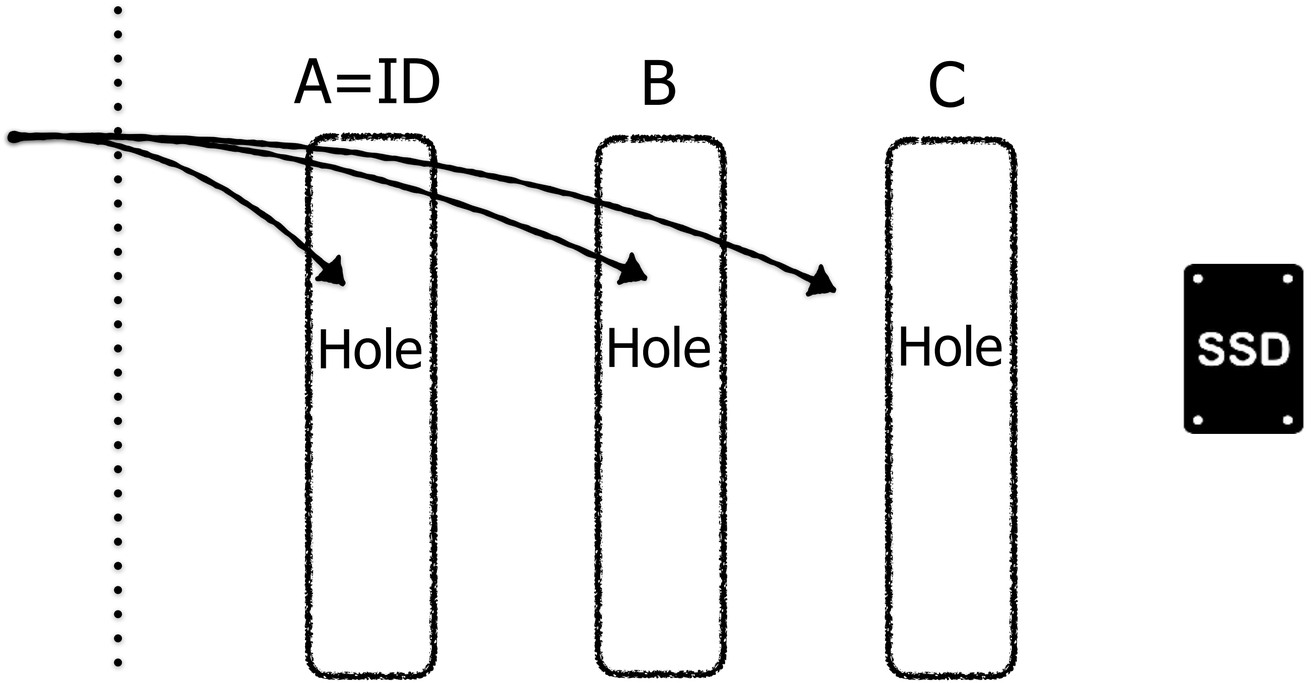
How would you do this?



Handling Deletes

Option 1: In-place Deletes

delete from table where A = “x”

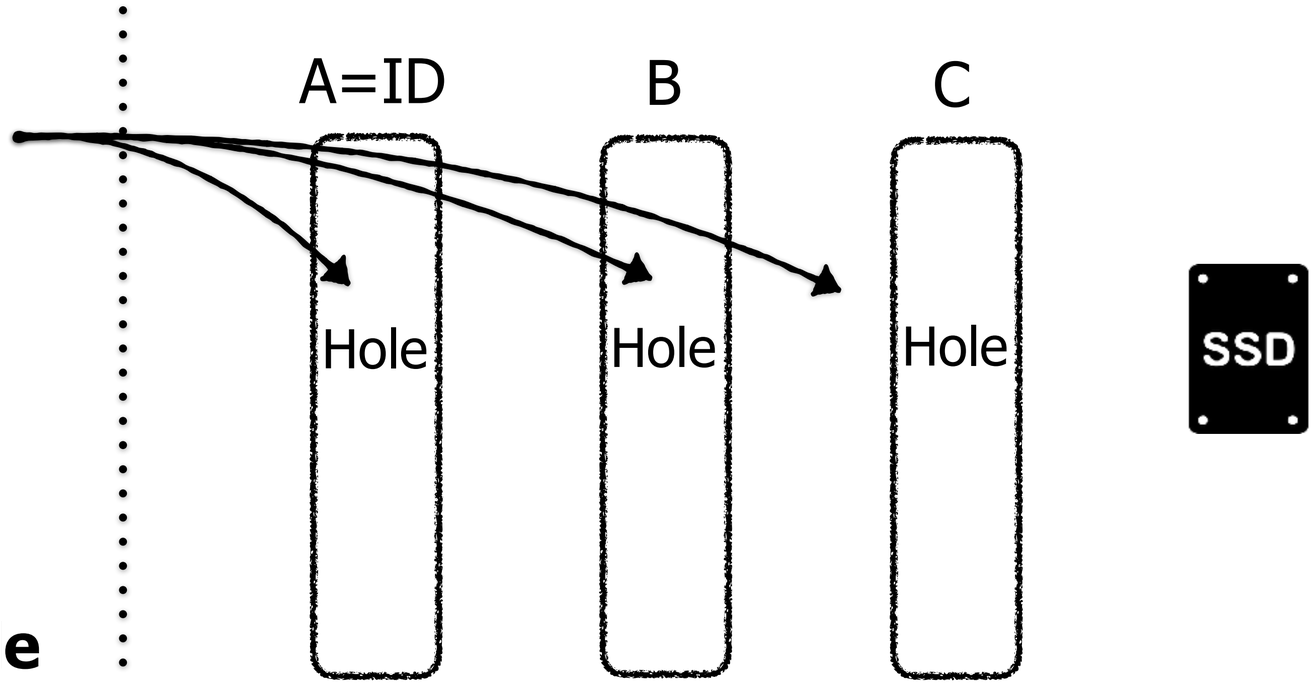


Problems:

Handling Deletes

Option 1: In-place Deletes

delete from table where A = “x”



Problems:

Cost: O(#cols) write I/Os

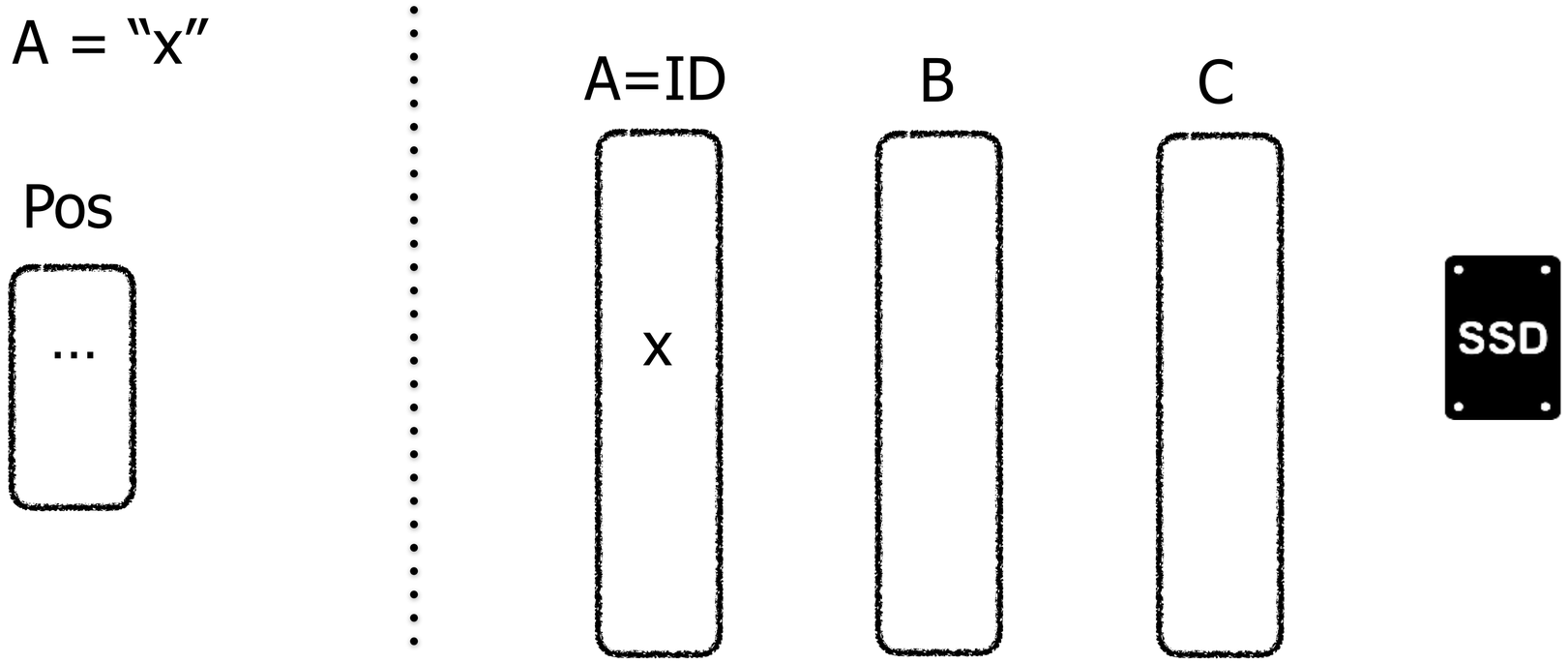
Holes slows down queries

Requires 1 extra bit to note a hole

Handling Deletes

Option 2: employ delete column

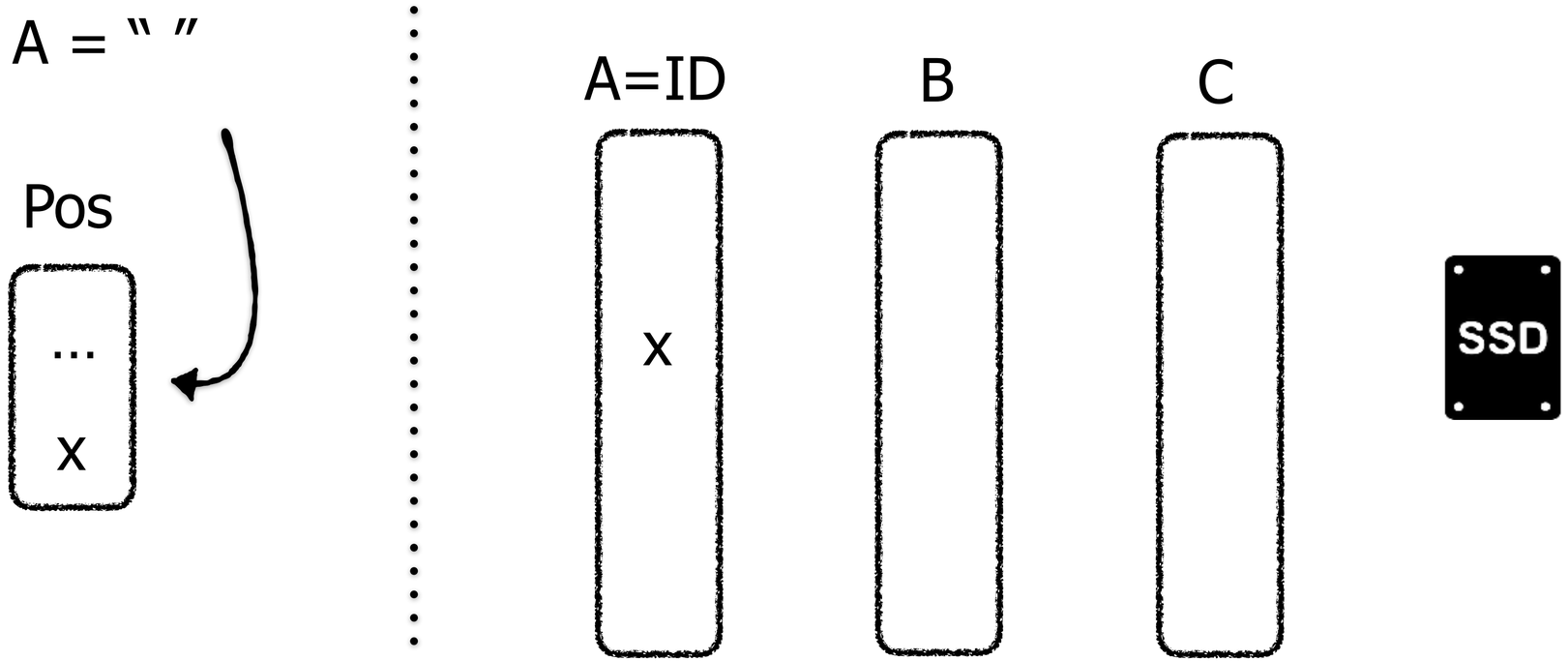
delete from table where A = “x”



Handling Deletes

Option 2: employ delete column

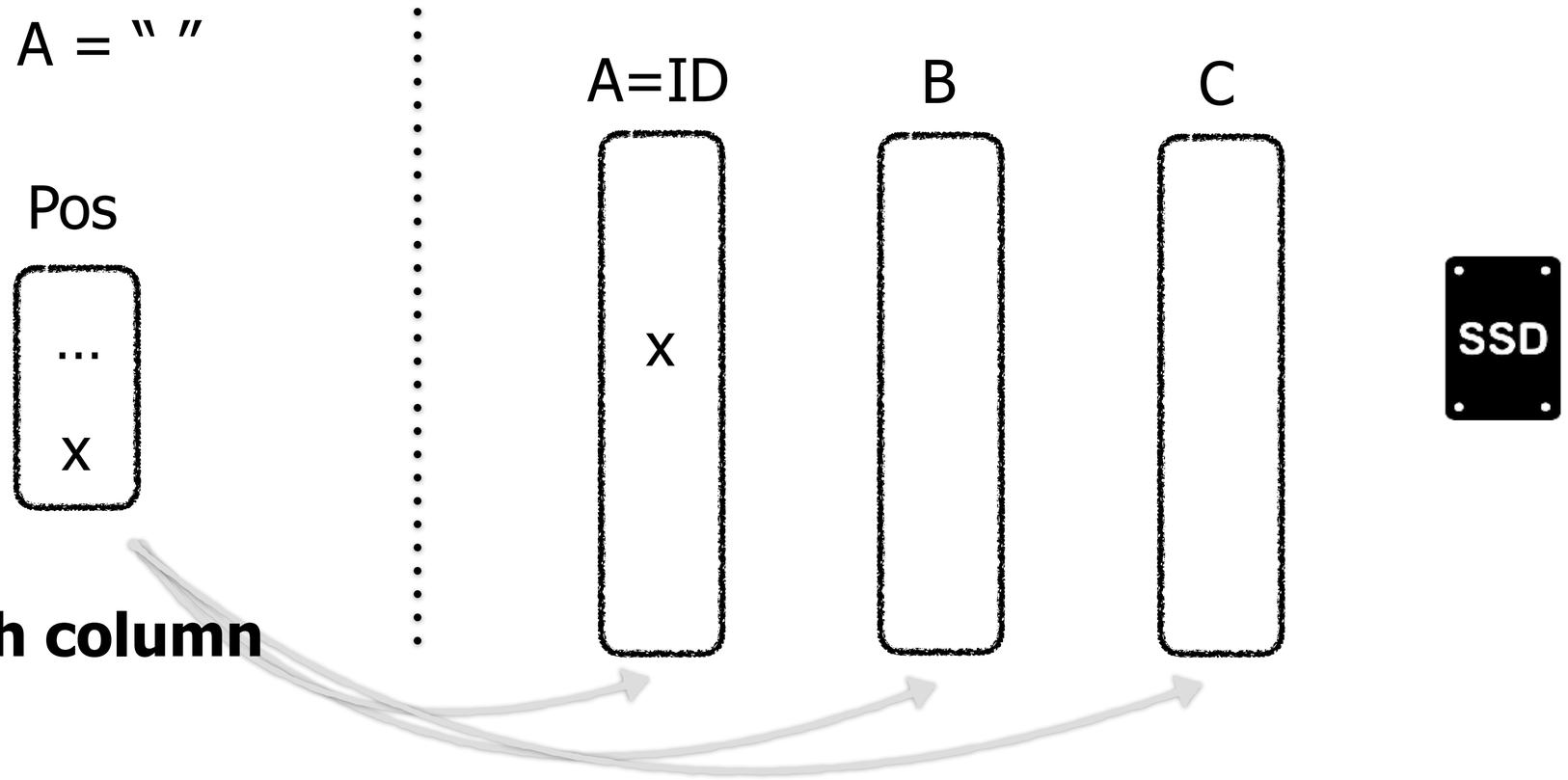
delete from table where A = “ ”



Handling Deletes

Option 2: employ delete column

delete from table where A = “ ”

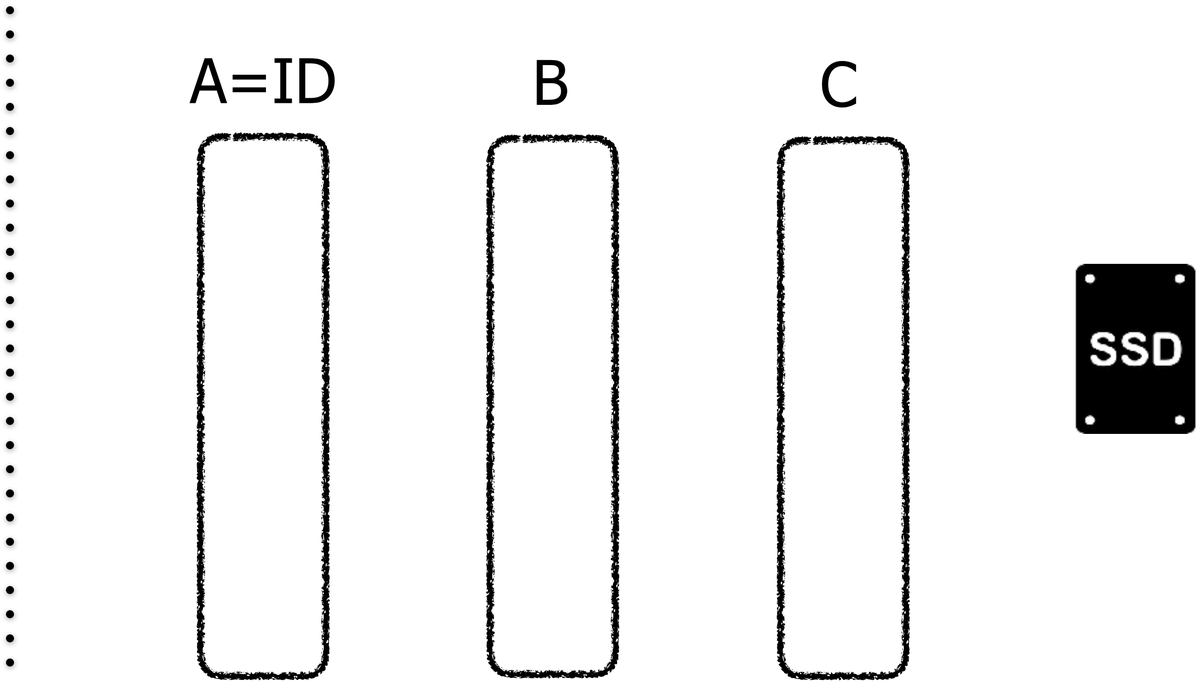


Eventually merge with column

Handling Updates

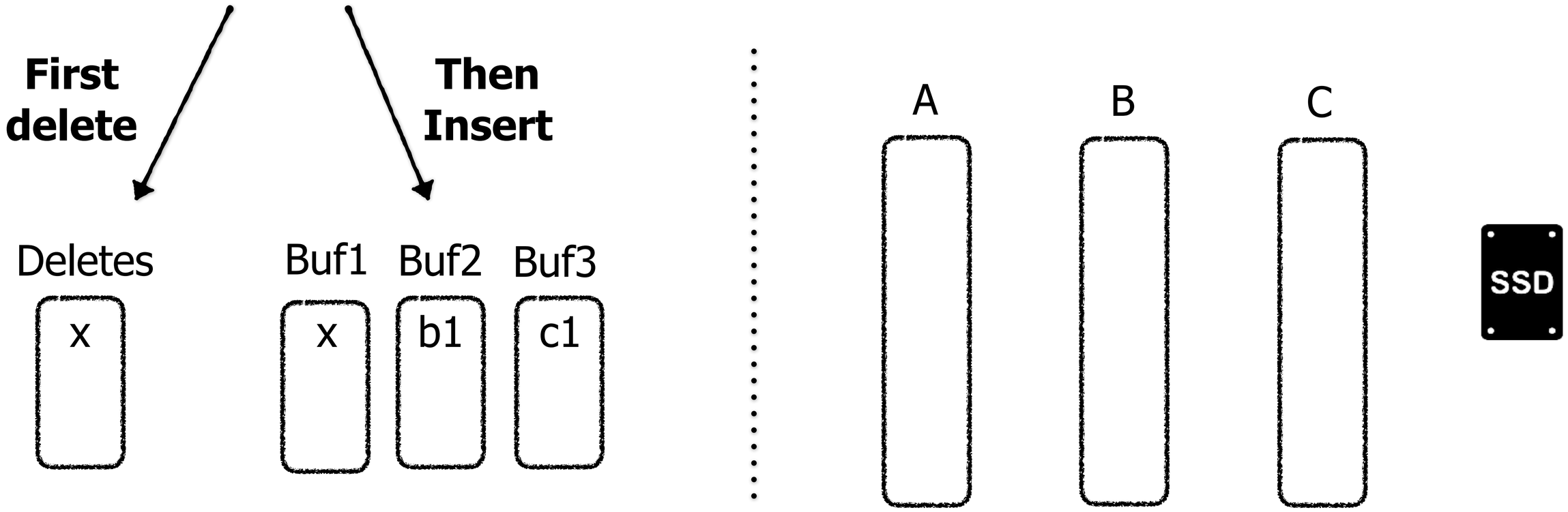
update table set **B="b1"**, **C="c1"** where **A="x"**

Can we combine our solutions for insertions & deletes to efficiently update?



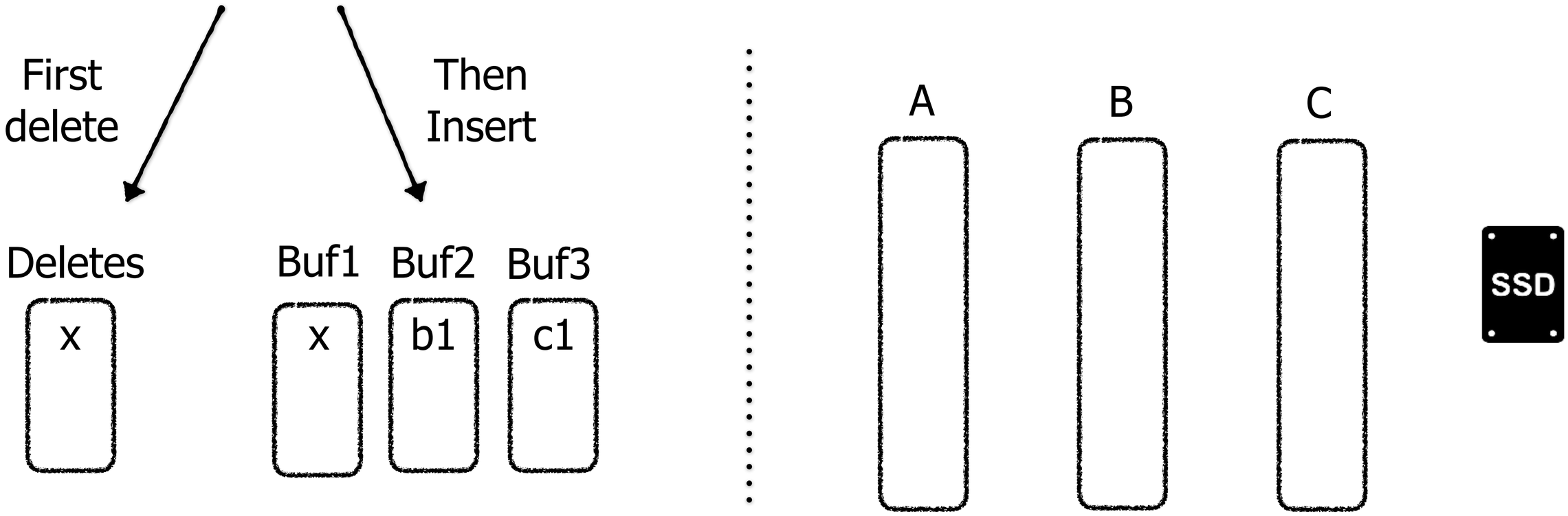
Handling Updates

update table set B="b1", C="c1" where A="x"



Handling Updates

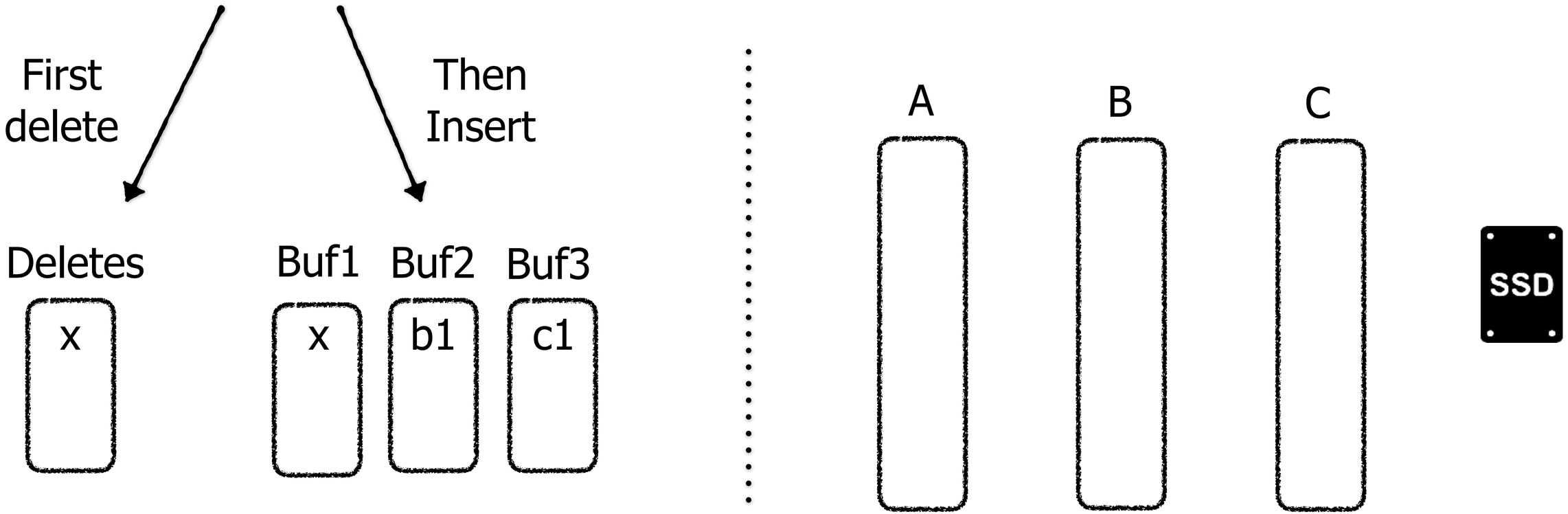
update table set B="b1", C="c1" where A="x"



Problem?

Handling Updates

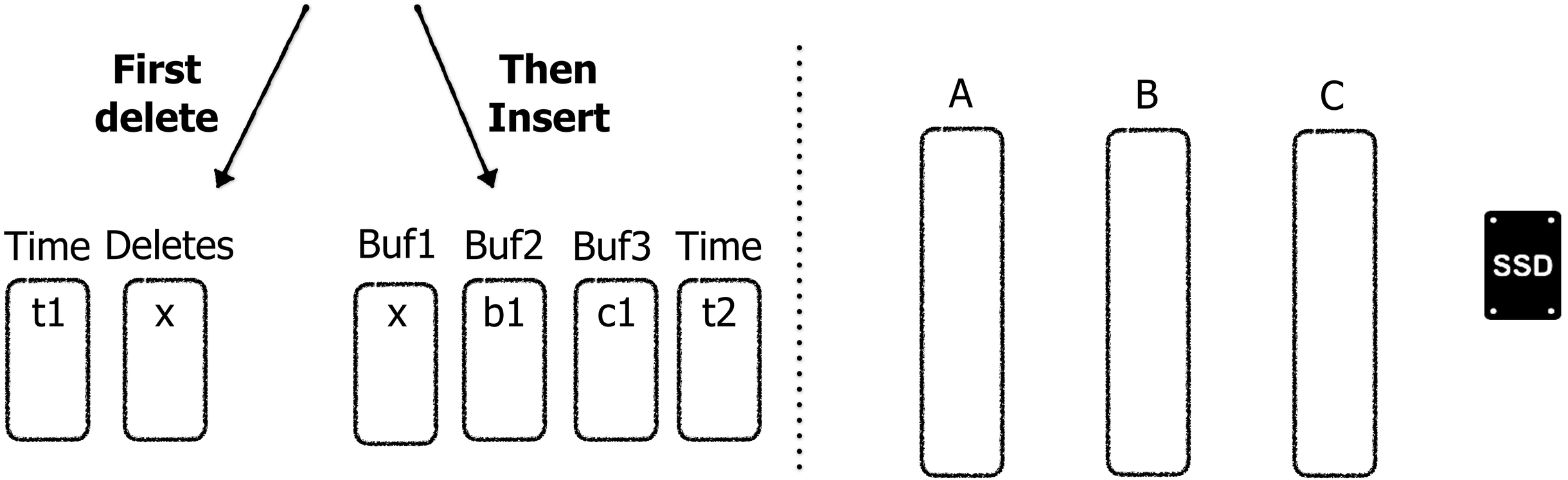
update table set B="b1", C="c1" where A="x"



Problem: Which came first?

Handling Updates

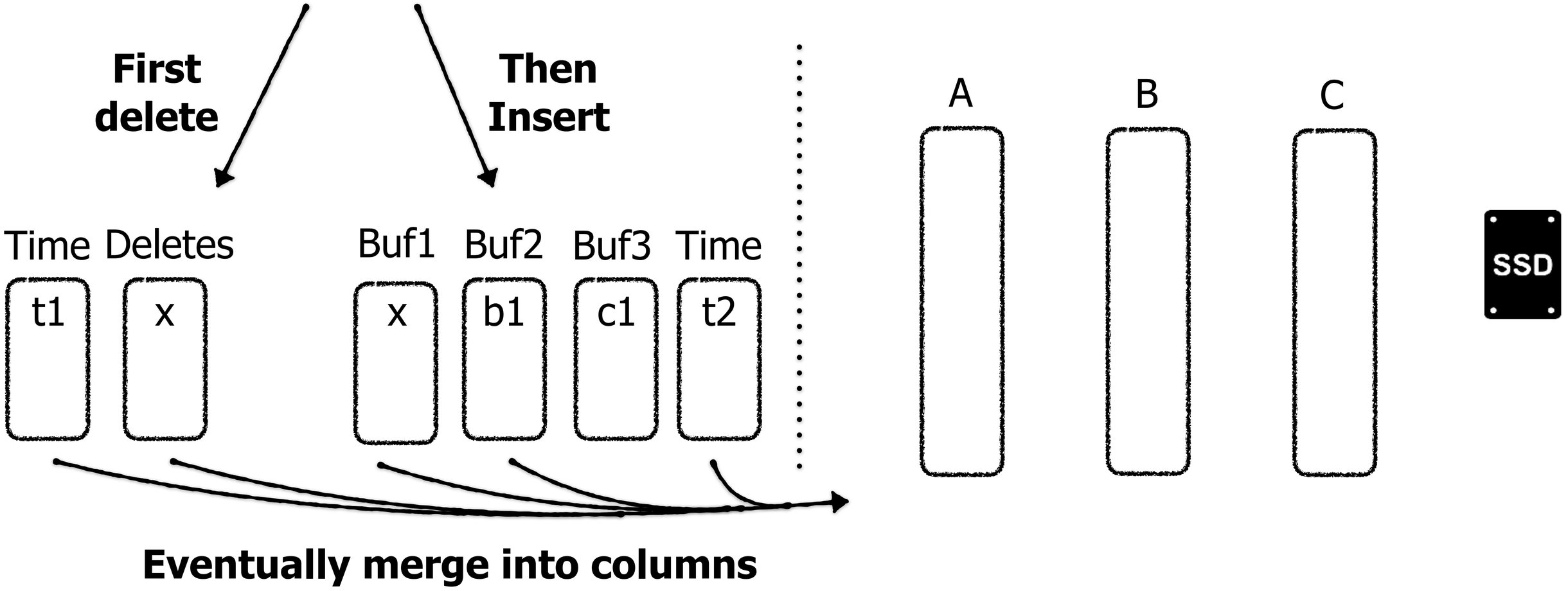
update table set B="b1", C="c1" where A="x"



Problem: Which came first?
Fix using timestamps

Handling Updates

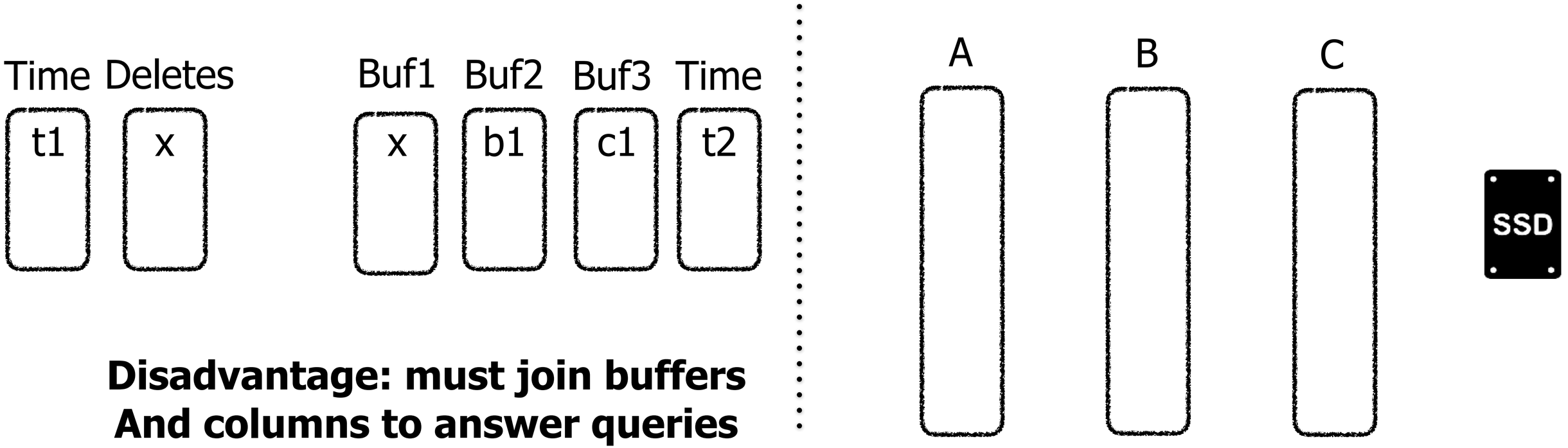
update table set B="b1", C="c1" where A="x"



Eventually merge into columns

Handling Updates

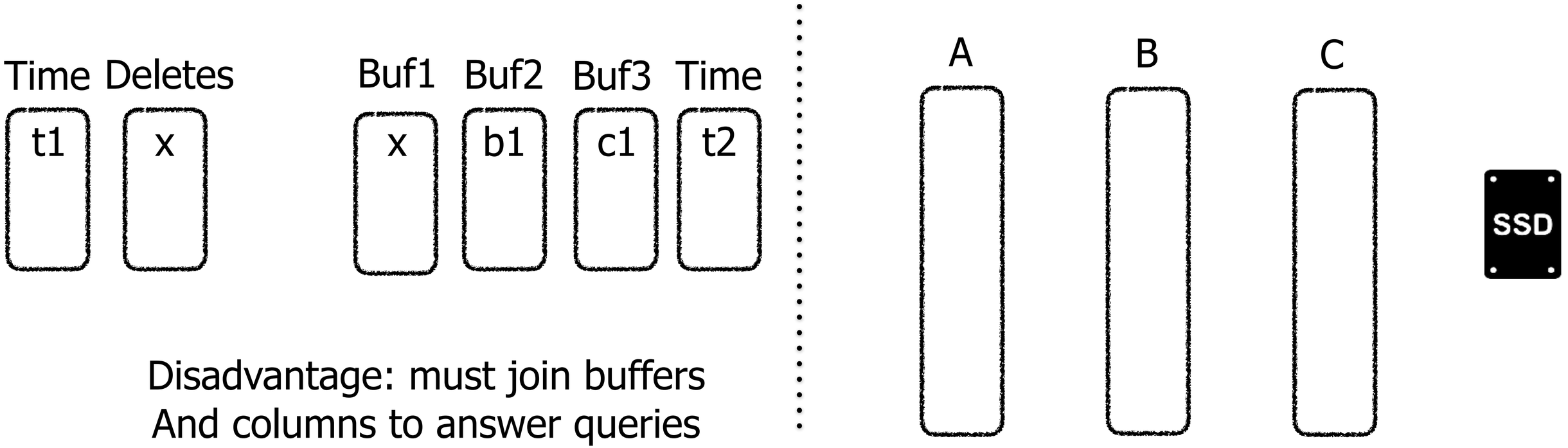
update table set B="b1", C="c1" where A="x"



Disadvantage: must join buffers
And columns to answer queries

Handling Updates

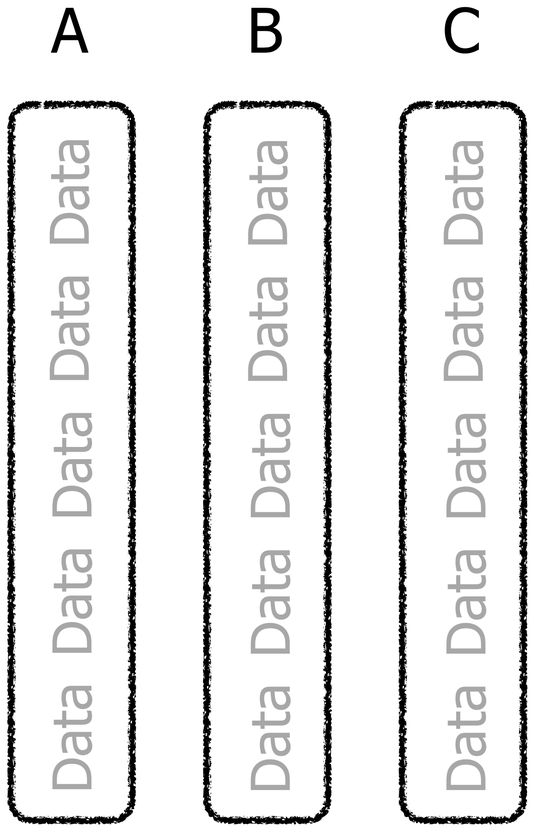
update table set B="b1", C="c1" where A="x"



Disadvantage: must join buffers
And columns to answer queries

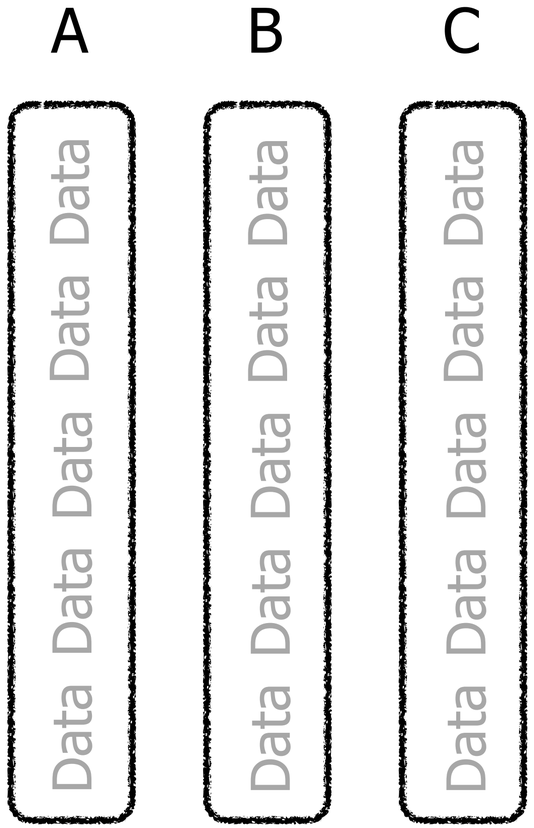
Hence, modifications are best done in batch and offline (e.g., overnight)

Columns are immutable and their values inside are mostly fixed-sized



Columns are immutable and their values inside are mostly fixed-sized

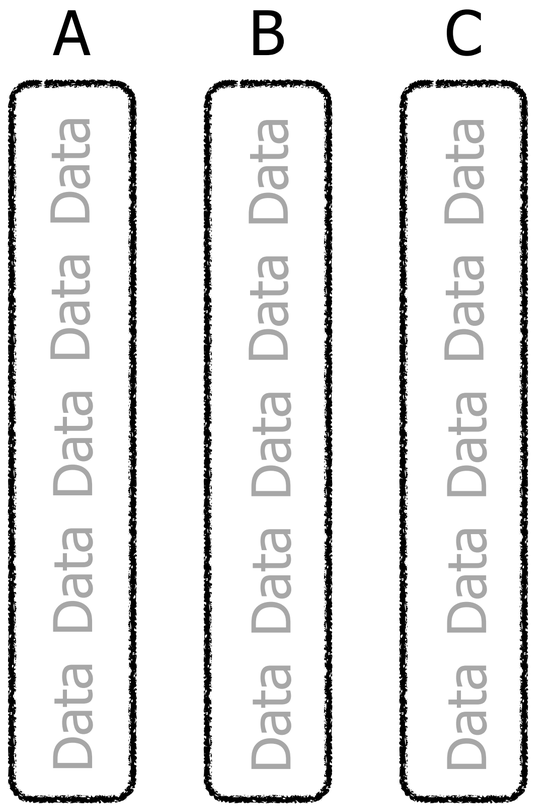
This allows for further optimizations



Tight processing loops with no pointer-chasing or function calls

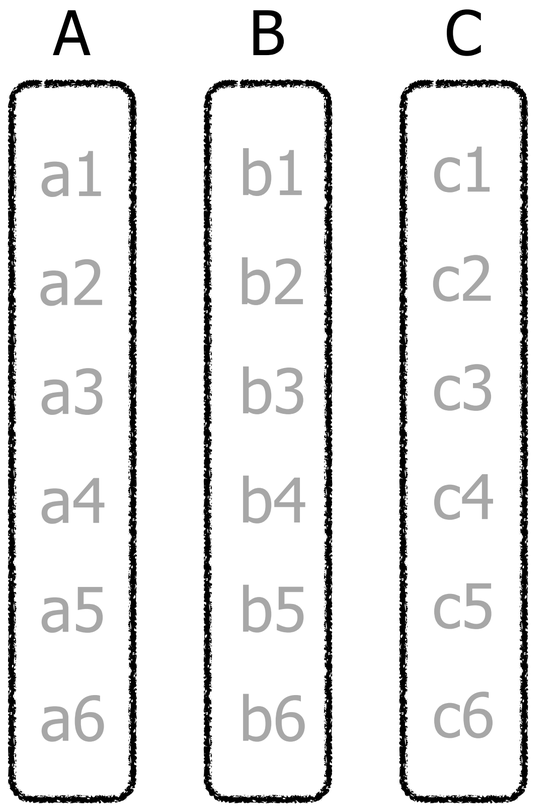
Select from table where A > v

```
for (i = 0; i < size; i++)
   if A[i] > v
      qualifying[j++] = i
```



SIMD - Same Instruction Multiple Data

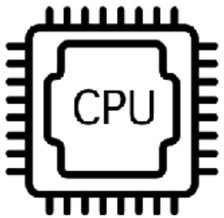
Apply one instruction in parallel to multiple values within one cache line (e.g., 128-256 bits at a time)



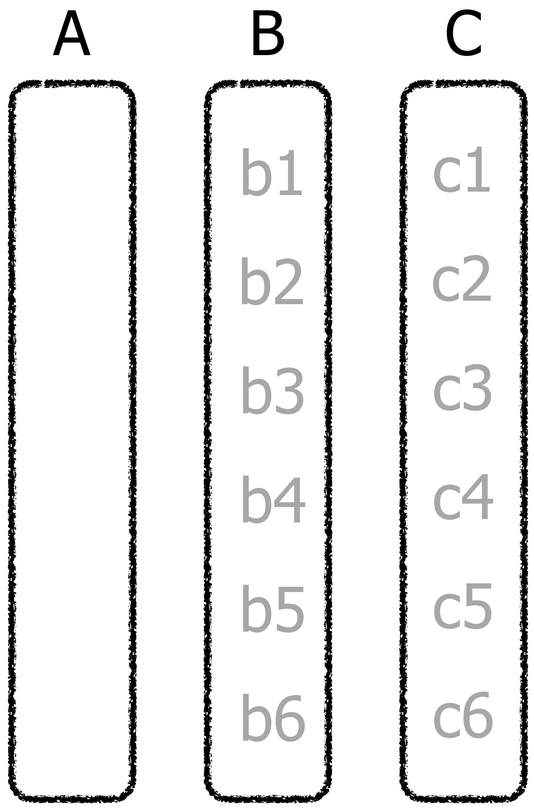
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to
multiple values within one cache line
(e.g., 128-256 bits at a time)

Select sum(A) from table where A > v



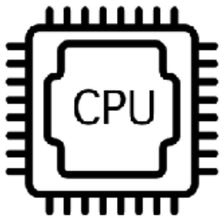
a1	a2	a3	a4	a5	a6	a7	a8
>	>	>	>	>	>	>	>
v	v	v	v	v	v	v	v



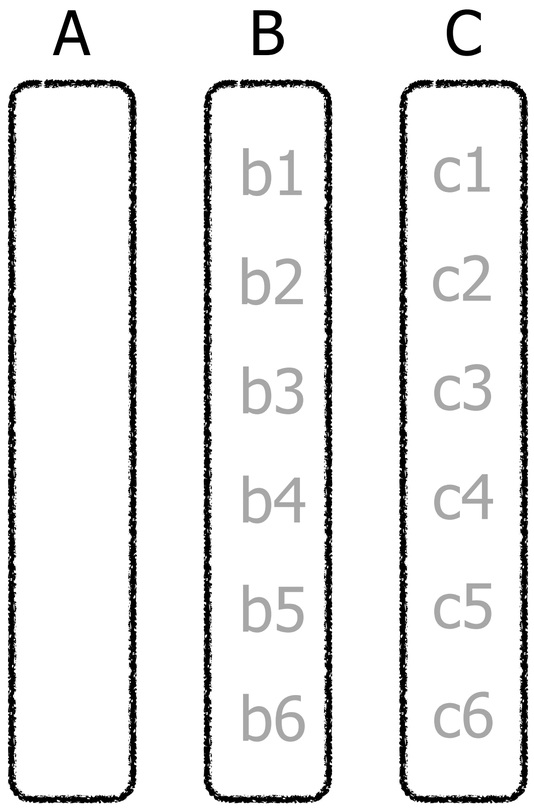
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to
multiple values within one cache line
(e.g., 128-256 bits at a time)

Select sum(A) from table where A > 5



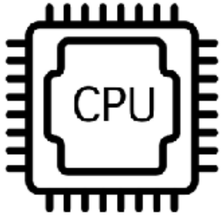
3	1	7	4	8	9	2	8
>	>	>	>	>	>	>	>
5	5	5	5	5	5	5	5



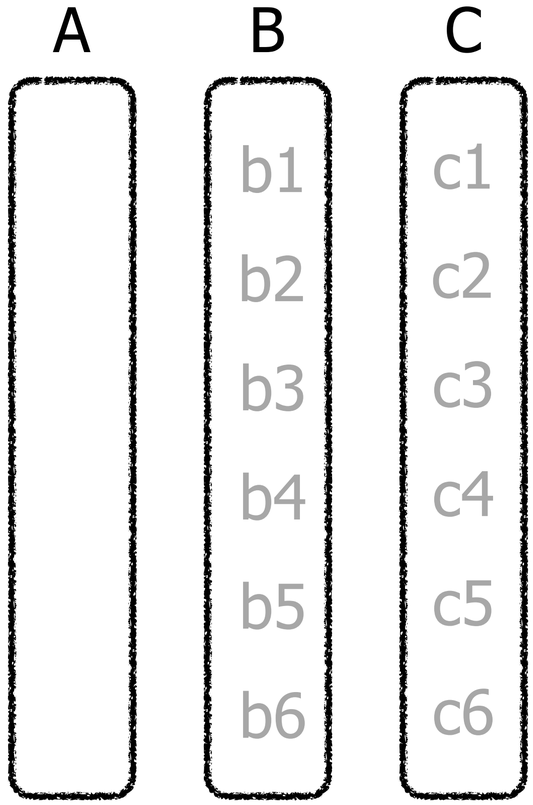
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to multiple values within one cache line (e.g., 128-256 bits at a time)

Select sum(A) from table where A > 5



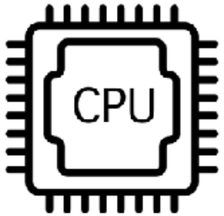
3	1	7	4	8	9	2	8
>	>	>	>	>	>	>	>
5	5	5	5	5	5	5	5
=	=	=	=	=	=	=	=
0	**0**	**1**	**0**	**1**	**1**	**0**	**1**



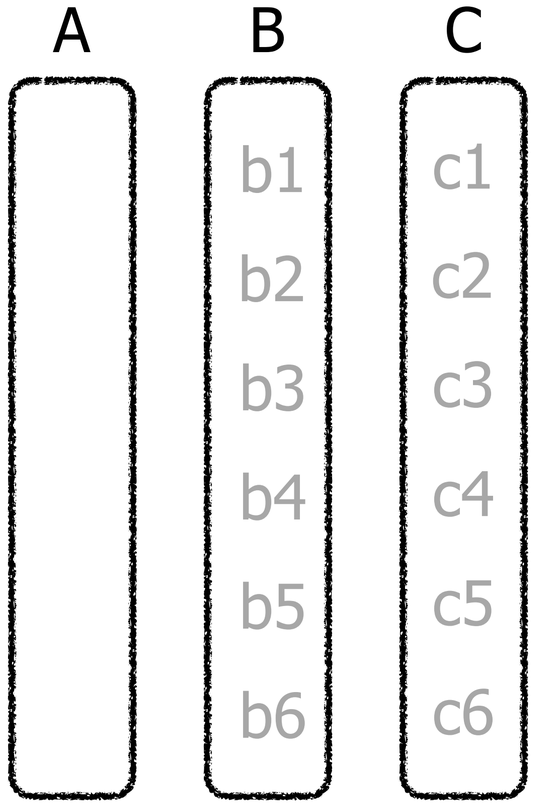
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to
multiple values within one cache line
(e.g., 128-256 bits at a time)

Select sum(A) from table where A > 5



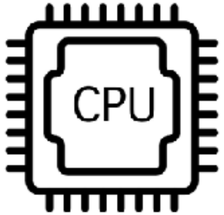
3	1	7	4	8	9	2	8
&	&	&	&	&	&	&	&
0	**0**	**1**	**0**	**1**	**1**	**0**	**1**



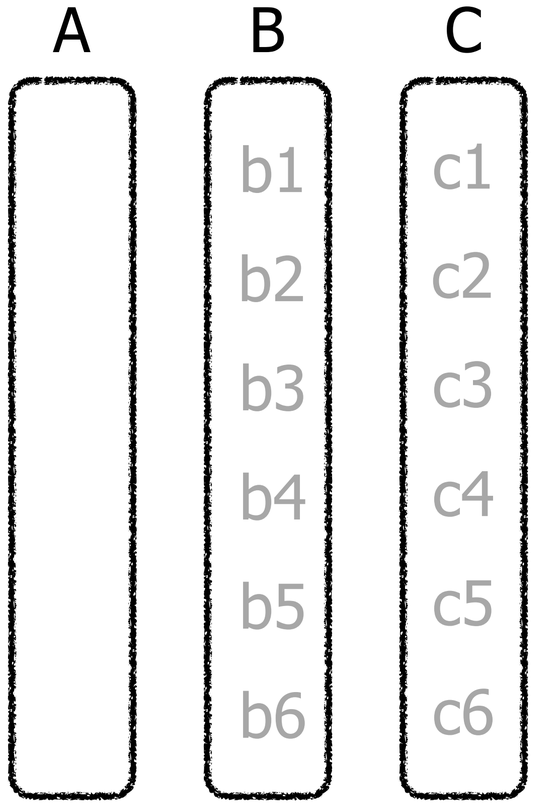
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to multiple values within one cache line (e.g., 128-256 bits at a time)

Select sum(A) from table where A > 5



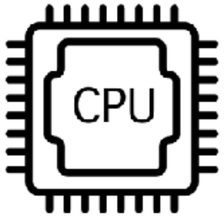
3	1	7	4	8	9	2	8
&	&	&	&	&	&	&	&
0	**0**	**1**	**0**	**1**	**1**	**0**	**1**
=	=	=	=	=	=	=	=
0	**0**	**7**	**0**	**8**	**9**	**0**	**8**



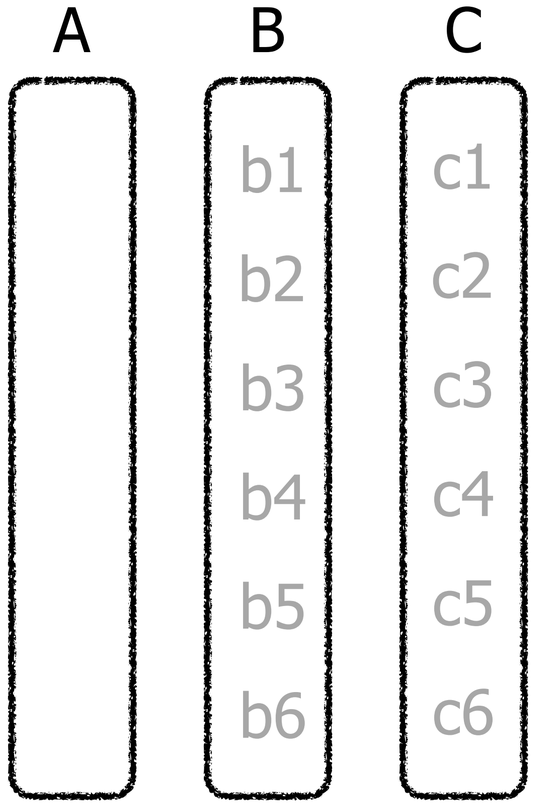
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to
multiple values within one cache line
(e.g., 128-256 bits at a time)

Select sum(A) from table where A > 5



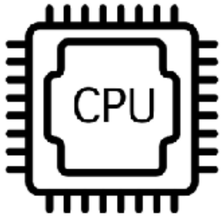
0	**0**	**7**	**0**	**8**	**9**	**0**	**8**
+	+	+	+	+	+	+	+
C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**
=	=	=	=	=	=	=	=
C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**



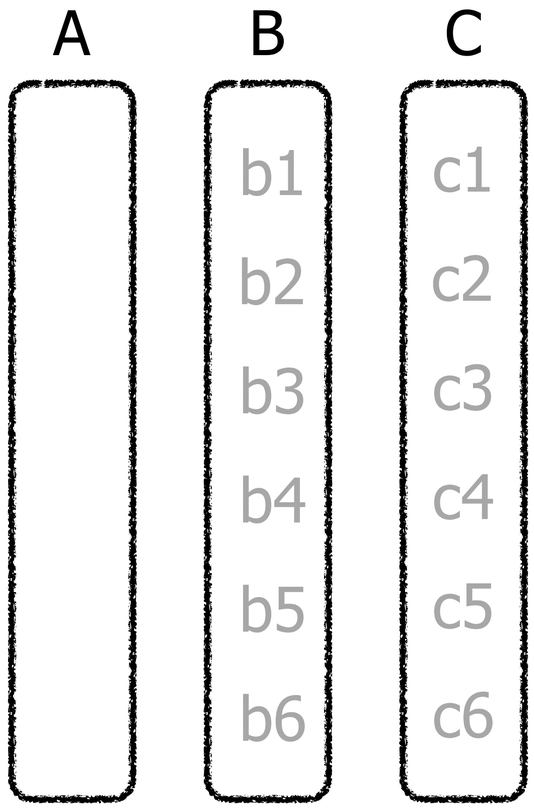
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to
multiple values within one cache line
(e.g., 128-256 bits at a time)

Select sum(A) from table where A > 5



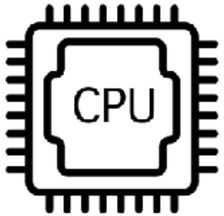
C1+C2+C3+C4+C5+C6+C7+C8= **Res**



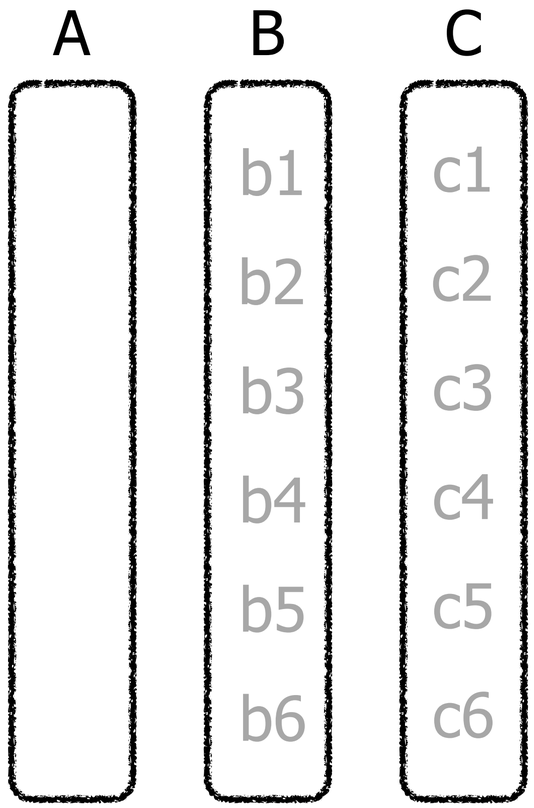
SIMD - Same Instruction Multiple Data

Apply one instruction in parallel to
multiple values within one cache line
(e.g., 128-256 bits at a time)

Select sum(A) from table where A > 5

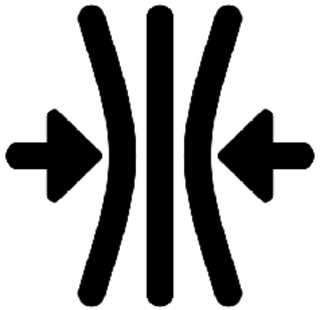


C1+C2+C3+C4+C5+C6+C7+C8= Res



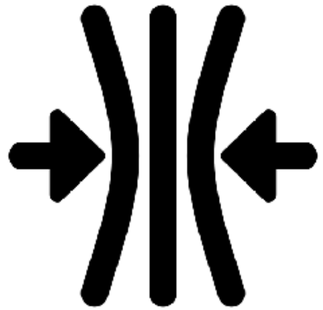
Works best with late materialization :)

Compression



Compression

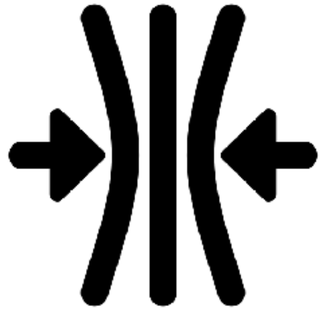
crucial for column-stores



Compression

crucial for column-stores

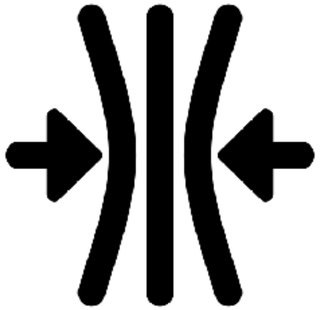
The reason is not only to save space but to improve performance. How?



Compression

crucial for column-stores

The reason is not only to save space but to improve performance. How?



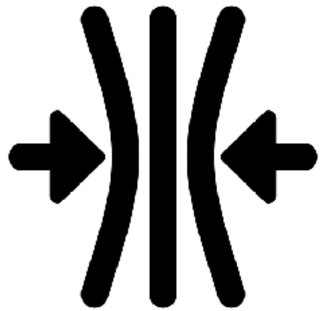
CPU cost of compression and decompression **<** **cost savings of moving data across the memory hierarchy**

Compression

crucial for column-stores

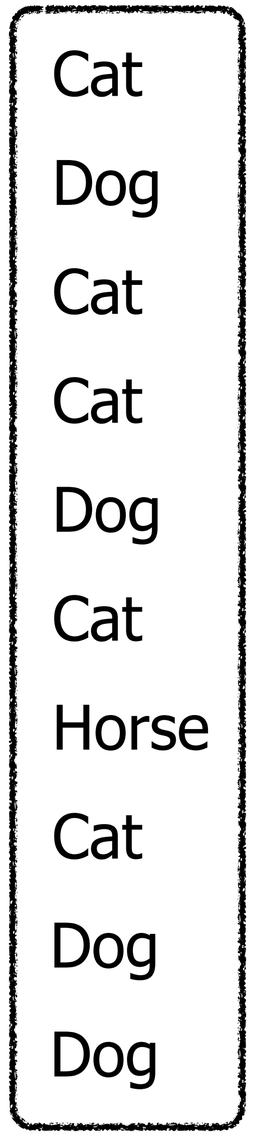
The reason is not only to save space but to improve performance. How?

Important for compressed values to be fixed-size to support positional ID lookups



(1) Bit-Vector Encoding

Employ one bit string for each possible value indicating if the entry has the given value



(1) Bit-Vector Encoding

Employ one bit string for each possible value indicating if the entry has the given value

Horse	Cat	Dog	Pet
0	1	0	Cat
0	0	1	Dog
0	1	0	Cat
0	1	0	Cat
0	0	1	Dog
0	1	0	Cat
1	0	0	Horse
0	1	0	Cat
0	0	1	Dog
0	0	1	Dog

(1) Bit-Vector Encoding

Employ one bit string for each possible value indicating if the entry has the given value

e.g., select * from table where specie = “Cat”

Pros and cons?

Horse	**Cat**	Dog	Pet
0	**1**	0	Cat
0	**0**	1	Dog
0	**1**	0	Cat
0	**1**	0	Cat
0	**0**	1	Dog
0	**1**	0	Cat
1	**0**	0	Horse
0	**1**	0	Cat
0	**0**	1	Dog
0	**0**	1	Dog

(1) Bit-Vector Encoding

Employ one bit string for each possible value indicating if the entry has the given value

e.g., select * from table where specie = “Cat”

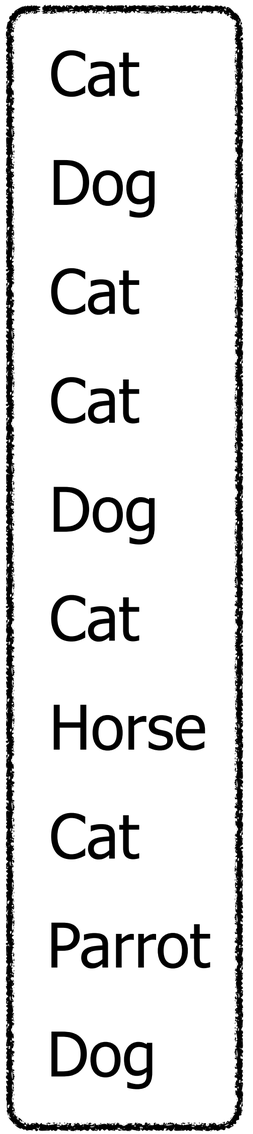
Pro: Fast to read & compare 1 bit per entry.

Con: Only applicable if there are very few values.

Horse	Cat	Dog	Pet
0	1	0	Cat
0	0	1	Dog
0	1	0	Cat
0	1	0	Cat
0	0	1	Dog
0	1	0	Cat
1	0	0	Horse
0	1	0	Cat
0	0	1	Dog
0	0	1	Dog

(2) Dictionary Encoding

Employ a dictionary with smaller strings to represent larger ones



(2) Dictionary Encoding

Employ a dictionary with smaller strings to represent larger ones

Dictionary

00	Cat
01	Dog
10	Horse
11	Parrot

Pet

Cat
Dog
Cat
Cat
Dog
Cat
Horse
Cat
Parrot
Dog

(2) Dictionary Encoding

Employ a dictionary with smaller strings to represent larger ones

Dictionary	
00	Cat
01	Dog
10	Horse
11	Parrot

Compressed
00
01
00
00
01
00
10
00
11
10

Pet
Cat
Dog
Cat
Cat
Dog
Cat
Horse
Cat
Parrot
Dog

(2) Dictionary Encoding

Employ a dictionary with smaller strings to represent larger ones

Dictionary	
00	Cat
01	Dog
10	Horse
11	Parrot

Compressed
00
01
00
00
01
00
10
00
11
10

Pet
Cat
Dog
Cat
Cat
Dog
Cat
Horse
Cat
Parrot
Dog

Compare to bit-vector encoding from before

(2) Dictionary Encoding

Employ a dictionary with smaller strings to represent larger ones

Dictionary	
00	Cat
01	Dog
10	Horse
11	Parrot

Compressed
00
01
00
00
01
00
10
00
11
10

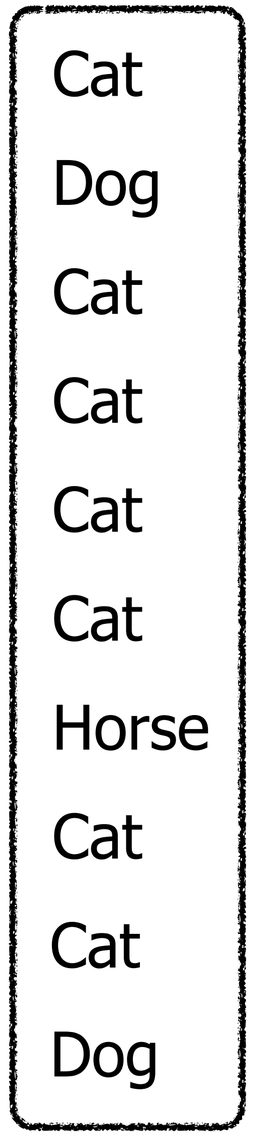
Pet
Cat
Dog
Cat
Cat
Dog
Cat
Horse
Cat
Parrot
Dog

Pro: applicable across larger value spaces

Con: slower as we need to read more bits

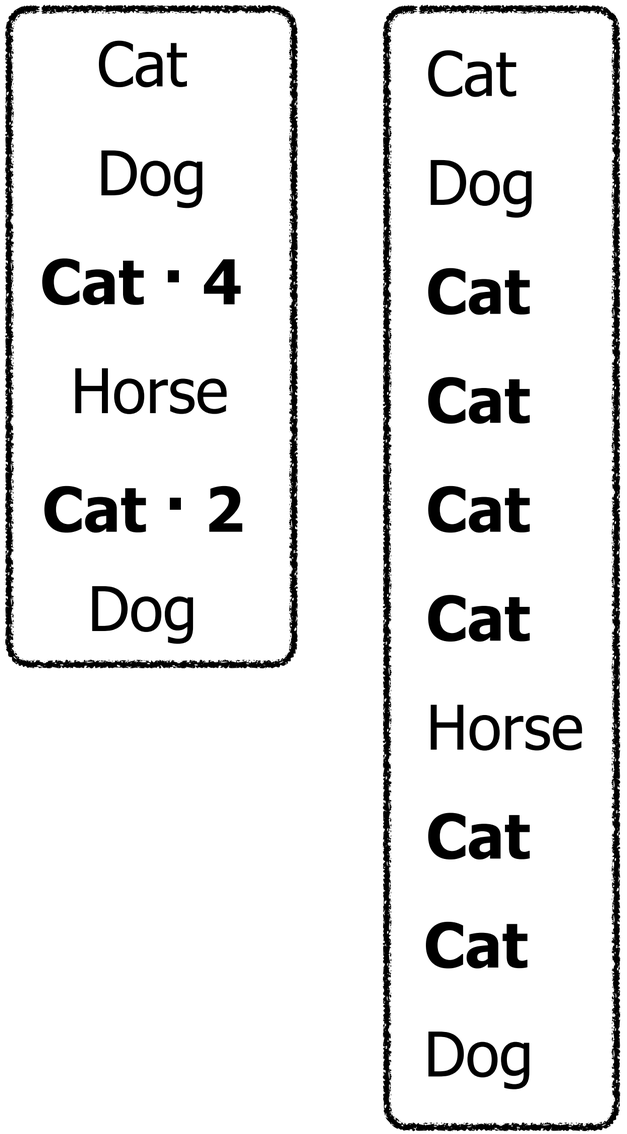
(3) Run-length encoding

Represent repeating values using one entry



(3) Run-length encoding

Represent repeating values using one entry



(3) Run-length encoding

Represent repeating values using one entry

Compatible with dictionary encoding

Pros and Cons?

Compressed
Cat
Dog
Cat · 4
Horse
Cat · 2
Dog

Pet
Cat
Dog
Cat
Cat
Cat
Cat
Horse
Cat
Cat
Dog

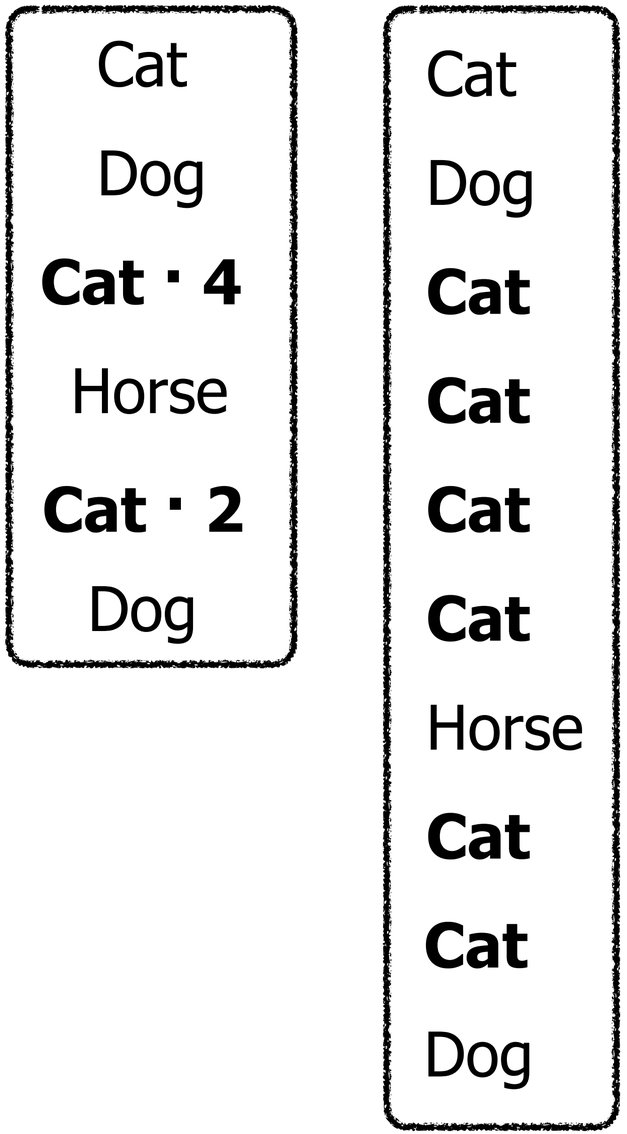
(3) Run-length encoding

Represent repeating values using one entry

Compatible with dictionary encoding

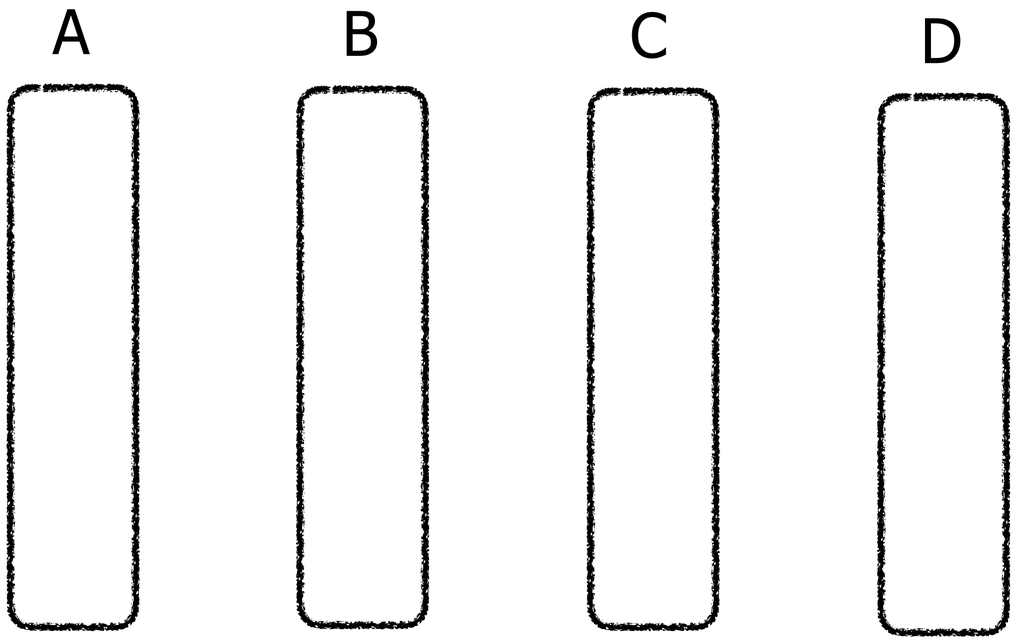
Pros: compatible with dictionary encoding & can further improve compression

Cons: must scan column to get entry at a given offset



Indexing in Column Stores

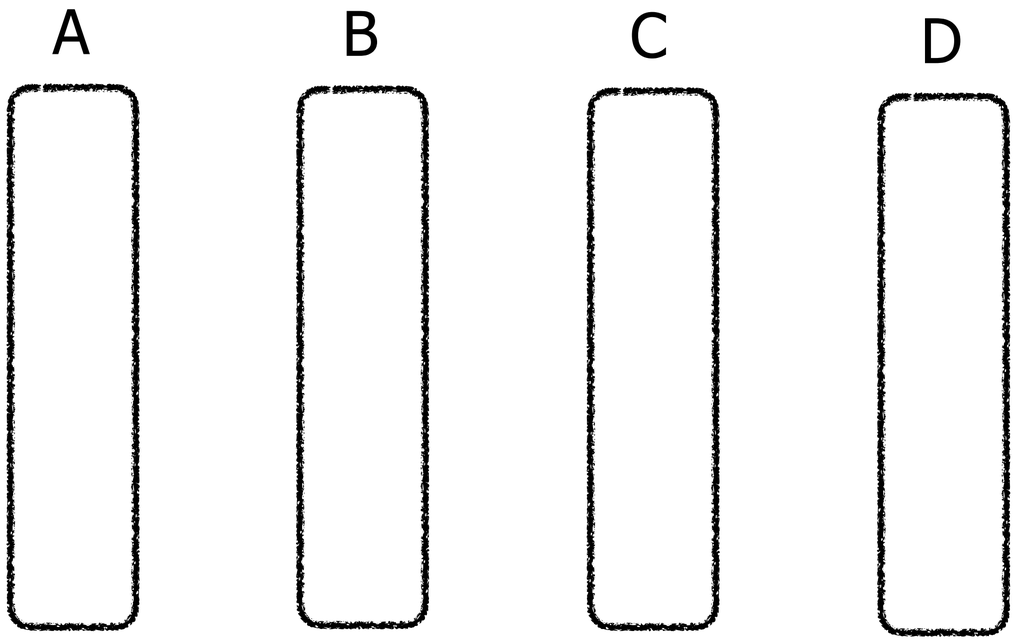
Suppose we have two selective queries over different columns in the same table



Suppose we have two selective queries over different columns in the same table

Select avg(B)
where A > x and A < y

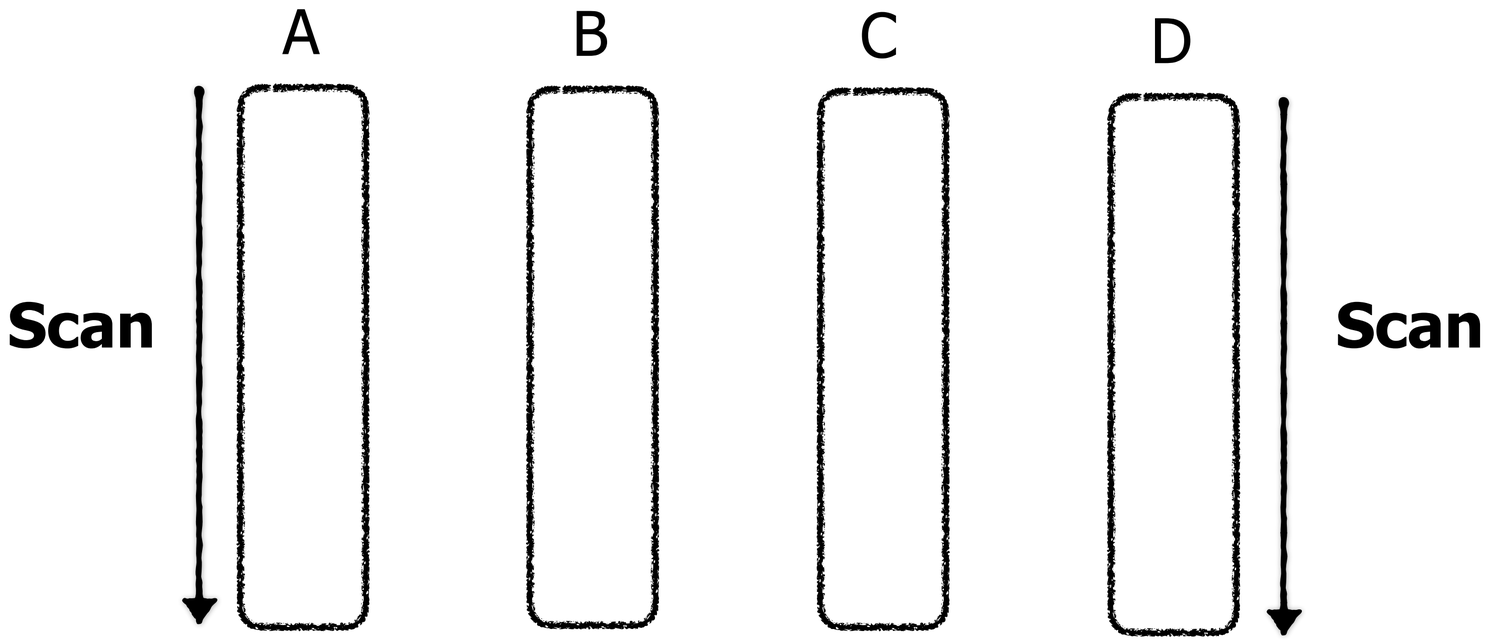
Select avg(C)
where D > x and D < y



Suppose we have two selective queries over different columns in the same table

Select avg(B)
where A > x and A < y

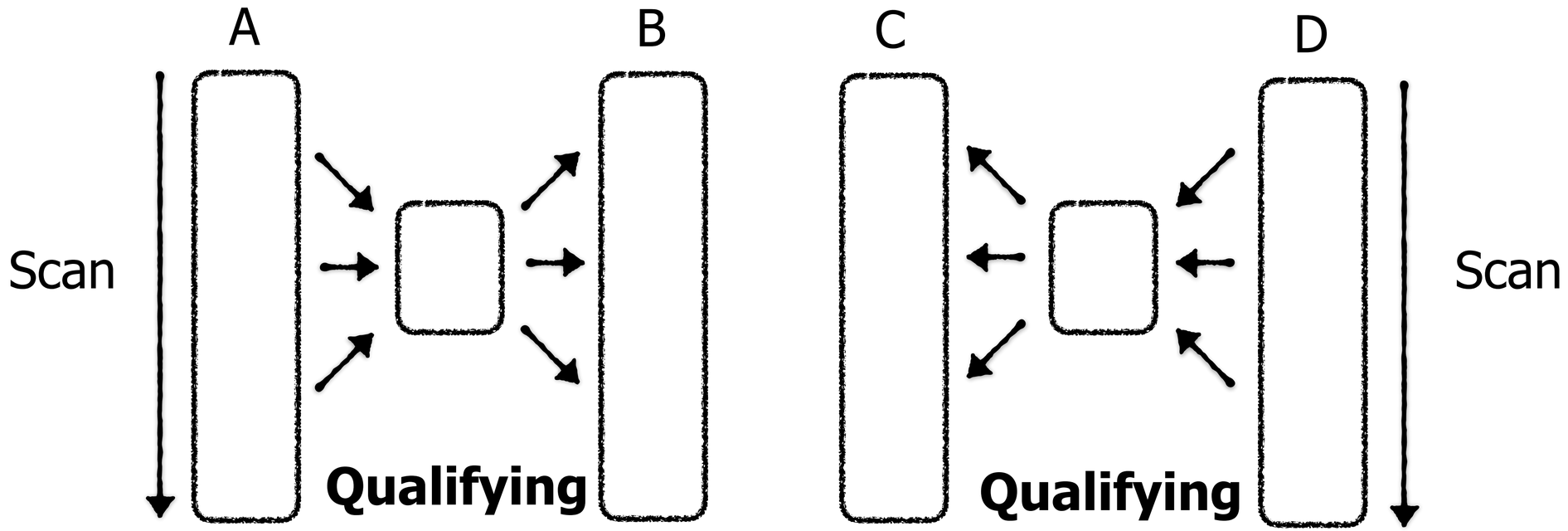
Select avg(C)
where D > x and D < y



Suppose we have two selective queries over different columns in the same table

Select avg(B)
where A > x and A < y

Select avg(C)
where D > x and D < y

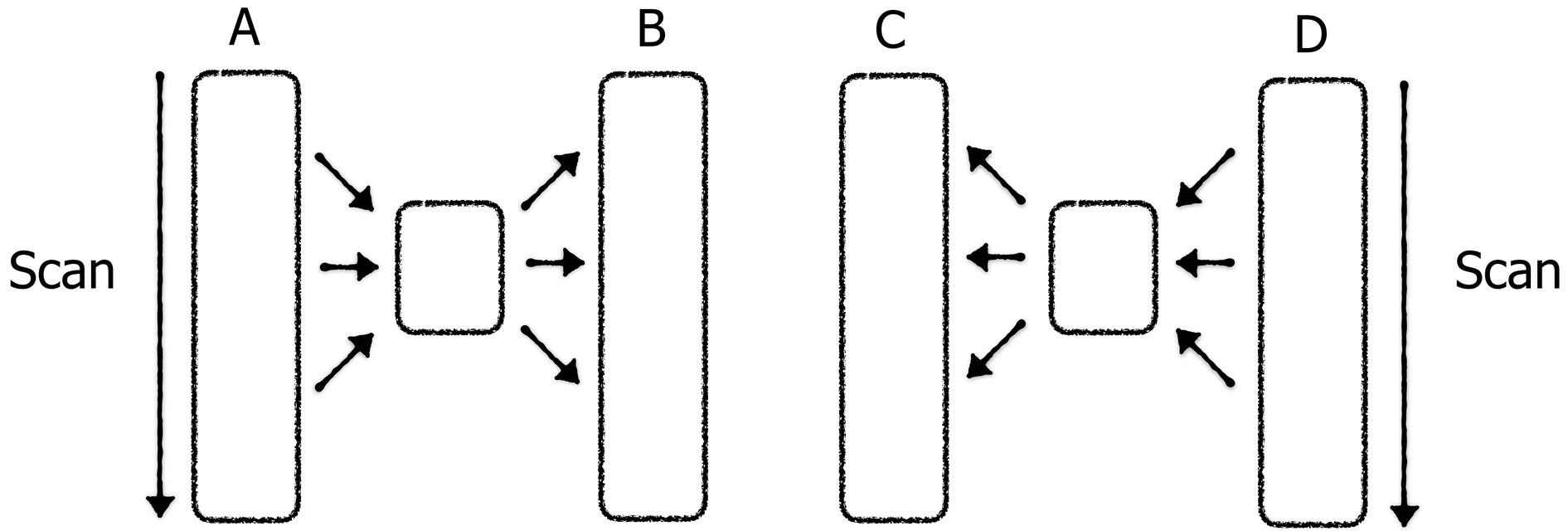


We can answer both queries with full scan and late materialization

Suppose we have two selective queries over different columns in the same table

Select avg(B)
where A > x and A < y

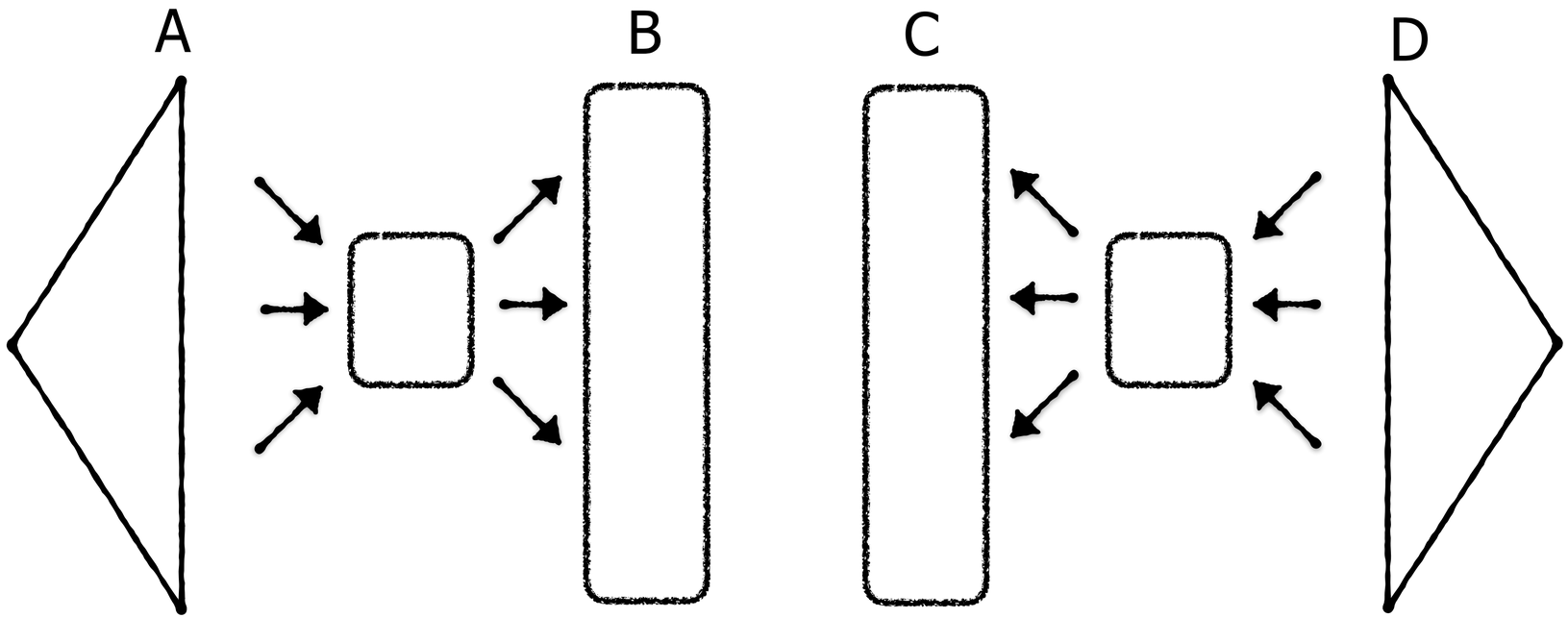
Select avg(C)
where D > x and D < y



But can we avoid the full scans?

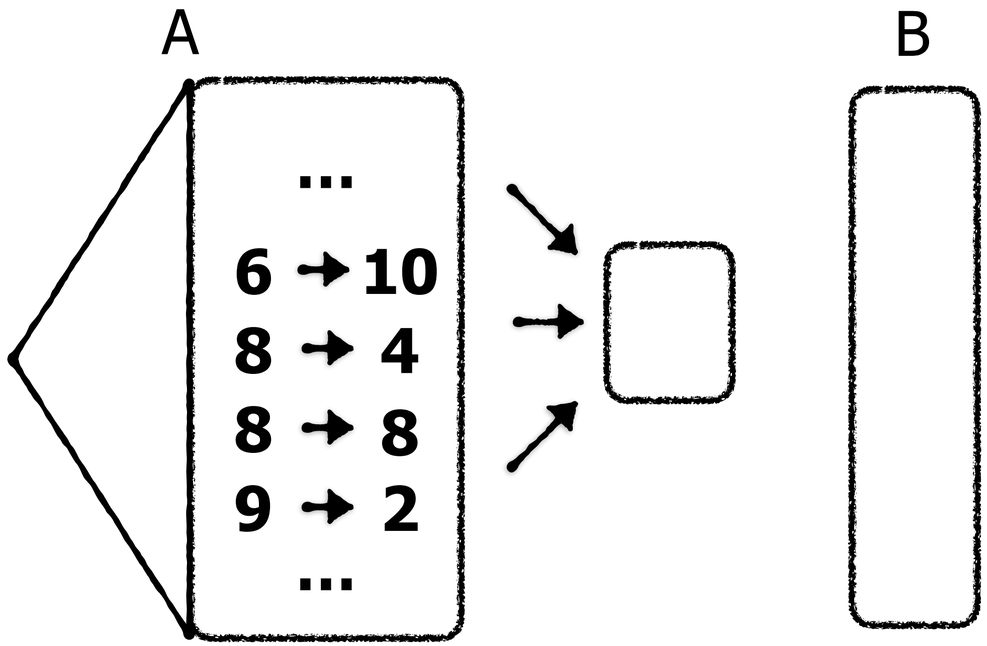
Select avg(B)
where A > x and A < y

Select avg(C)
where D > x and D < y



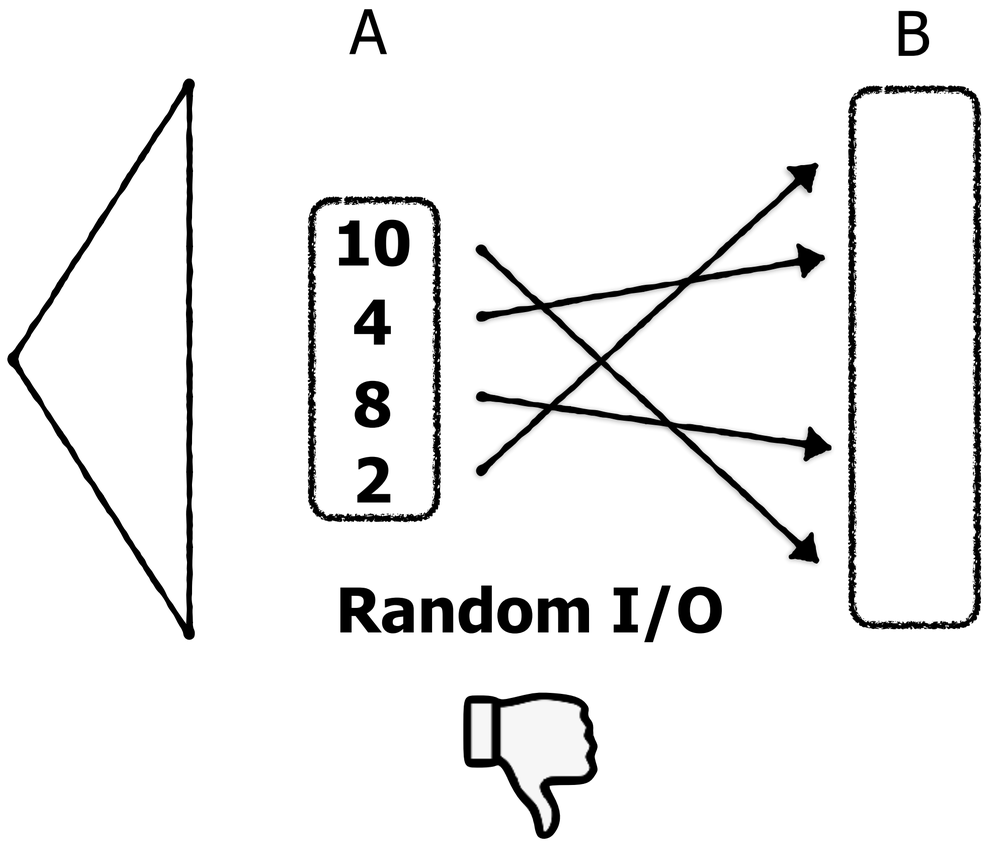
We can employ indexes mapping from key to positional ID

Select avg(B)
where **A > 5** and **A < 10**

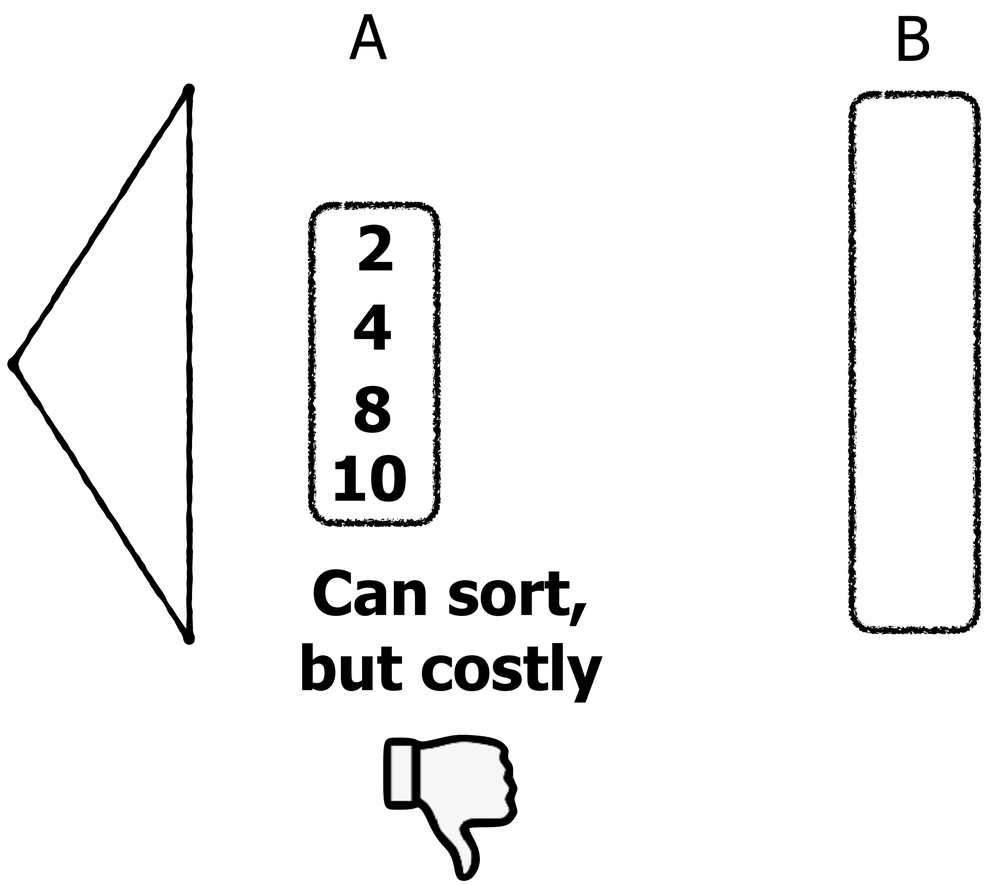


We can employ indexes mapping from key to positional ID

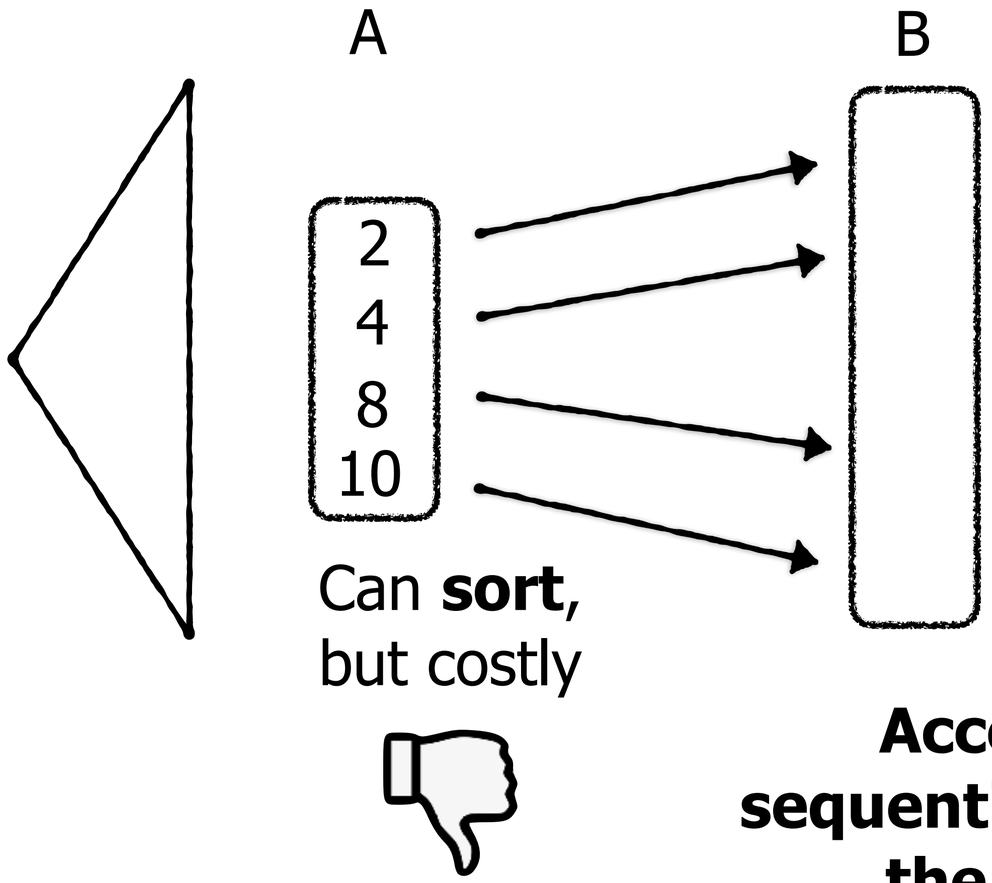
Select avg(B)
where A > 5 and A < 10



Select avg(B)
where A > 5 and A < 10

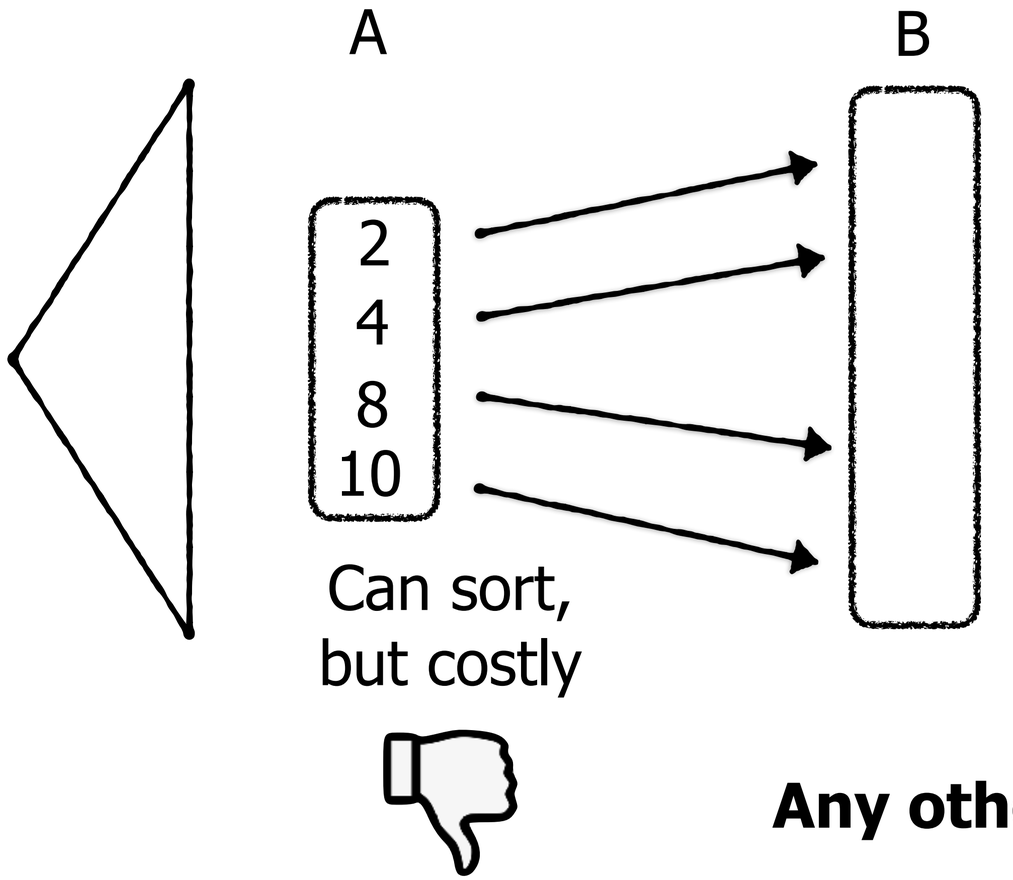


Select avg(B)
where A > 5 and A < 10



Access to B becomes "skip-sequential" rather than random, but the sorting can be an issue

Select avg(B)
where A > 5 and A < 10

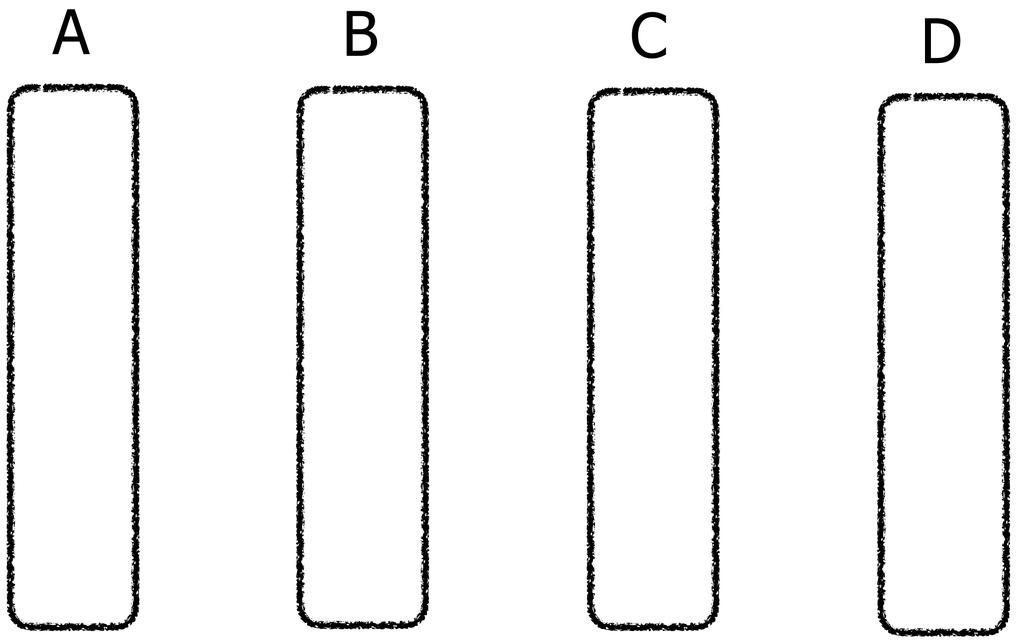


Any other solution?

Column Projections: sort subset of columns by one column

Select avg(B)
where A > x and A < y

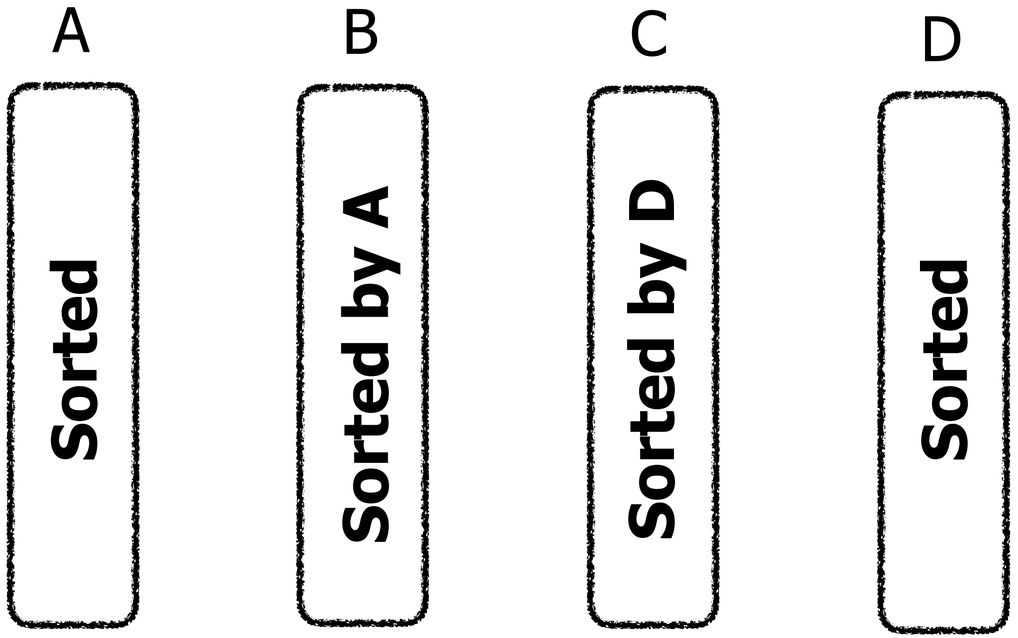
Select avg(C)
where D > x and D < y



Column Projections: sort subset of columns by one column

Select avg(B)
where A > x and A < y

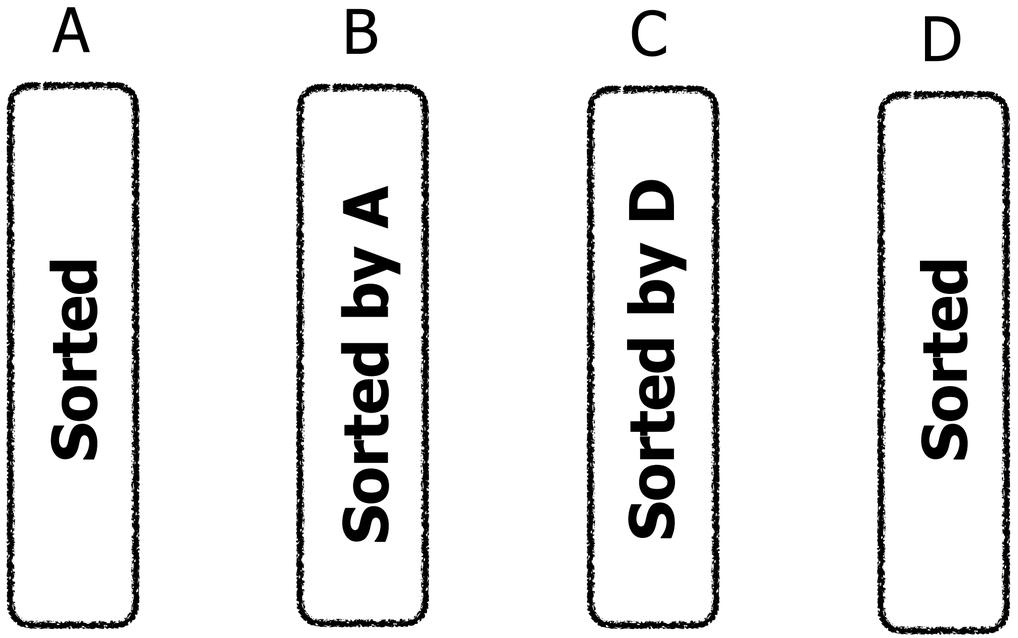
Select avg(C)
where D > x and D < y



Column Projections: sort subset of columns by one column

Select avg(B)
where A > x and A < y

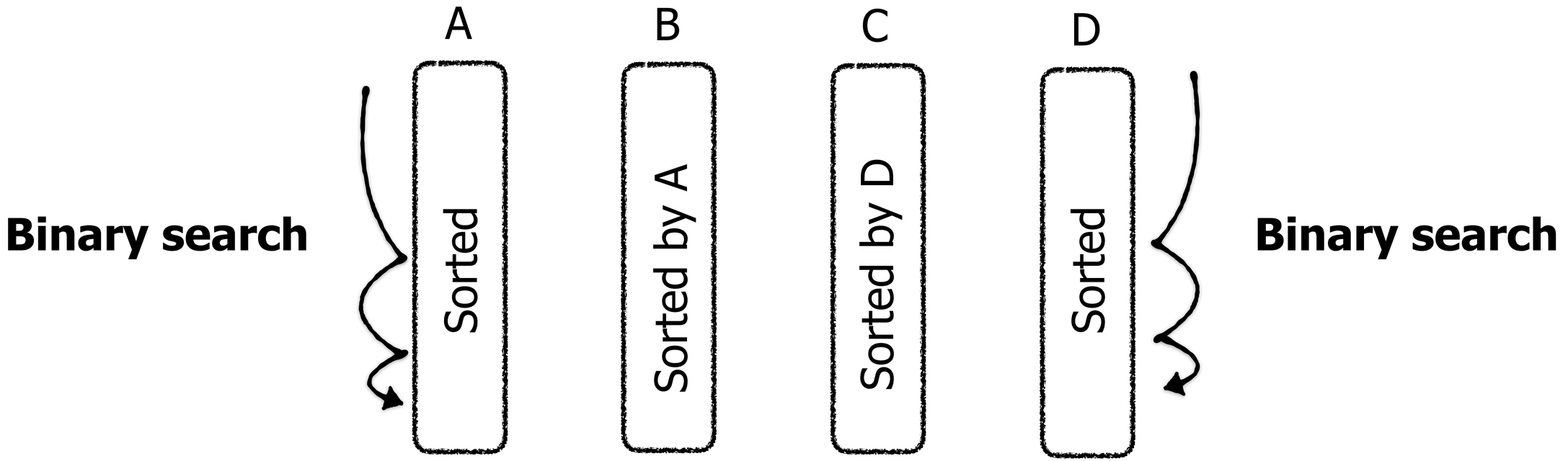
Select avg(C)
where D > x and D < y

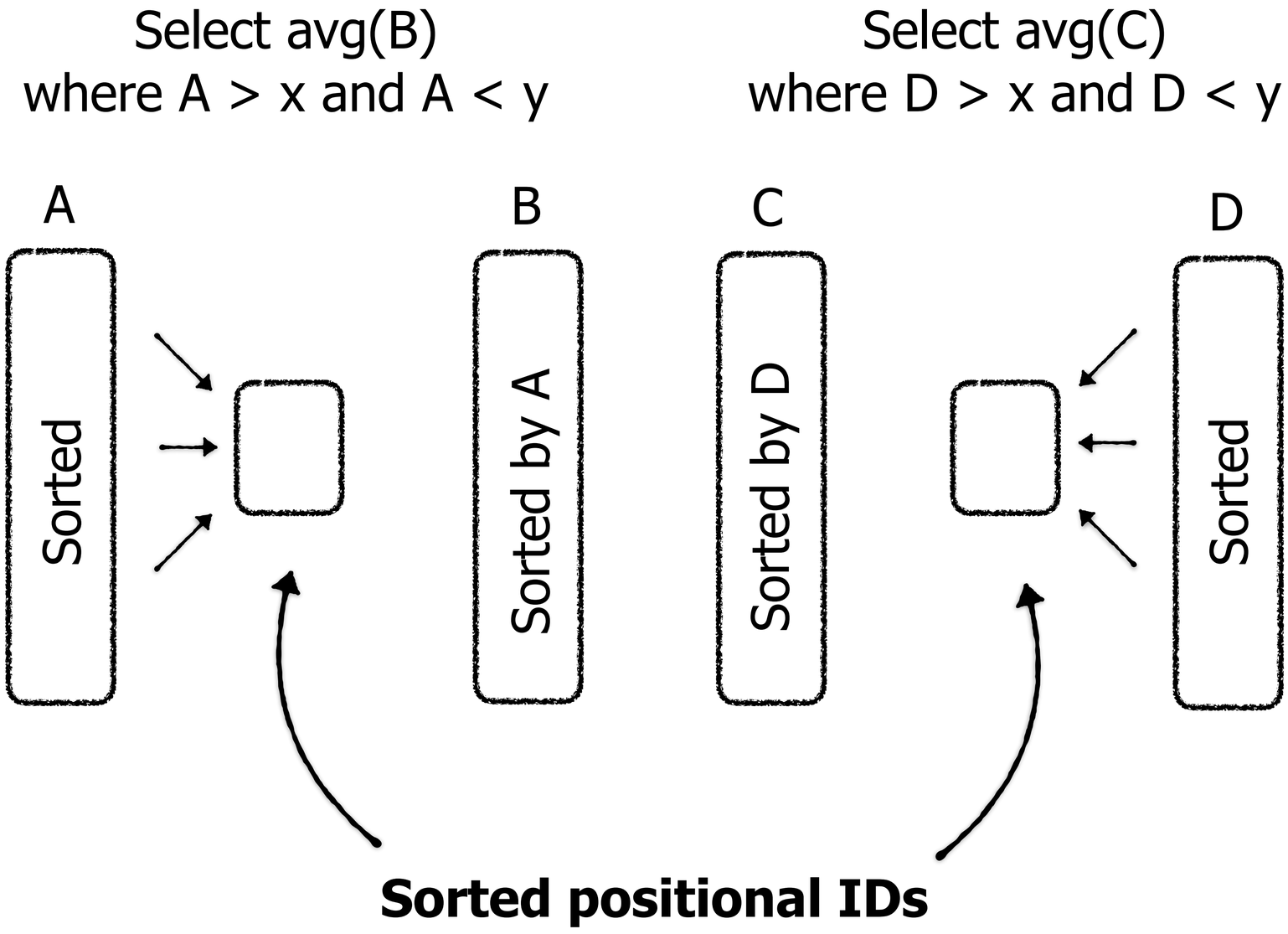


Like having multiple clustered indexes on subset of columns

Select avg(B)
where A > x and A < y

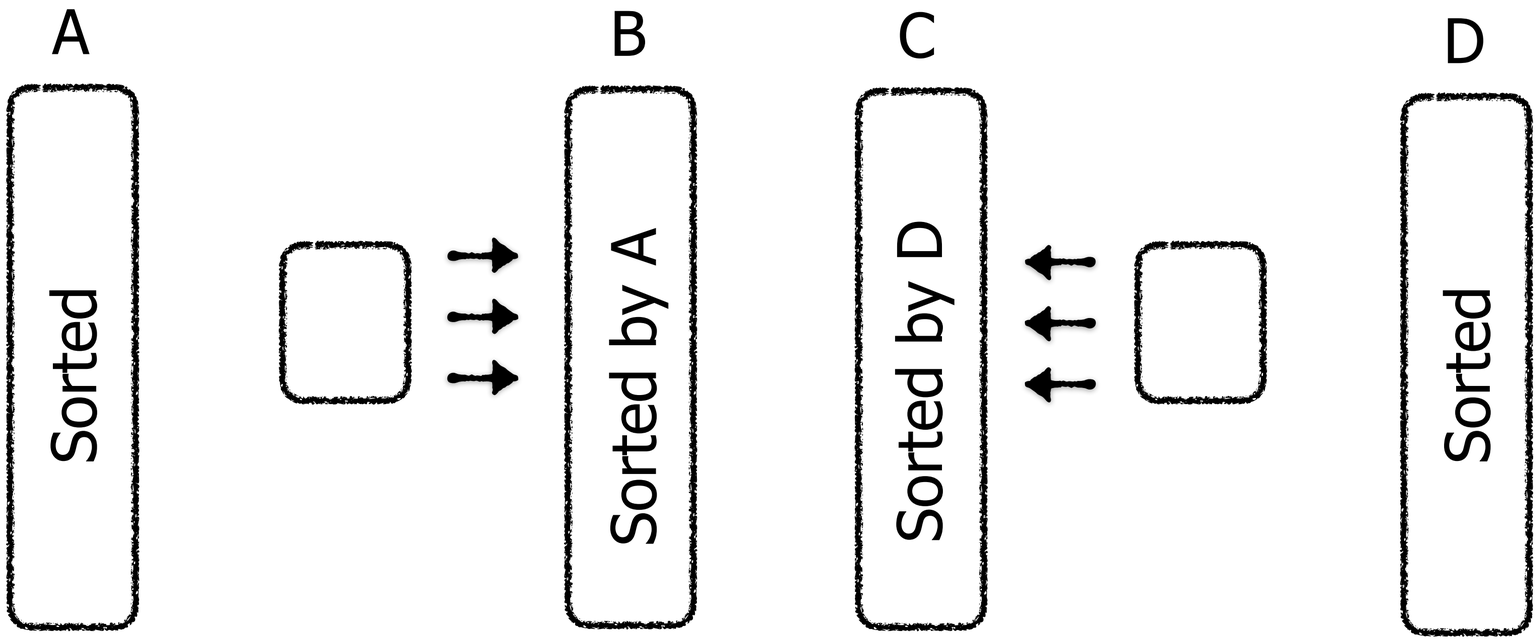
Select avg(C)
where D > x and D < y




Select avg(B)
where A > x and A < y
Select avg(C)
where D > x and D < y
A
B
C
D
Sorted
Sorted by A
Sorted by D
Sorted
Sorted positional IDs

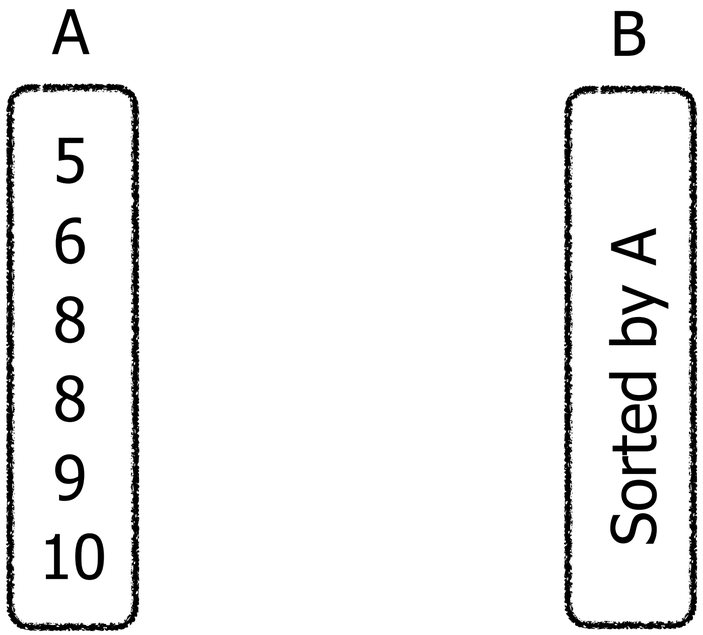
Select avg(B)
where A > x and A < y

Select avg(C)
where D > x and D < y



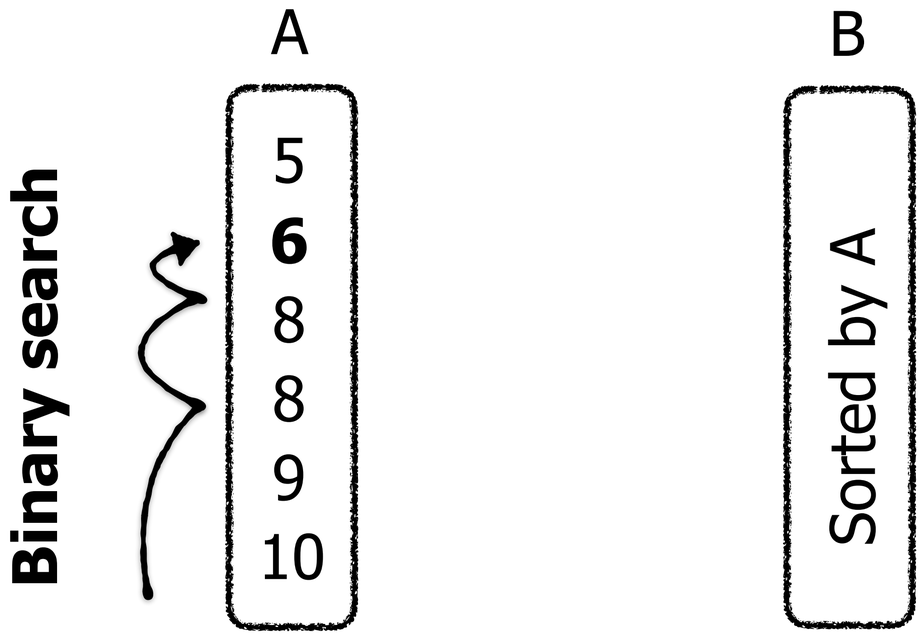
Enables sequential access over B and C

Select avg(B)
where A > 5 and A < 10



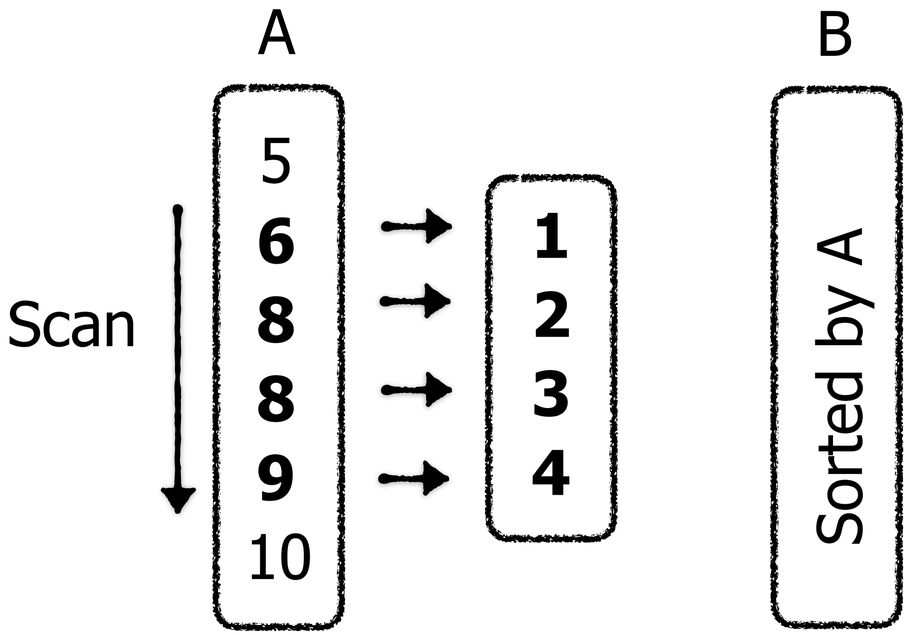
For example

Select avg(B)
where A > 5 and A < 10



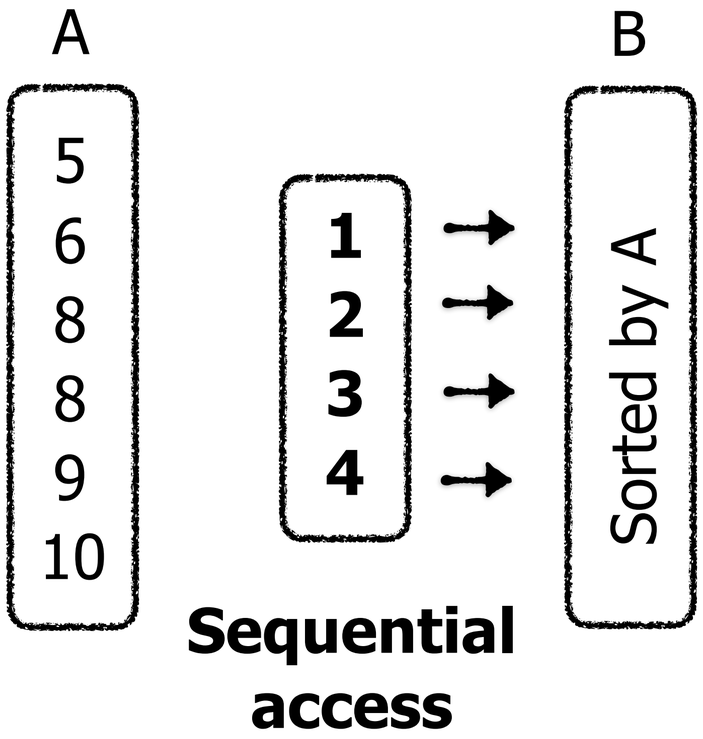
For example

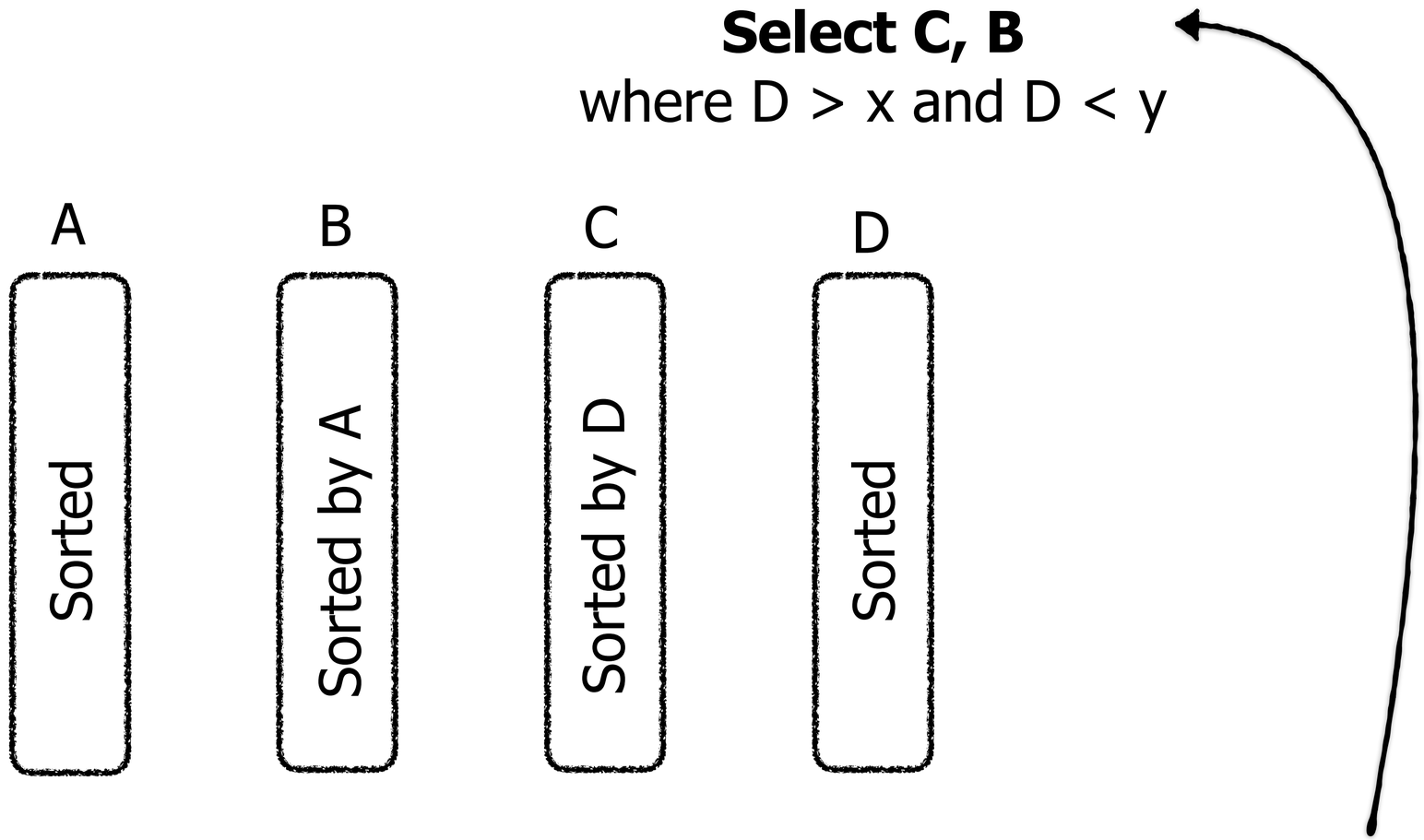
Select avg(B)
where A > 5 and A < 10



For example

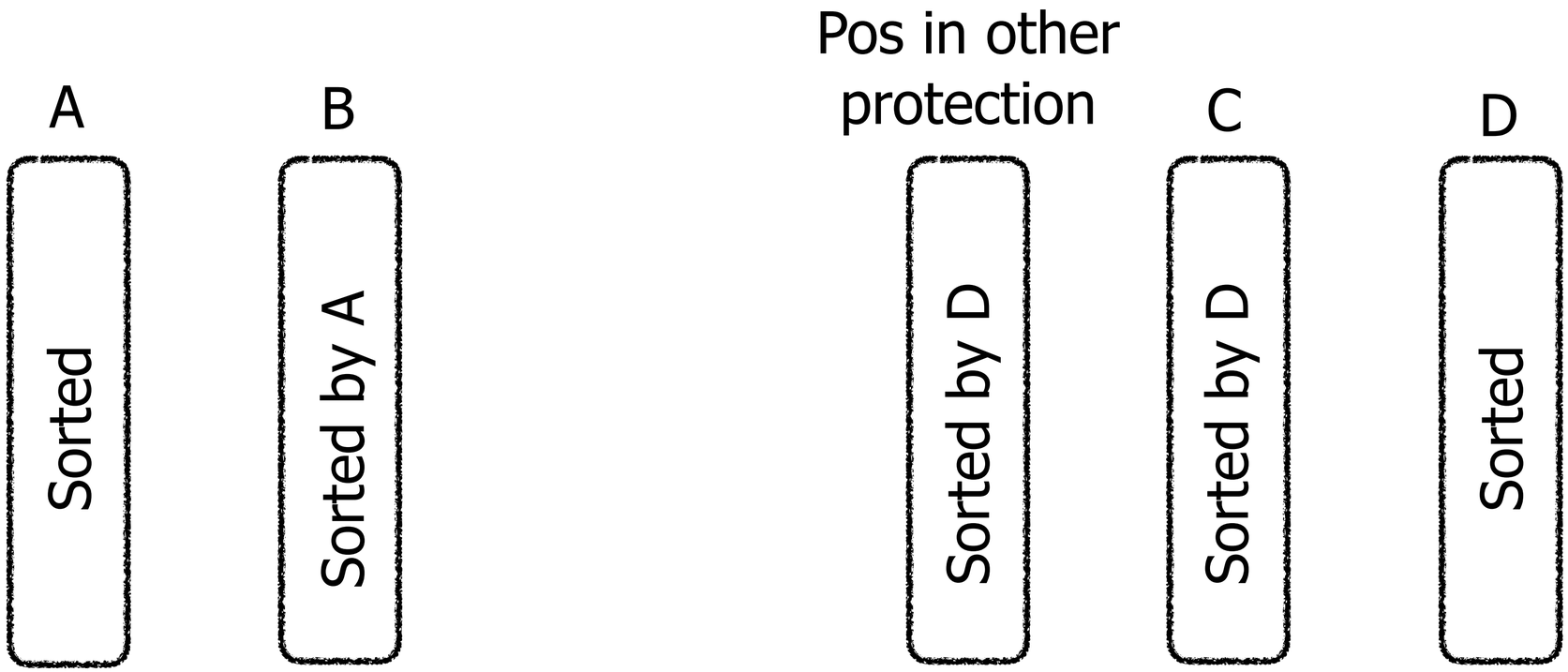
Select avg(B)
where A > 5 and A < 10





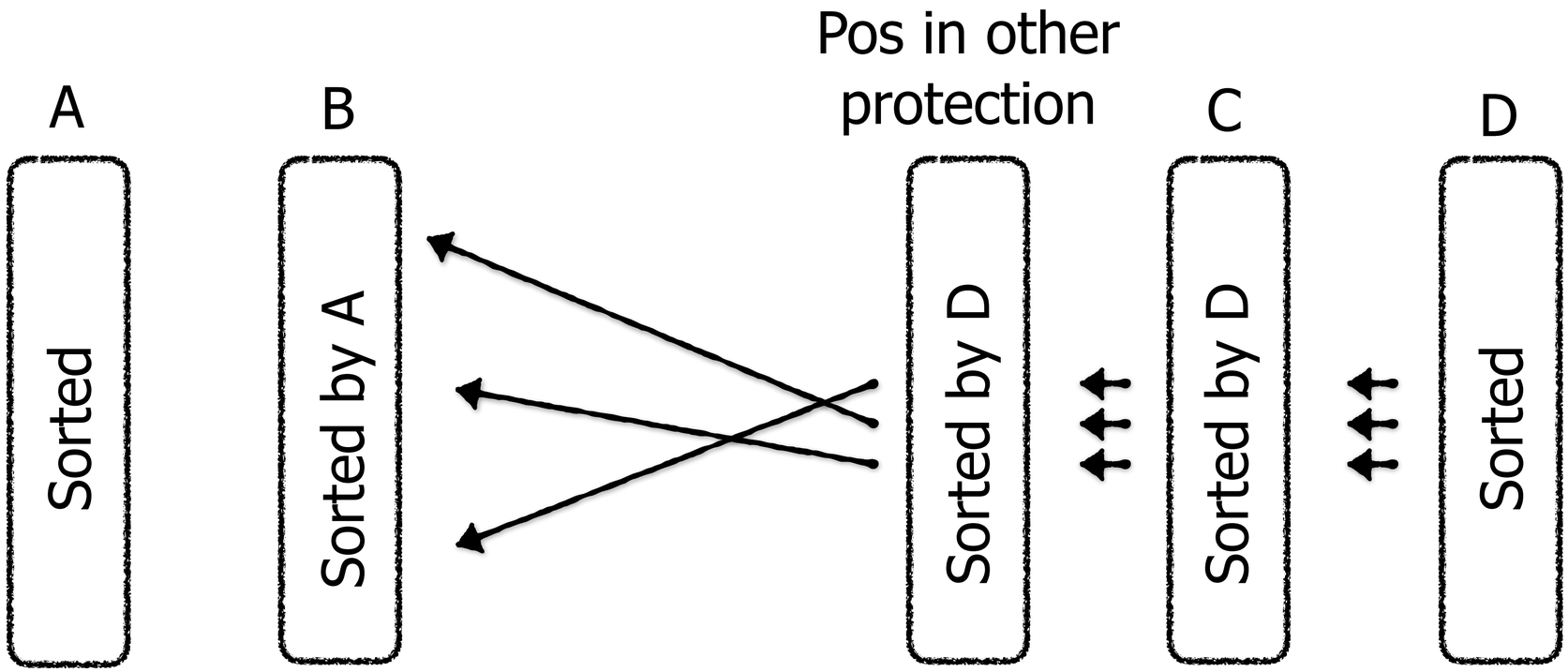
What if the queries do not target mutually exclusive columns?

Select C, B
where D > x and D < y



What if the queries do not target mutually exclusive columns?

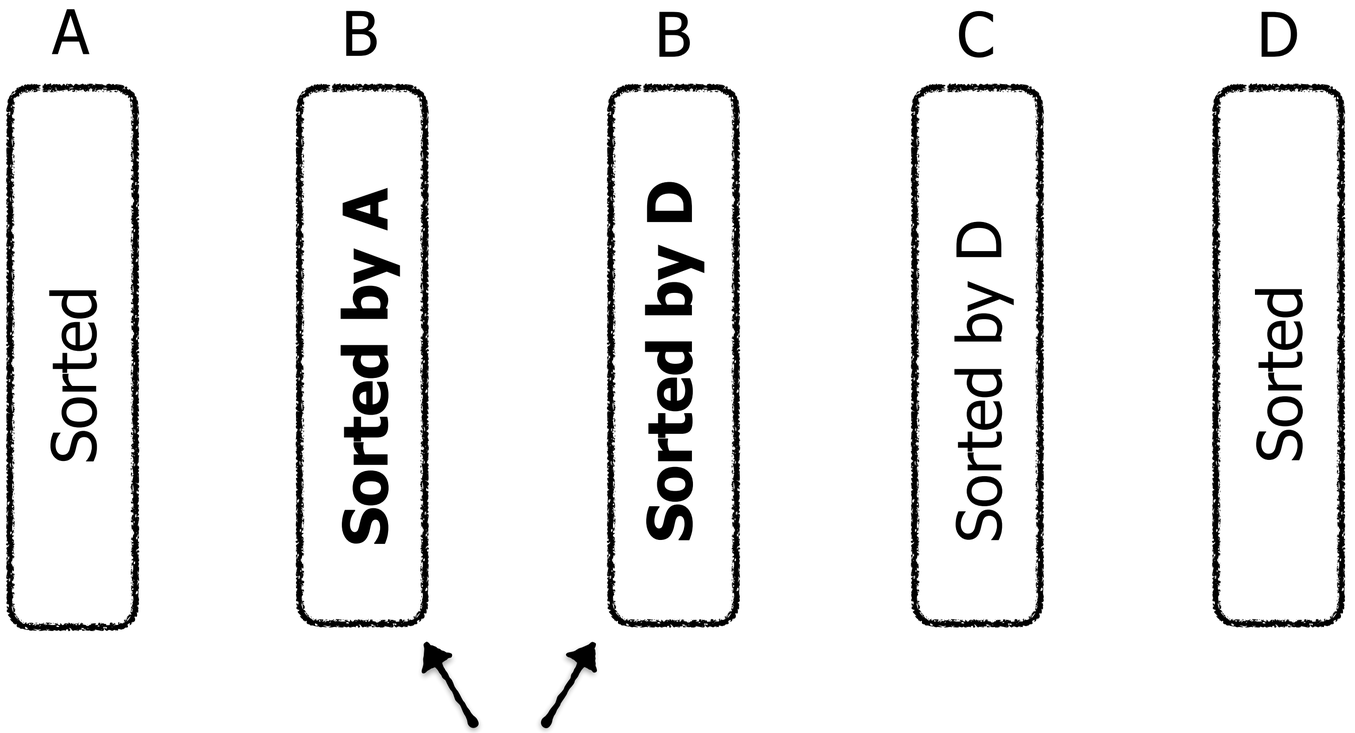
Select C, B
where D > x and D < y



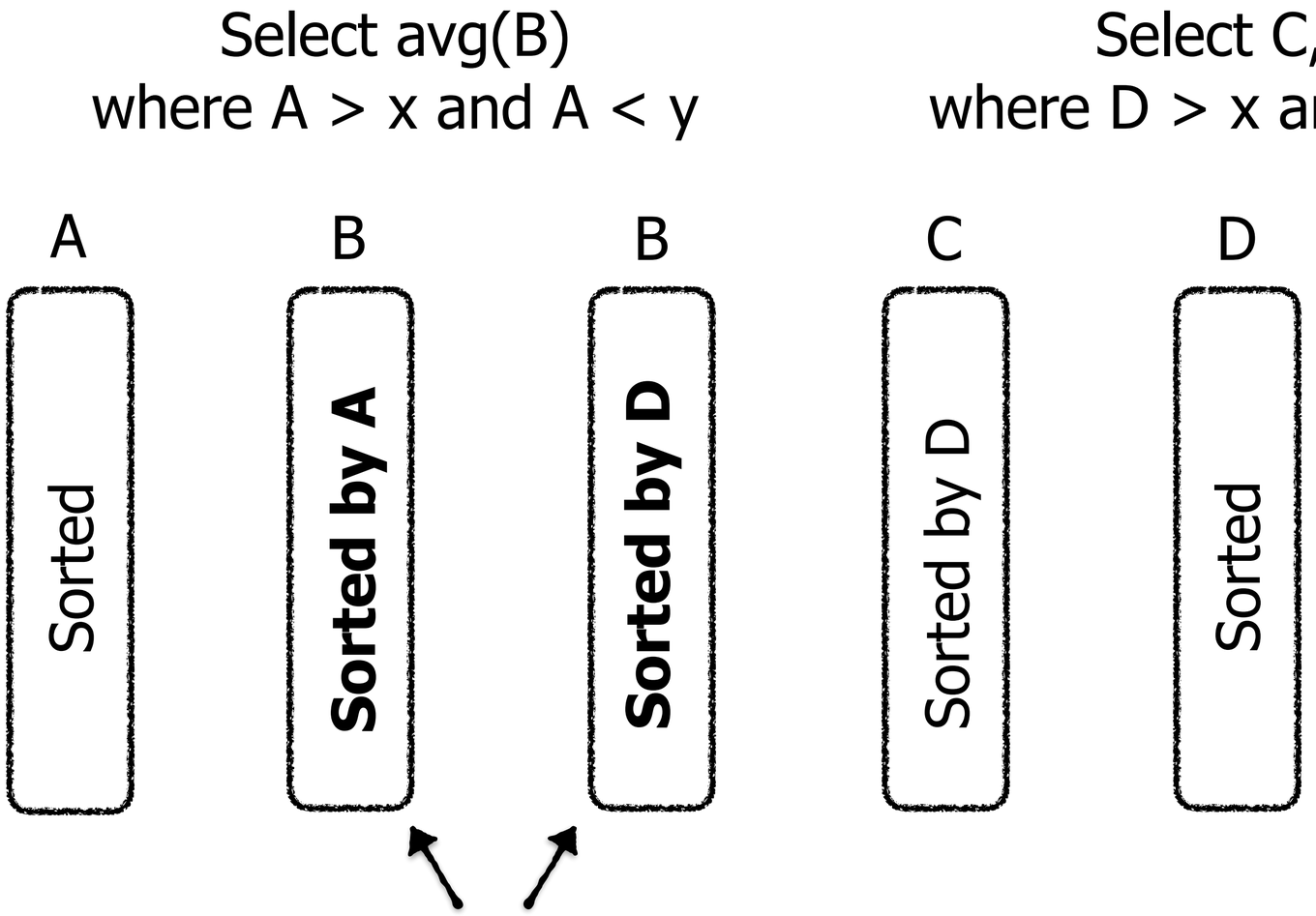
What if the queries do not target mutually exclusive columns?

Select avg(B)
where A > x and A < y

Select C, B
where D > x and D < y



We can duplicate some columns and sort them in different orders

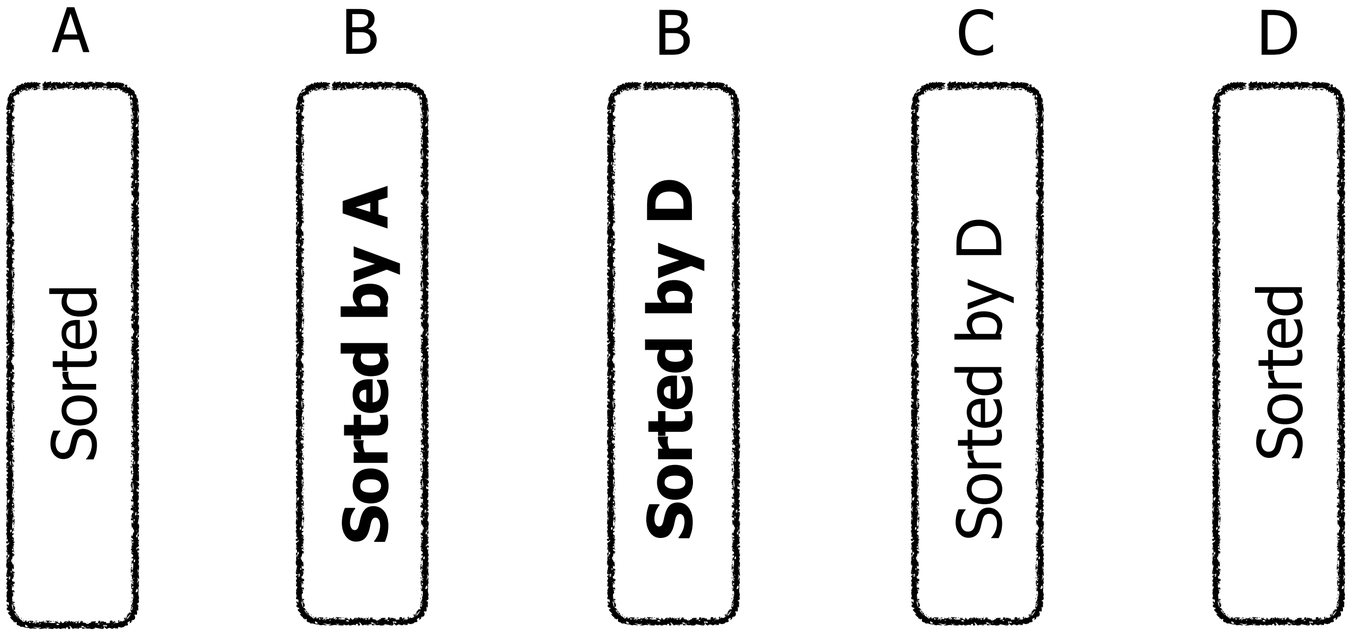


We can duplicate some columns and sort them in different orders

Downsides?

Select avg(B)
where A > x and A < y

Select C, B
where D > x and D < y



Downsides:

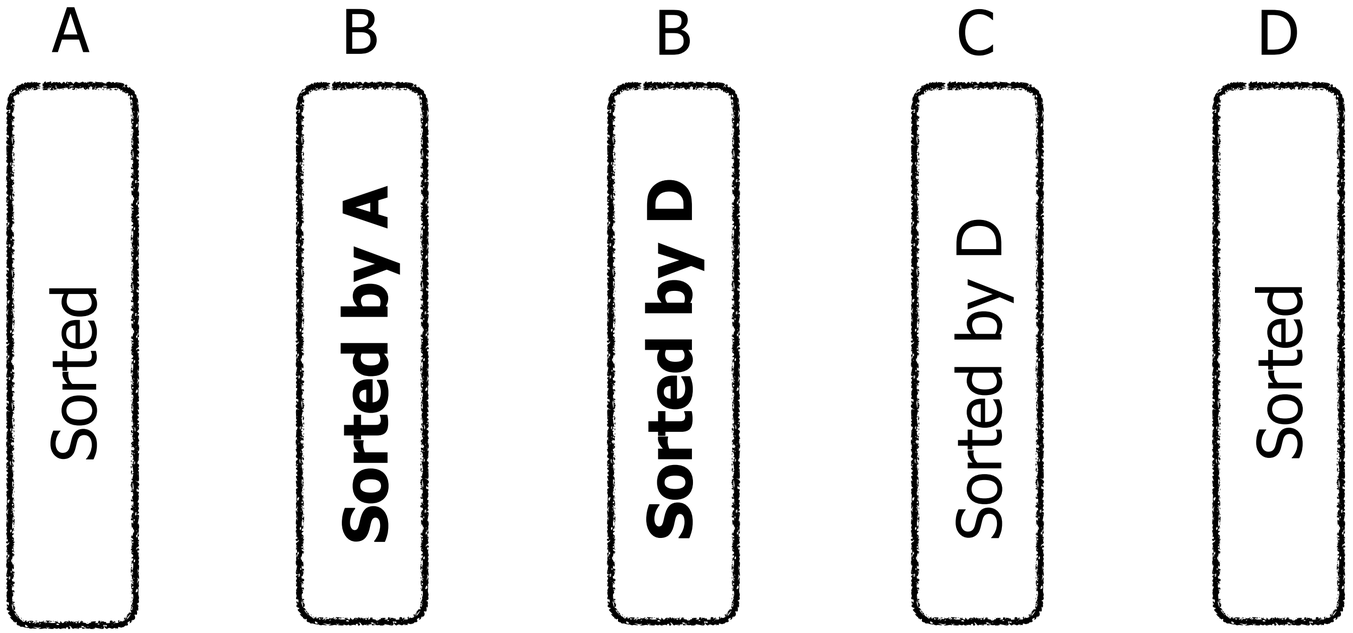
(1) More space

(2) Inserts are more expensive

(3) Construction can take a long time with many projections

Select avg(B)
where A > x and A < y

Select C, B
where D > x and D < y



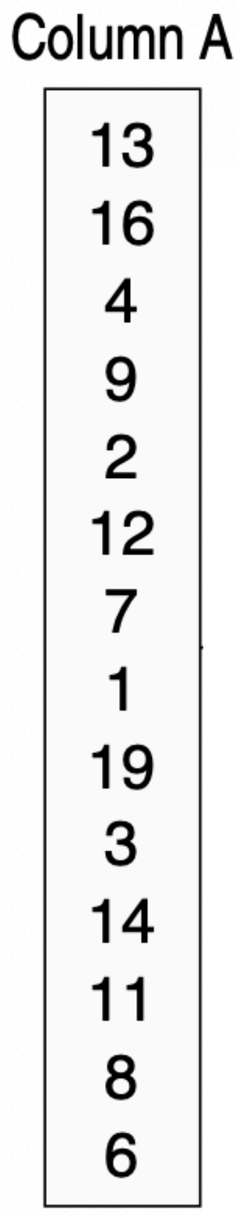
(1) More space

(2) Inserts are more expensive

Tackle this: → (3) Construction can take a long time with many projections

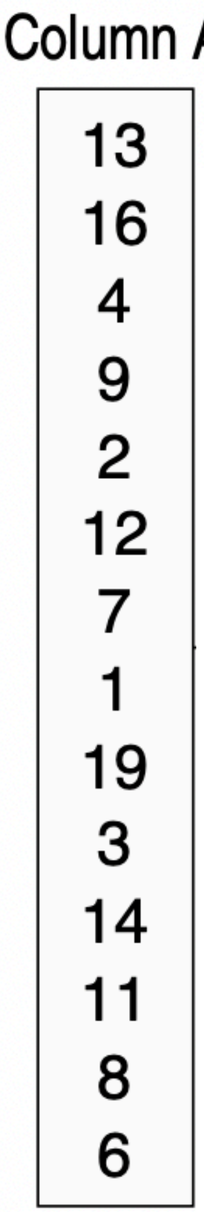
Database Cracking: Adaptively quick-sorting a column projection

Database Cracking: Adaptively quick-sorting a column projection



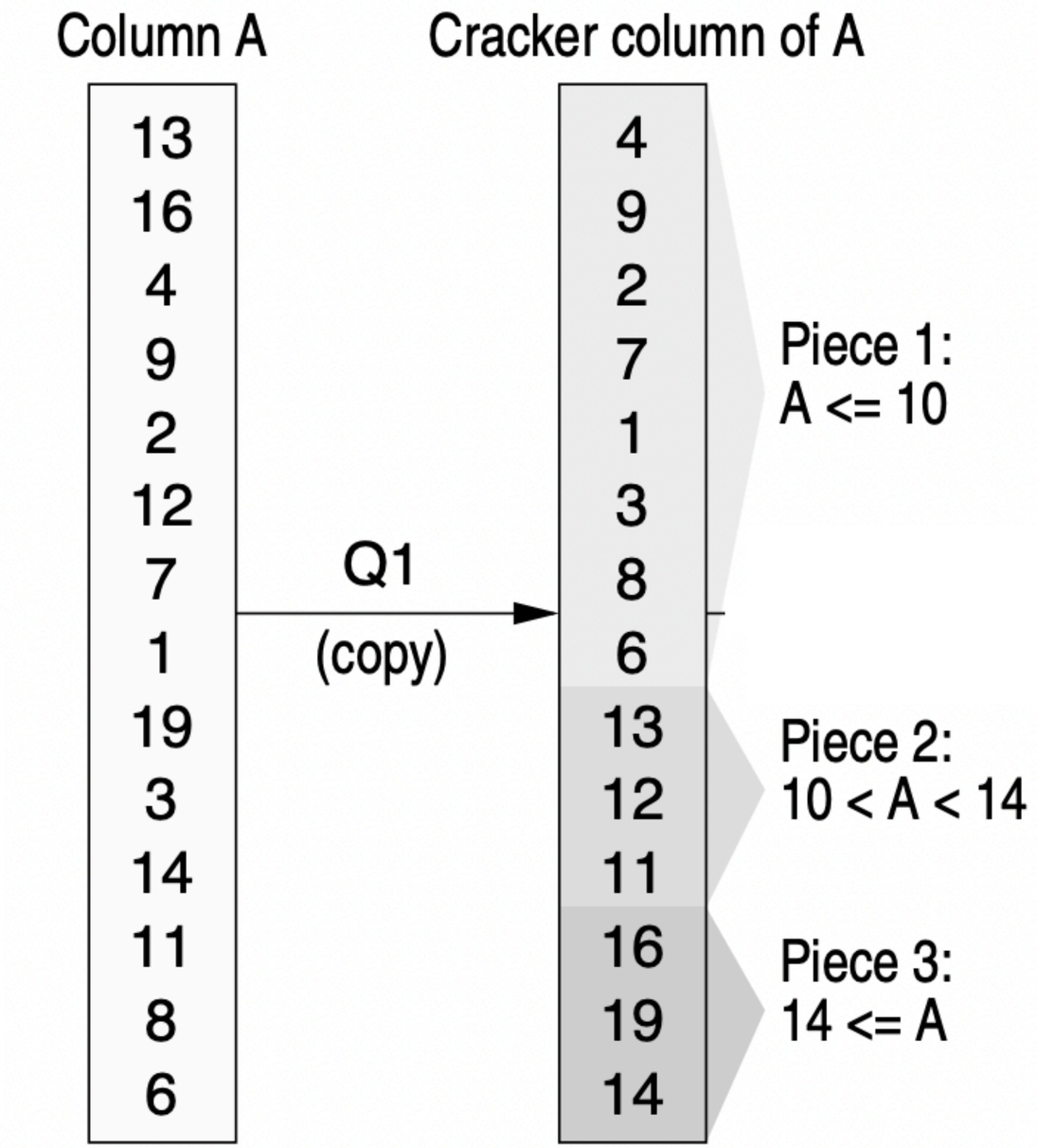
Database Cracking: Adaptively quick-sorting a column projection

Q1:
select *
from R
where R.A > 10
and R.A < 14

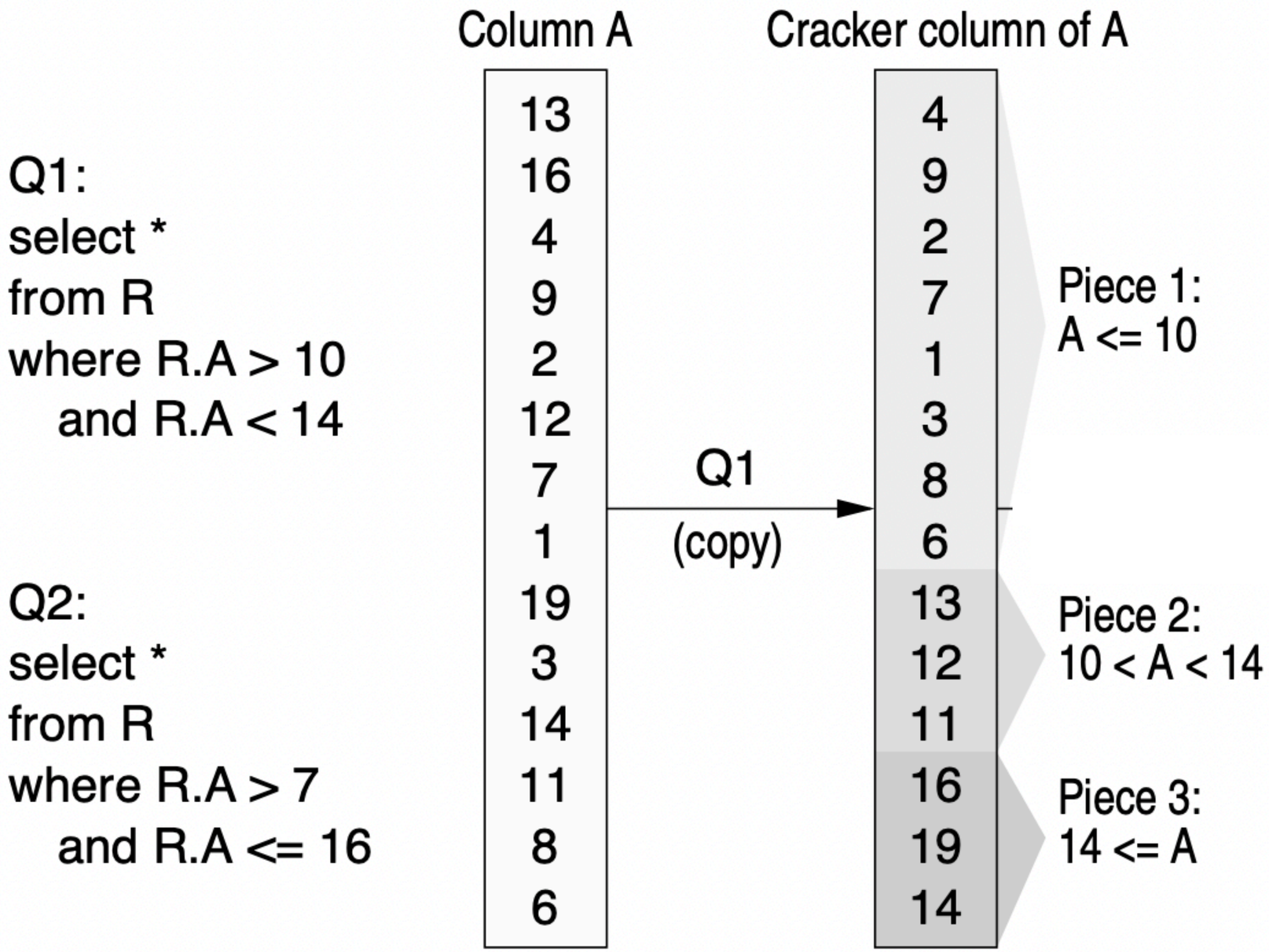


Database Cracking: Adaptively quick-sorting a column projection

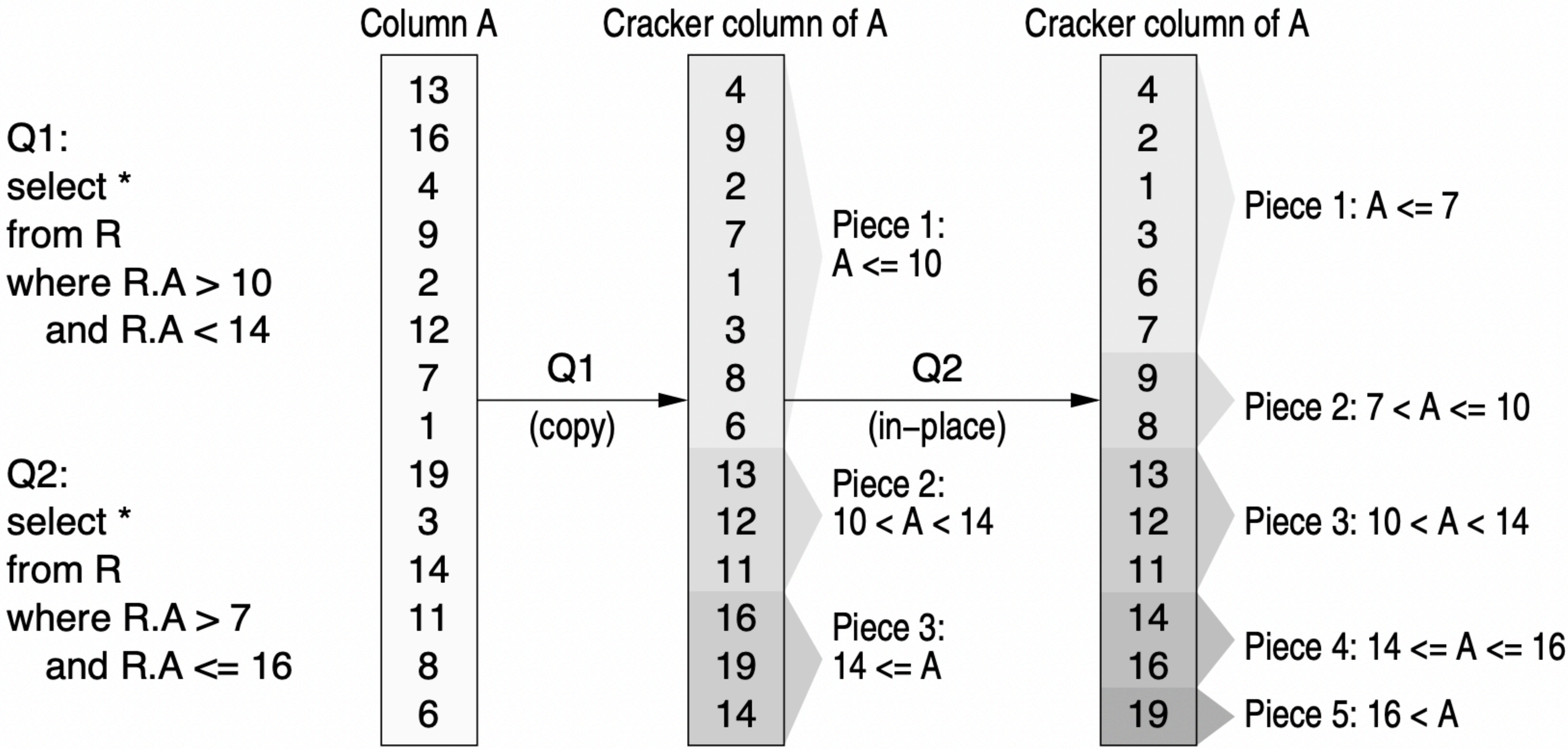
Q1:
select *
from R
where R.A > 10
and R.A < 14



Database Cracking: Adaptively quick-sorting a column projection

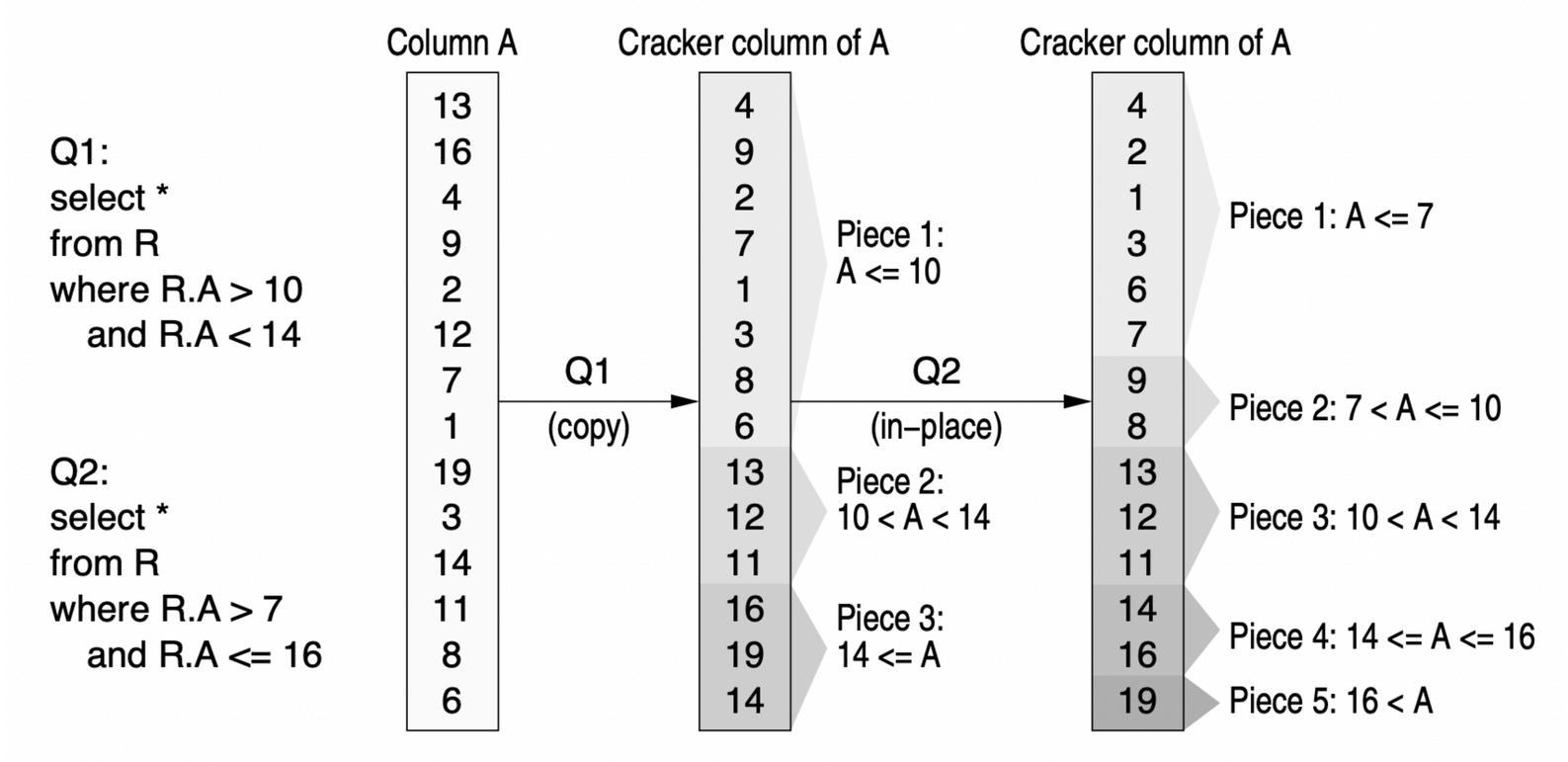


Database Cracking: Adaptively quick-sorting a column projection



Database Cracking: Adaptively quick-sorting a column projection

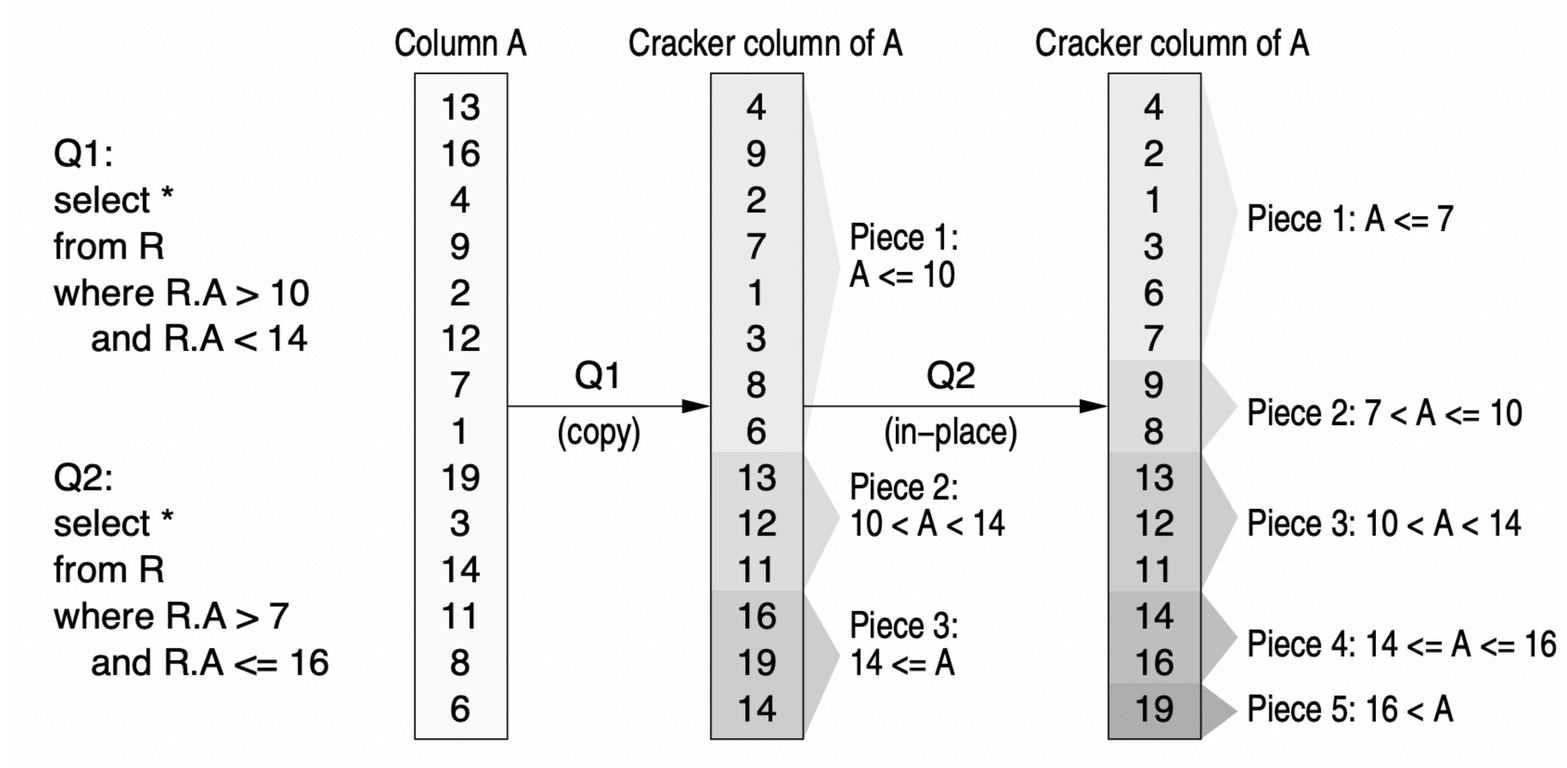
Queries adaptively refine the sorting and speed up subsequent queries



Database Cracking: Adaptively quick-sorting a column projection

Queries adaptively refine the sorting and speed up subsequent queries

We can begin querying the data immediately without creating too many projections from the onset, which would take a while



And now: office hours