

Efficient Scans

Catalog keeps track of all tables' directories

Directories allow reading multiple pages/extents asynchronously (good for SSDs)

Extents allow reading many pages sequentially (Good for disks)

Name	Data type	Size (Bytes)	Start
Customers			
Orders			

Internal Page Organization for Fixed-Sized Rows

Entries compactly packed at front of page

Can also use a bitmap to mark occupied slots

Metadata	N, etc.
Slot 1	
Slot 2	
...	
Slot N	
Free	

delete
Move

Metadata	Free Bitmap
Slot 1	101...1
Slot 2	
Slot 3	
...	
Slot N	

Variable-Sized Row Organization

Delimiters

Pointers

F1	\$	F2	\$	F2	\$
----	----	----	----	----	----

Smaller
No random access

		F1	F2	F2
--	--	----	----	----

More space
Random access (faster)

column_stores 11 / 156 56%

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

Great for queries that examine most columns

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

column_stores 16 / 156 56%

ID1 name1 email1 addr1 **salary1** ID2 name2 email2 addr2 **salary2** ID3 name3 email3 addr3 **salary3**

128B access 128B access 128B access

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

column_stores 22 / 156 56%

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

ID	Name	email	Addr	Salary

ID Name Email Addr Salary

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

↑ ↑ ↑ ↑ ↑

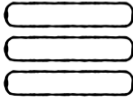
Select * from Customers where ID = 1

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

column_stores 26 / 156 56%

The two database families

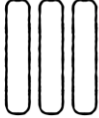
Row-Stores



Selective queries/updates

Online Transaction Processing (OLTP)

Column-Stores



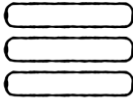
Large statistical calculations & batch updates

Online Analytical Processing (OLAP)

column_stores 28 / 156 56%

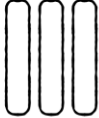
The two database families

Row-Stores



Postgres, MariaDB, etc.

Column-Stores



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

column_stores 33 / 156 56%

The two database families

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

1975 2000 2005 now

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**

column_stores 35 / 156 56%

Should we store each column with a materialized ID?

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

column_stores 42 / 156 56%

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.

column_stores 44 / 156 56%

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

column_stores 47 / 156 56%

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

A > 10 B C

Scan & evaluate predicate

Early Materialization: express intermediate results in a row-store format

column_stores 52 / 156 56%

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

A > 10 B C

Scan & evaluate predicate

Early Materialization: express intermediate results in a row-store format

Problem? Many random accesses

column_stores 58 / 156 56%

Late Materialization requires more memory to withhold intermediate results, but it entails less random access and is therefore far faster than early materialization.


A > 10 B < 20 min(C)


Fetch values & take their min

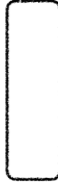
Result

column_stores 64 / 156 56%

Which column should we filter on first?

$A > 10$


$B < 20$


$\min(C)$


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

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


column_stores 75 / 156 56%

Handling Insertions

Option 2: In-memory Buffering


Insert into table (, ,)

A




a1
a2

B




b1
b2

C



c1
c2



SSD

Cost model: $O(1/B)$

Cheap insertions are possible :)


column_stores 78 / 156 56%

Handling Deletes

Option 1: In-place Deletes


delete from table where A = "x"

A=ID




Hole

B




Hole

C



Hole



SSD

Problems:

- Cost: $O(\#cols)$**
- Holes slows down queries**
- Requires 1 extra bit to note a hole**

column_stores 82 / 156 56%

Handling Deletes

Option 2: employ delete column

delete from table where A = " "

Pos

...

X

Eventually merge with column

Cost? $O(N / \text{Buffer size})$

A=ID B C SSD

column_stores 87 / 156 56%

Handling Updates

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

First delete

Then Insert

Time Deletes

t1 x

Buf1 Buf2 Buf3 Time

x b1 c1 t2

A B C SSD

Problem: Which came first?

Fix using timestamps

column_stores 90 / 156 56%

Handling Updates

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

Time Deletes

t1 x

Buf1 Buf2 Buf3 Time

x b1 c1 t2

A B C SSD


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

column_stores
106 / 156
56%

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 heirachty**

column_stores
111 / 156
56%

(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 applicible if there are very few values.

	Horse	Cat	Dog	Specie
	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

column_stores
116 / 156
56%

(2) Dictionary Encoding

Employ a dictionary with smaller strings to represent larger ones

Pro: applicable across larger domains

Con: slower as we need to read more bits

	Dictionary	Compressed	Specie
	00 Cat	00	Cat
	01 Dog	01	Dog
	10 Horse	00	Cat
	11 Parrot	00	Cat
		01	Dog
		00	Cat
		10	Horse
		00	Cat
		11	Parrot
		10	Dog

Specie

Cat

Dog

Cat

Cat

Cat

Cat

Cat

Cat

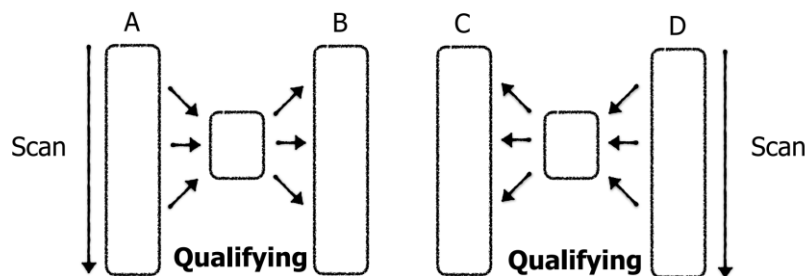
Dog

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

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

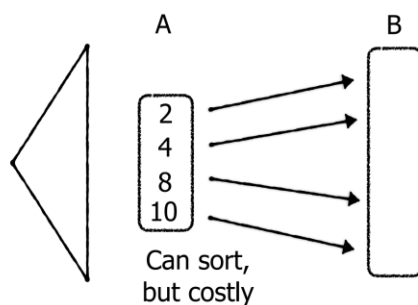
Select avg(B)
where A > x and A < y

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



We can answer both queries with full scan and late materialization

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



Can sort,
but costly



Access to B b becomes "skip-sequential" rather than random, but this is still not as good as sequential

column_stores 135 / 156 56%

Column Projections: sort subset of columns by one column

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

A

Sorted

B

Sorted by A

C

Sorted by D

D

Sorted

Like having multiple clustered indexes on subset of columns

column_stores 143 / 156 56%

Column Projections: sort subset of columns by one column

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

A

Sorted

B

Sorted by A

C

Sorted by D

D

Sorted

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

What if the queries do not target mutually exclusive columns?

column_stores 146 / 156 56%

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

A

Sorted

B

Sorted by A

B

Sorted by D

C

Sorted by D

D

Sorted

Downsides:

- (1) Requires more space, but column compression can help
- (2) Inserts are more expensive, but ok if done in batch
- (3) Construction can take a long time with many projections

