

Column Stores

Alvin Cheung

Aditya Parameswaran



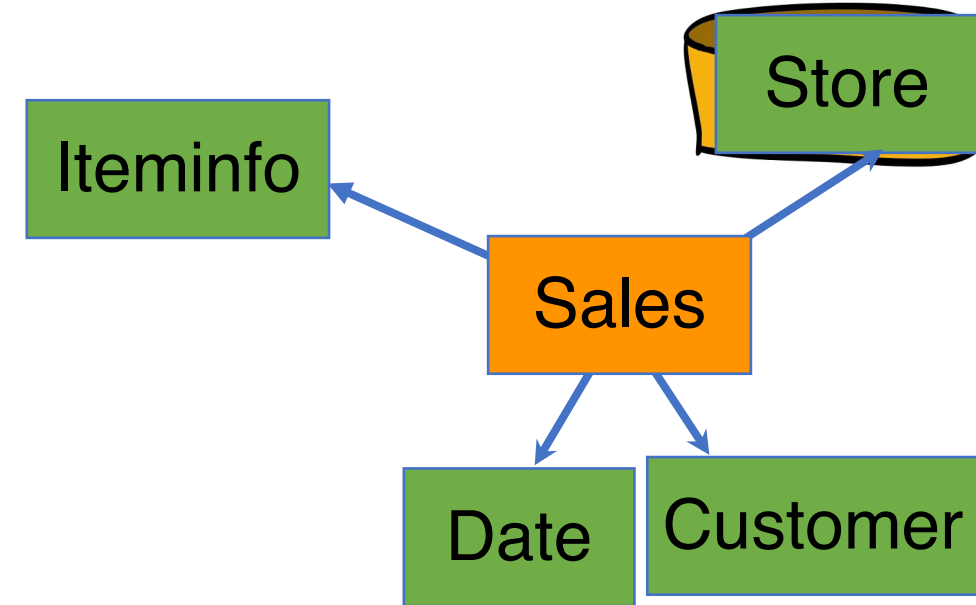
Star Schema

- Data warehouse example:

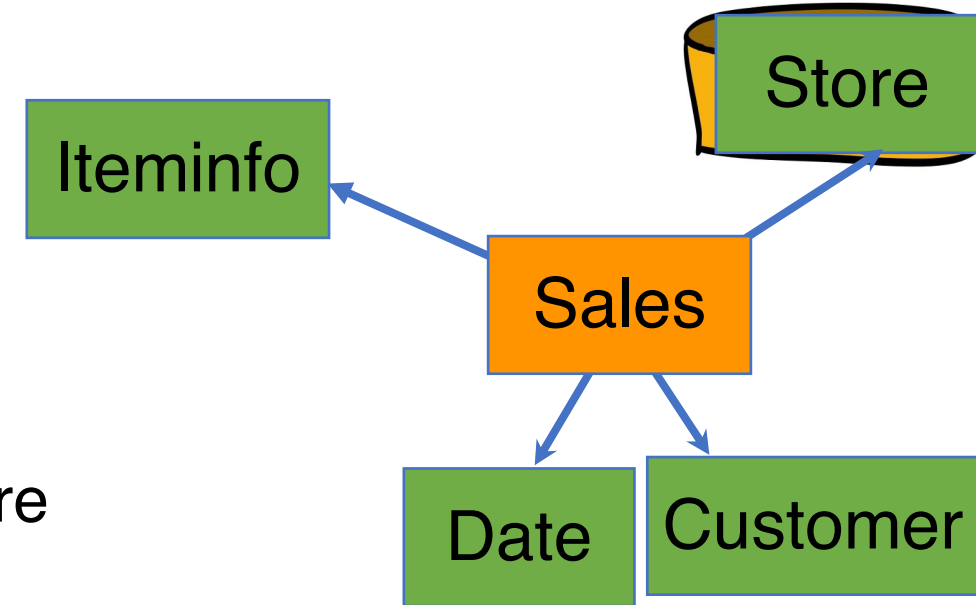
- Fact table: Sales (itemid, storeid, customerid, date, number, price)
- Dim table: Iteminfo (itemid, itemname, color, size, category)
- Dim table: Store (storeid, city, state, country)
- Dim table: Customer (customerid, name, street, city, state)
- Dim table: Dateinfo (date, month, quarter, year)

- Reminder:

- Fact table records info. about an event
- Dim. table records auxiliary info.
- Even this simple example has lots of attributes.
- Fact and dim. tables in practice often have 100s of attributes each



Star Schema



- However, most analytical/OLAP queries don't require accessing all 100-odd attributes across all tables
- Rarely will you be doing `SELECT *` on a 1PB Sales fact table.
- Instead, most queries would touch 2-3 attributes at a time to construct a data cube

```
SELECT category, country, month, COUNT(number)
```

```
FROM Sales NATURAL JOIN Iteminfo NATURAL JOIN Store  
CUBE BY category, country, month
```

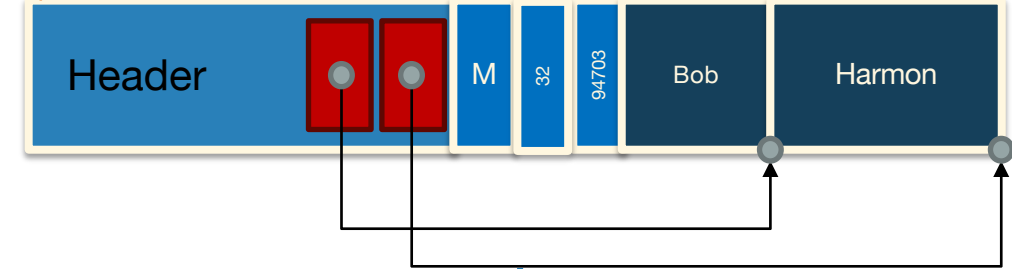
Recall Our Slotted Page Representation



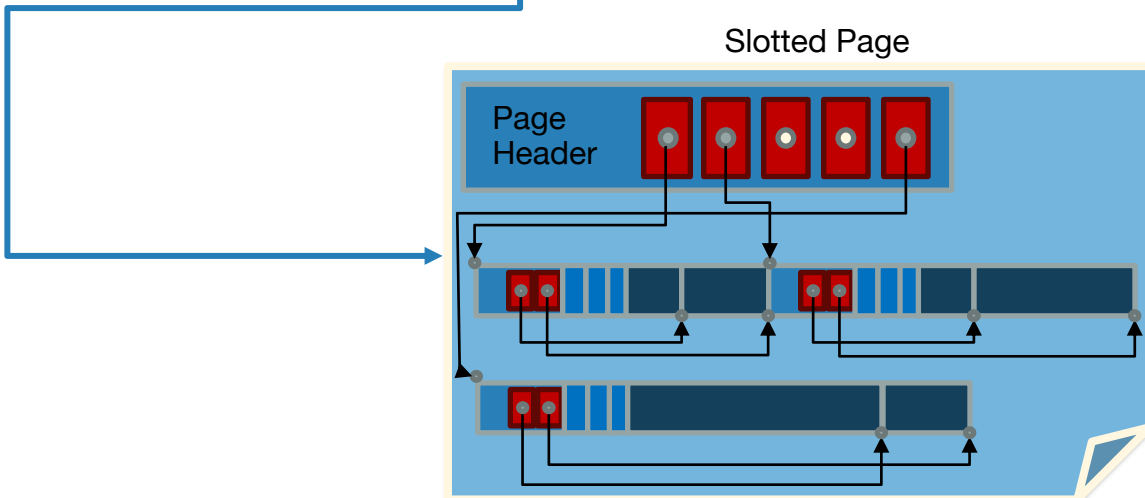
Record

Bob	Harmon	M	32	400
Varchar	Varchar	Char	Int	Int

Byte Representation of Record



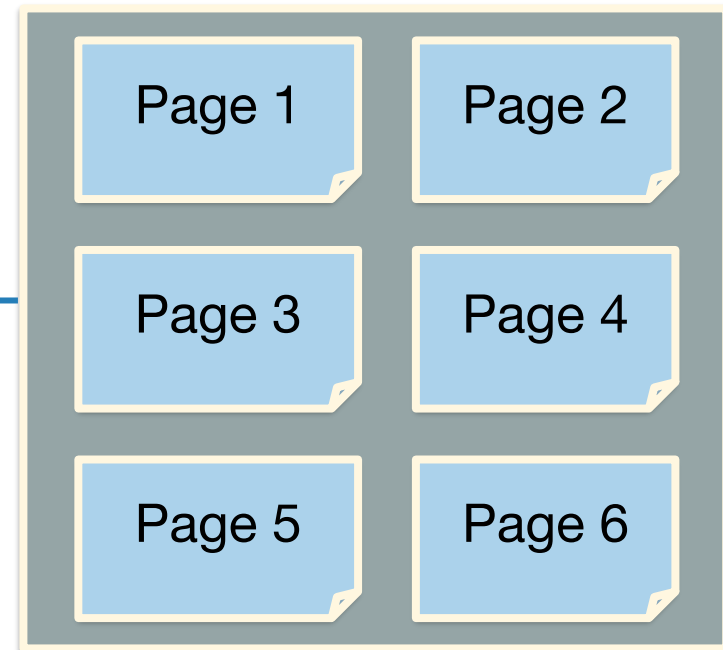
Slotted Page



SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	\$400
443	Grouch	Oscar	32	\$300
244	Oz	Bert	55	\$140
134	Sanders	Ernie	55	\$400

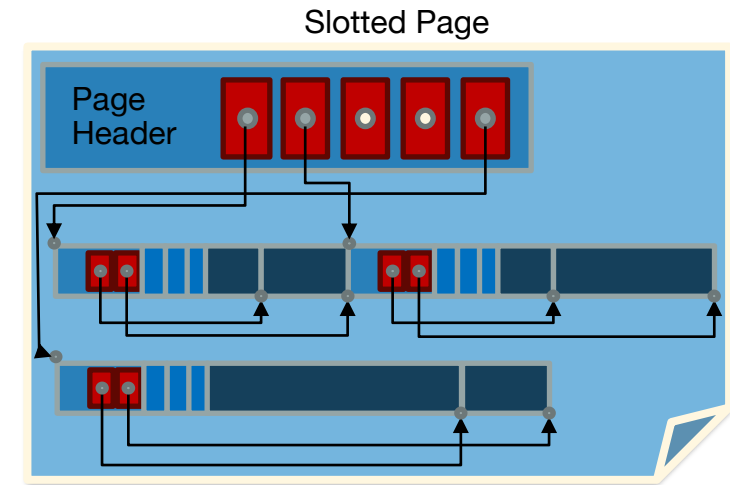


File



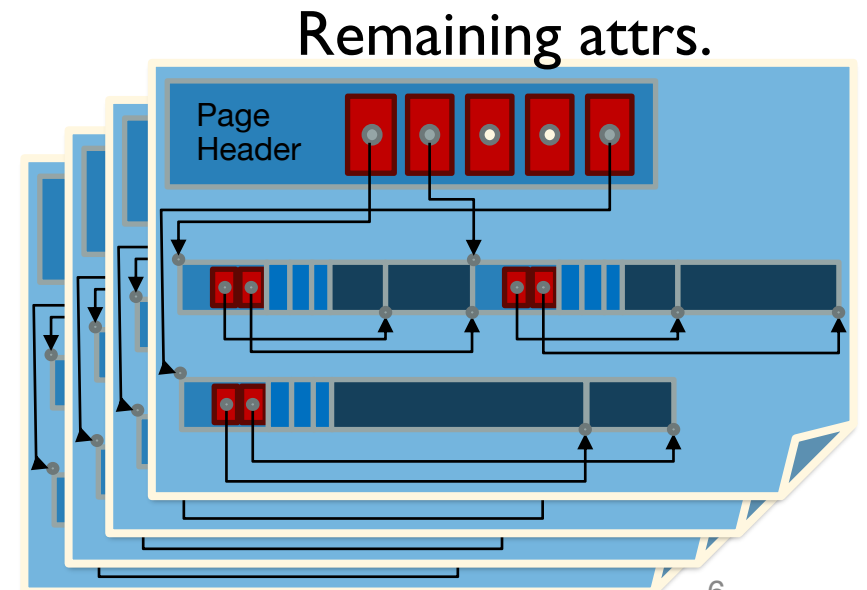
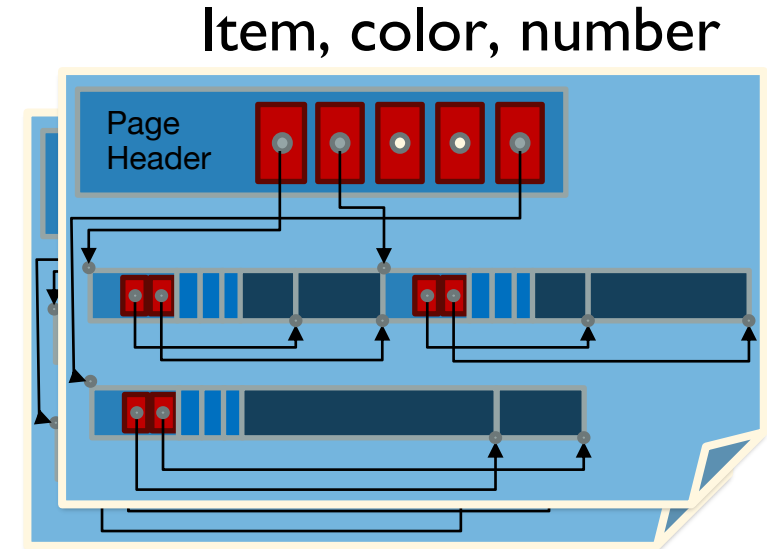
Traditional Storage: Row-oriented

- RDBMSs store row-oriented data across disk pages
- Page will store all attrs. per row, incl. variable and constant length fields
- Imagine a 100+attribute fact table Sales.
- Example query: `SELECT item, color, SUM(number) FROM Sales CUBE BY item, color`
- For this CUBE BY query the remaining 97/100 attrs. is wasted
 - → Only 3% of the I/O is useful
 - Remaining attrs. will get projected out
- Unnecessary work:
 - Reading redundant data from disk
 - Having to project out this data within memory



An Alternative: Split Columns?


- Why not store groups of columns separately from each other
- Eg., if the most frequent query is:
 - `SELECT item, color, SUM(number)`
`FROM Sales GROUP BY item, color`
- Storing *item, color, and number* separate from rest, will help avoid reading and processing ~97% of the data
- Q: Does this suffice?
- A: No! We need a way to reconstruct original tuples. Recall lossless decomposition.
- Return to this later



Column-Oriented vs Row-Oriented Storage

- Basic idea:
 - Storing groups of columns separate from each other
- Can be one column each or subsets of columns
- Some systems allow same column to be repeated, we'll discuss later

Item	Color	Size	Number



Item	Color	Size	Number



Item	Size	Color	Number



Column Stores

- *Column stores* are a specialization of RDBMSs for OLAP that use column-oriented storage (among many other innovations)
 - Traditional RDBMS, in contrast, called row stores
- Many industrial offerings: Vertica, Vector, Druid, Greenplum, Amazon Redshift, SAP IQ, ParAccel ...
- Column store ideas have made their way into traditional RDBMSs like PostgreSQL, MS SQL Server, DB2, Oracle, ...
- Column storage ideas have been used in NoSQL, e.g., Dremel, Impala, HBase, ...
- Ideas have been around for >2 decades but really rose to prominence in the late 2000s

Reconstruction Alternative 1: Explicit IDs

- If we store columns separate from each other, how do we reconstruct the tuples?

Item	Color	Size	Number

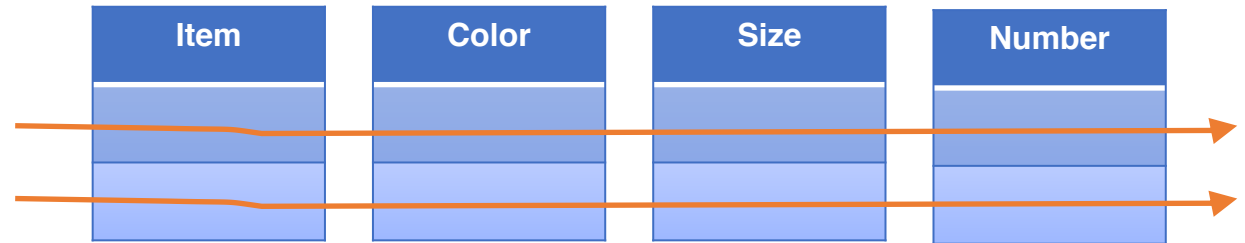
- One option, explicit IDs
 - Join to reconstruct
 - Not necessary for each column to be stored sorted
- Downside: extra storage and processing overhead

TID	Item	TID	Color	TID	Size	TID	Number

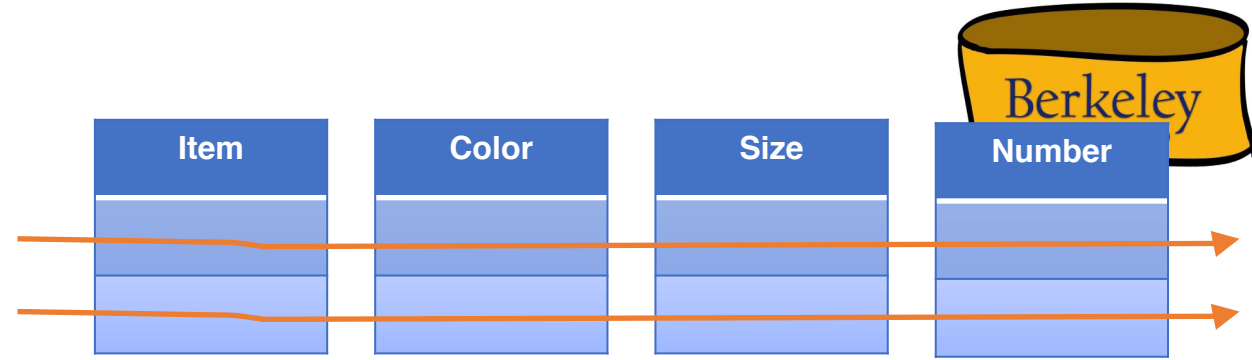
Alternative 2: Virtual IDs

Instead, most column stores implicitly use position to record the n th value for a given column.

- If fixed width, easy to lookup
 - $\text{Start} + (n \times \text{width})$.
- Else, will need either pointers (recall: var length fields in record encoding) or encode length as part of each var. length field.



Drawbacks?



Sound magical! Do we have a better design than row stores?

- Imagine wanting to read a single tuple
 - Can simply read the corresponding page in row store
 - In column stores need multiple I/Os one per column group
- Specifically, hard to do updates: will need to individually update all columns
- Column stores work better when we want to read a *subset of columns of most of the tuples*
 - Thus, a good fit for OLAP!
 - E.g., constructing a data cube of some select columns

OK, so are we done?

- Not quite.
- Turns out there are lots of additional improvements beyond columnar storage in column stores...
- First up, compression
- Compression wasn't relevant in row stores because we were storing heterogeneously-typed columns.

Compression

- Q: How would you go about compressing a collection of salaries? states? product codes? zipcodes?

Desirable properties:

- Must reduce space considerably
- Must be quick to decompress
 - Most “fancy” approaches are too slow for OLAP

First Approach: Run-Length Encoding (RLE)

RLE simply encodes a sequence of values based on the number of times each value appears

A A A B B B B A A A C
<3, A> <5, B> <3, A> <1, C>

Q: When would this work well?

- Works really well when data is already pre-sorted by the attribute; doesn't work too well with randomized orders
- Doesn't work too well with numeric data with many unique values
- Advantage: can use it effectively if we want to do aggregates:
e.g., $\text{SUM}(\langle 3, 10 \rangle \langle 5, 6 \rangle) = 3 * 10 + 5 * 6$

Second Approach: Dictionary Encoding

Mapping every distinct value into another that takes less space.

Eg, instead of using variable length strings for encoding states, can encode states into a single byte ($2^8 > 50$)

Alabama = 0, Arkansas = 1, ...

Q: When does this work well?

Strings where once again # of distinct values is small; doesn't require ordering

Third Approach: Frame of Reference

Store first value, rest stored as “deltas” from the previous one

e.g., 1000, 1001, 1003, 1004, ...

Stored as: 1000, +1, +2, +1 etc.

Q: When does this work well?

When the data is already sorted, or if there is locality: works better with numbers than with strings

Fourth Approach: Bit Vector Encoding

Encode every single possible value as a bit vector

1 1 3 2 2 3 1 encoded as:

Bit string for 1: 1 1 0 0 0 0 1

Bit string for 2: 0 0 0 1 1 0 1

Bit string for 3: 0 0 1 0 0 1 0

Q: When does this work well?

- Small number of distinct values
- Benefit of the bit strings is that they can be easily processed on
 - If we wanted to get all the values corresponding to 2, grab the bit string for 2
 - If we wanted 2 or 3, we can OR the two bit strings
- Bit vectors are very efficiently manipulated by modern processors

So: what compression should we use?

- Answer: it depends
 - On the type of data
 - The locality
 - What you want to do with it
 - The # of space you have
 - etc.

Now, back to Column Stores

- Since columns are stored separate from each other, we can compress better than if we were to apply it to entire rows
- e.g.,
 - <Washington, M, 100000>,
<California, F, 200000>,
<Louisiana, O, 150000>
- Can't easily apply any of the compression schemes at the row level
- But:
 - States (Washington, California, ...) can be dictionary encoded
 - Gender (M, F, O, ...) can be bit vector encoded
 - Salaries can be FOR-encoded
- Thus compression is more beneficial for column stores than row stores

Should we store entire columns separate from each other?

- Answer: it depends
 - Entire columns lead to more compression
 - But if multiple columns are often accessed together may want to store them together — known as “*column groups*” or *projections*
- In general, depends on the workload.
 - Say we primarily issue two queries on a relation with attributes A—Z, one with A, B, C and another with A, B, D?
 - Can store $\langle A, B, C, D \rangle$ and then $\langle E-Z \rangle$, or
 - Can store $\langle A, B, C \rangle$, $\langle D \rangle$, $\langle E-Z \rangle$, (+symmetric alternative) or
 - Can store $\langle A, B \rangle$, $\langle C \rangle$, $\langle D \rangle$, $\langle E-Z \rangle$, or
 - Can store $\langle A \rangle$, $\langle B \rangle$, $\langle C \rangle$, $\langle D \rangle$, $\langle E-Z \rangle$
 - The best choice depends on the frequency of the queries and the distributions of values in A, B, C, D, as well as combinations of A, B, C, D.

Column Stores

- Two key ideas so far:
 - Store (groups of) columns separate from each other
 - Columns can be compressed
- Other ideas:
 - Many different layouts
 - Different sort orders, different subsets of cols
 - Late materialization
 - Vectorized processing
 - Update support via WOS

Updates via a WOS

- Many column stores *also* support inserts, deletes, updates
- For example in Vertica
 - Handled via a separate row-oriented in-memory Write-Optimized Store (WOS) where new/updated tuples are stored
 - For “invalidating” existing records, maintain an in-memory list of all row numbers that have been deleted/updated
 - Can skip those rows during processing
 - This WOS data is periodically merged back into the columns

Takeaways

- For OLAP, column-oriented storage trumps row-oriented storage
 - Many tricks beyond splitting columns up
 - compression, late materialization, redundant layouts, efficient write processing
- For OLTP, the costs of many random accesses for updating columns makes a columnar layout not worth it
 - Hence OLTP systems look more traditional, and typically opt for row-oriented storage
- Many systems are now opting for hybrid layouts to try to support both OLAP and OLTP in the same system