Column Stores and Row Stores

Are they really that different?

By Sid Premkumar, Emmanuel Amponsah

Agenda

Intro to Row and Column Stores
(What are these? Pros and Cons of each design?)

Problem & Motivation

Context
(Background for experiment)

Implementing Row-Stores as Column Stores
(Why don't we just make row stores behave like column stores?)

Column Store Execution
(What are the optimizations for the Column Store?)

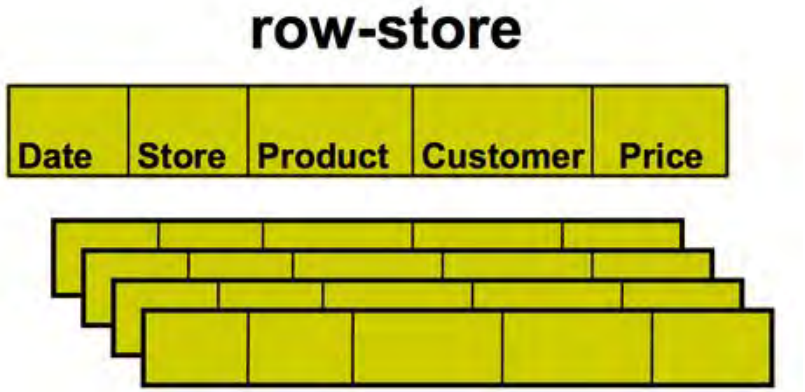
Row Store

What is it?

Storing data in successive blocks

Pros and Cons?

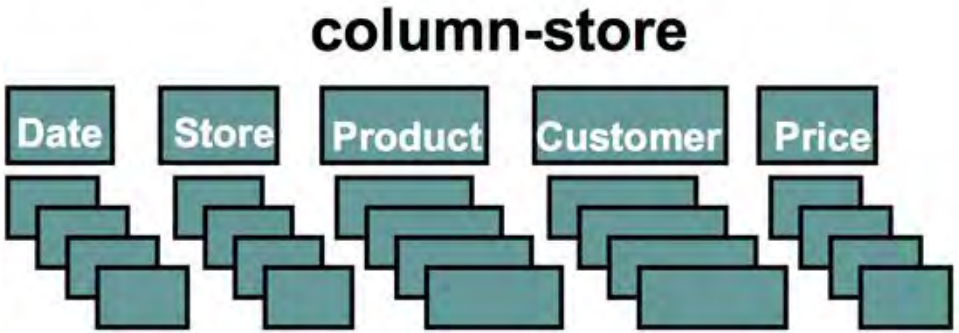
+ Easy to append new records
- Reads unnecessary data



Column Store

What is it?

Store data in separate pages



Pros and Cons?

+ Only read relevant data
- Tuple writes may require multiple seeks

Problem & Motivation

What is the problem and why is it important?

- Lots of legacy systems are built in row stores (would be expensive, time consuming to switch)
- It would be a game changer because then people could switch from column/row storage quickly

Why can’t row-store just emulate column store?

- This is what the paper is trying to figure out

https://svitla.com/blog/data-warehouse-vs-database

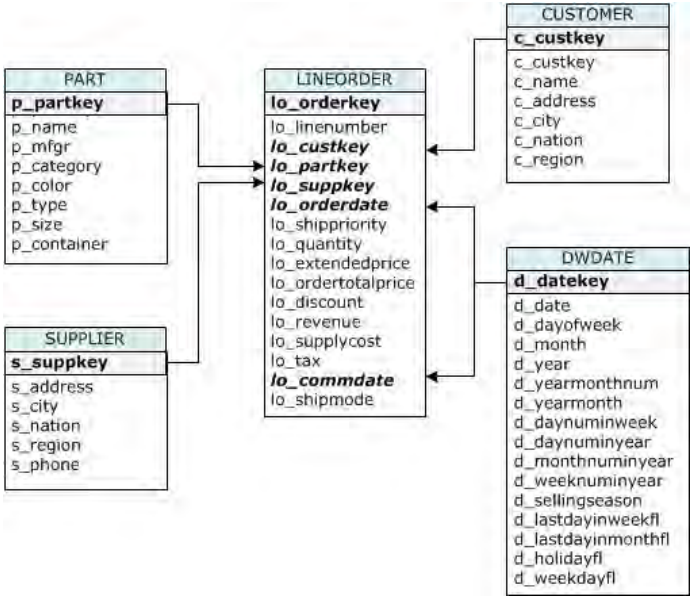
Context

How are we storing our data?

Star Schema

Where is the data coming from?

Generated in accordance with the Star Scheme



https://www.pilosa.com/use-cases/retail-analytics/

Context

Experiment setup

System X - Our Row Store Machine

C-Store - Our Column Store Machine

Each system has a materialized view (MV)
View version

https://www.wikiwand.com/en/Red_Hat_Enterprise_Linux

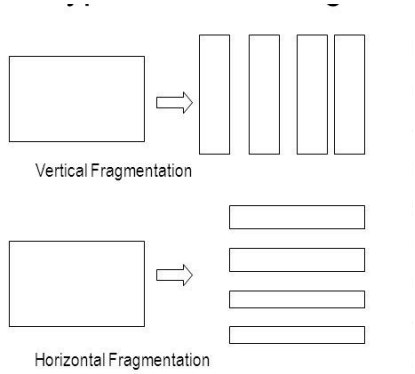
Implementing Row Store as Column Store

What are some approaches for row stores to emulate column stores?

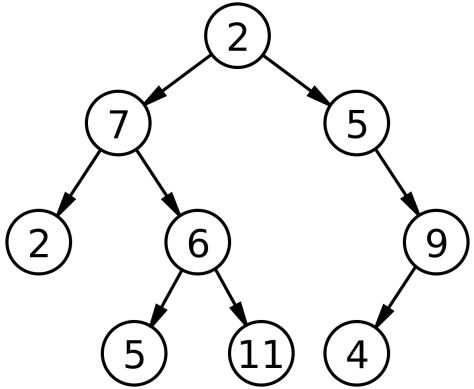
Vertical Partitioning

Index only plans

Materialized Views



https://slideplayer.com/slide/3422355/



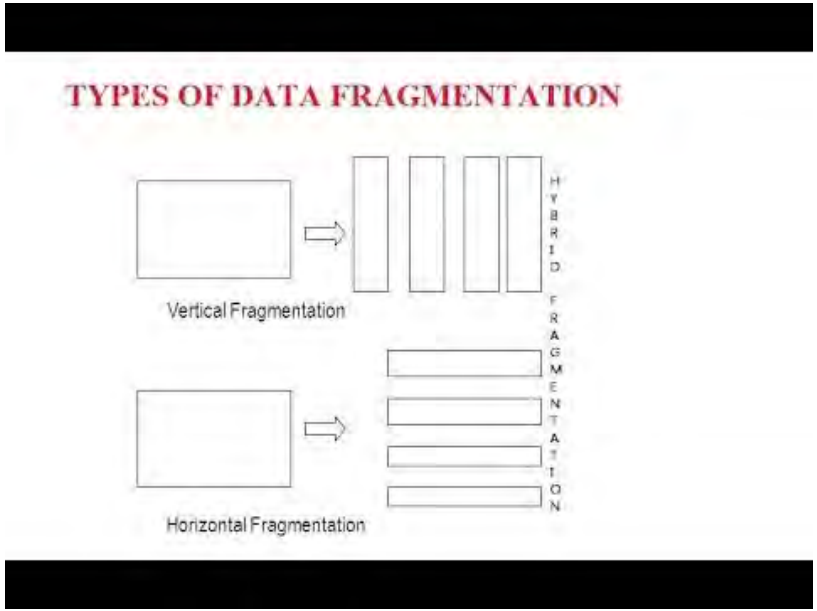
https://www.wikiwand.com/en/Bin

Vertical Partitioning

What is it?

To fully emulate vertical partitions, you add a position column to every table

Creates one physical table for each column in the schema, where the *i*th table has two columns one for the schema and the other for position column



Vertical Partitioning (cont.)

How do we join this partitioned data (what type of joins can we use)?

We can pick from:

- Hash Joins
- Index Joins
- Sort Merge Joins

Hash Joins

What is a Hash Join?

A hash join is performed by using one dataset into memory based on join columns and reading and probing the hash tables for matches

This happens in two steps; first a hash table is created using the contents of one relation **(build)**

After the build stage, scan the other relation for reach row probe the relevant rows by looking at the hash table **(probe)**

Index Joins

What is an Index Join?

An index join is a join that uses an index intersection with two or more relations to fulfil a query completely

Sort-Merge Joins

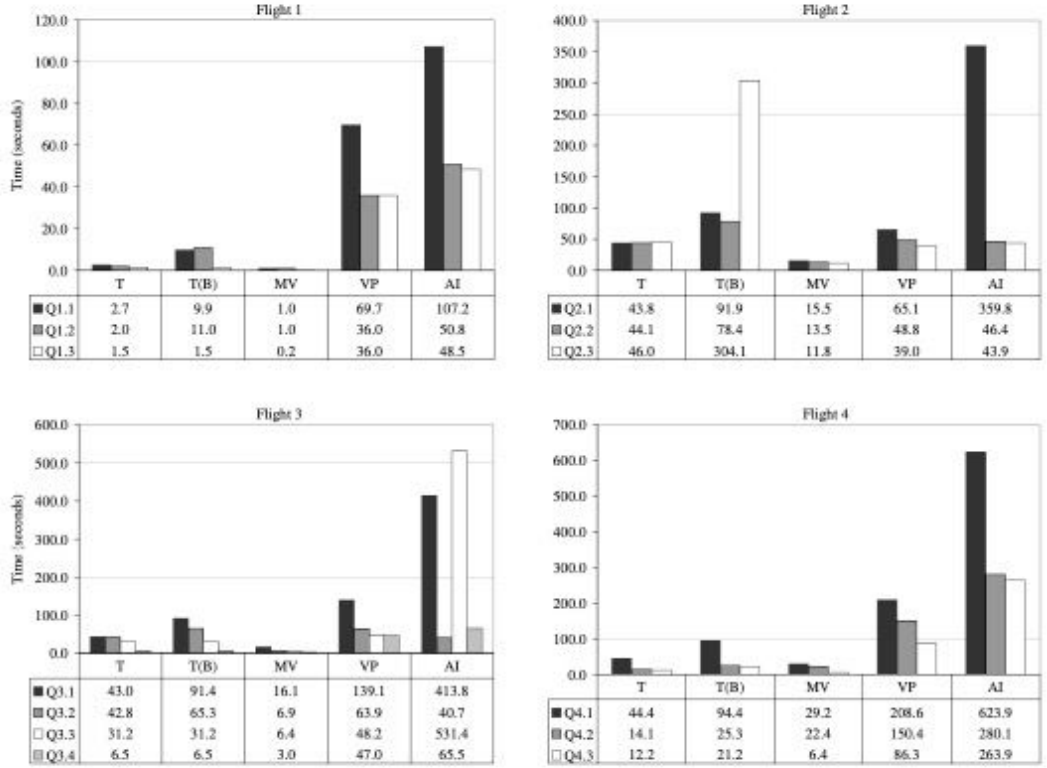
What is a sort-merge join?

A sort merge join is performed by sorting the data sets that are going to be joined, by a **join key** and then merging them together

What is more expensive here, sorting or merging?

Merging a cheap operation, but sorting can be expensive since data can be on the disk

Vertical Fragmentation



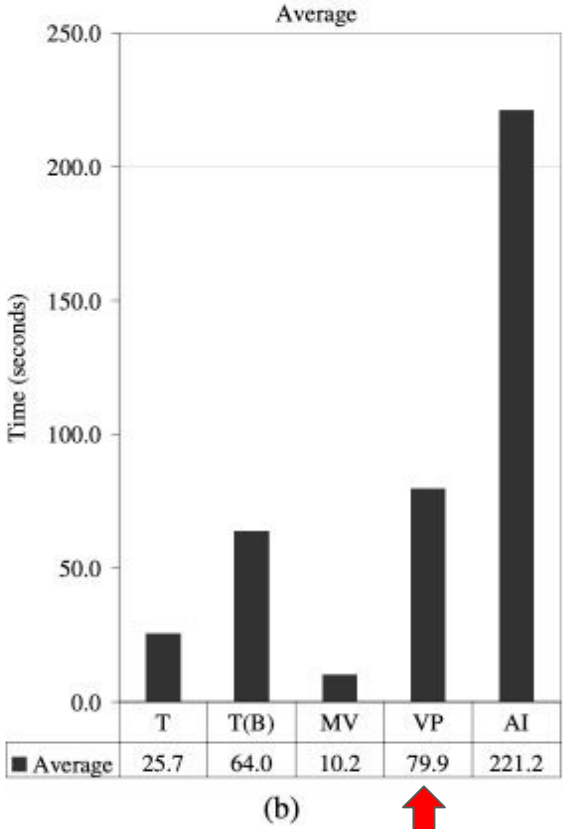


Figure 6: (a) Performance numbers for different variants of the row-store by query flight. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all queries.

Index Only Plans

What kind of data structure could we use?

B+ Trees (used by the paper)

B Trees

Hash Indexes

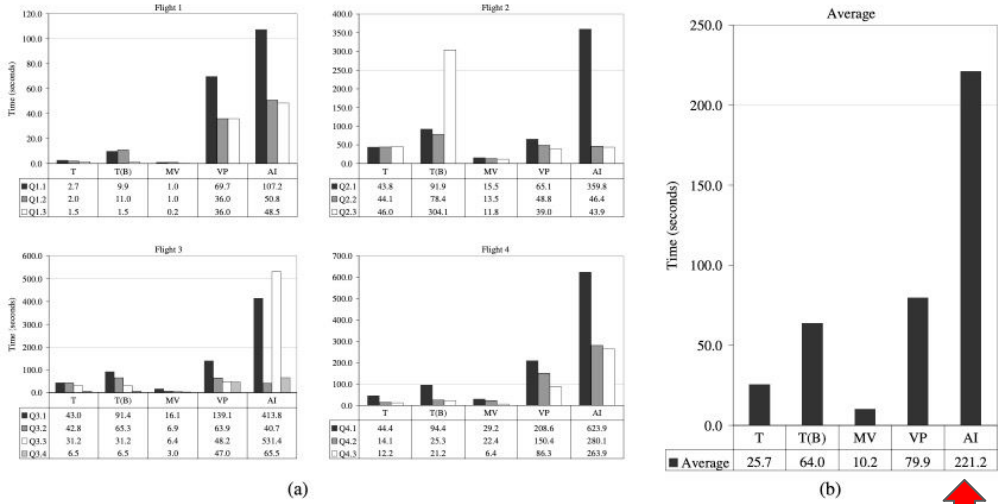


Figure 6: (a) Performance numbers for different variants of the row-store by query flight. Here, T is traditional, T(B) is t ditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all q ries.

Index Only Plans (cont.)

What are Index Only Plans?

- Emulating the column store index feature

How does this compare to vertical fragmentation?

+ Better in terms of space
+ Row stores have large headers so vertical partitioning waste memory
- Expensive Joins

Materialized Views

What is it?

Creating an optimal set of tables for every query flight in the workload

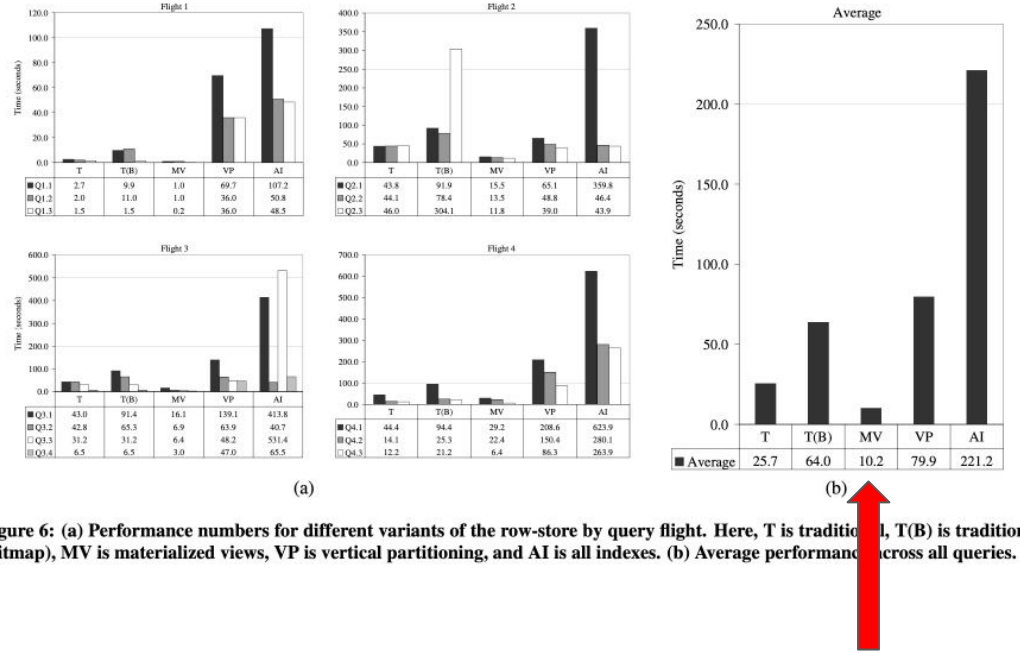


Figure 6: (a) Performance numbers for different variants of the row-store by query flight. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all queries.

Materialized Views (cont.)

What are some benefits of this?

- Quick query satisfaction

Limitations?

- You have to have a deep understanding of the data
- Think hard coding

Implementing Row Store as Column Store

Which is the best?

Tradiational!

Row Store is less efficient when we try to emulate column store

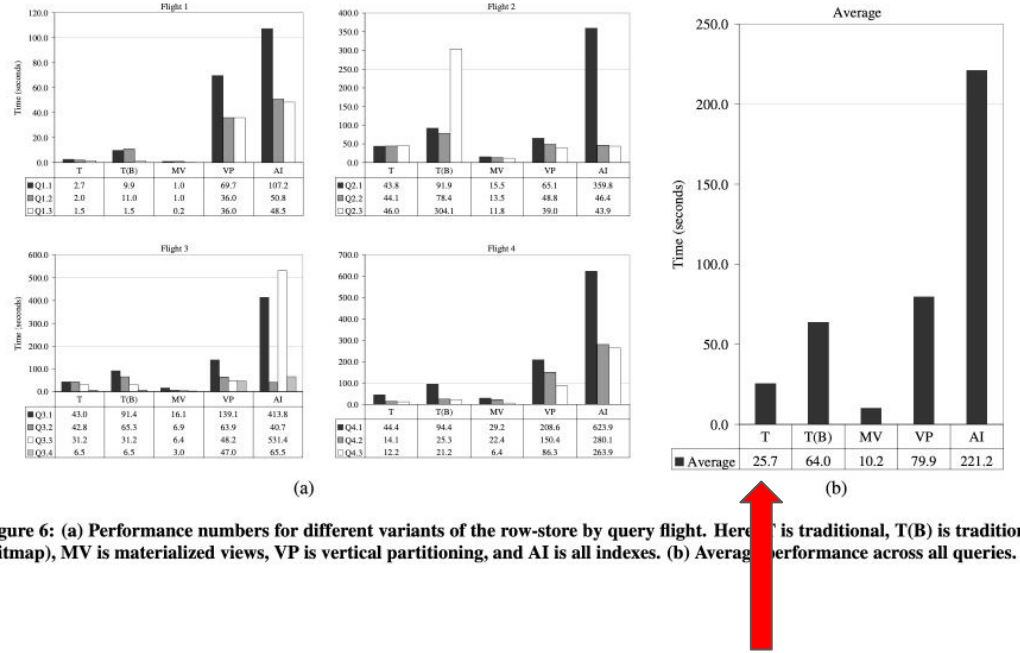


Figure 6: (a) Performance numbers for different variants of the row-store by query flight. Here T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all queries.

Column Stores

What are some column store optimizations that make it better than a row store?

- Compression
- Late Materialization
- Block Iteration
- Invisible Joins

Compression

What are ways for us to compress our data?

- Frequency
- Sorted

What are the benefits?

- Improved query performance
- Data stored in a column lends itself to being compressed, and is more compressible when the data is sorted

Compression (cont.)

What happens if we remove this optimization?

We lose our computation gained from skipping I/O

Late Materialization

What is it?

Late materialization is the process of waiting to construct the table of results until the end instead of reconstructing your tuples every time

What is tuple re-construction?

Rebuilding the tuple of qualifying data

LM (cont.)

Why is this effective for column stores?

+ Low tuple overhead
+ Stitch data together from column

Block Iteration

What is it?

Query for a block of the select statement, push your data to the next node to perform whatever work is left

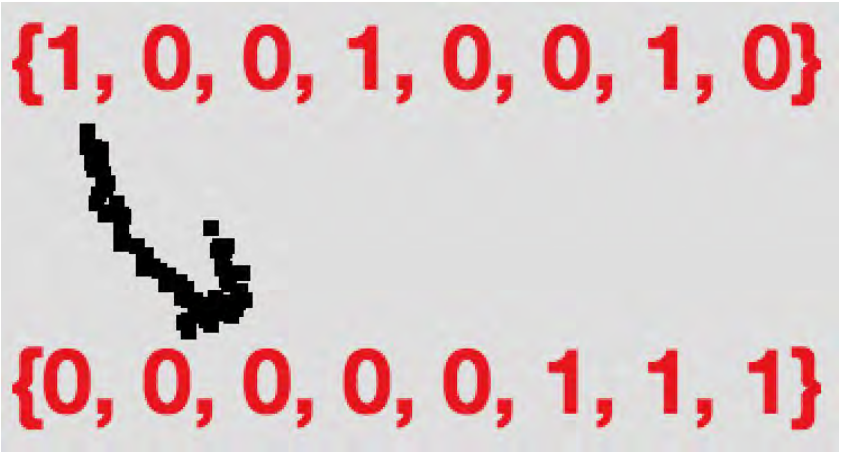
What is this similar to?

Pipelining in row-stores

Invisible Joins

What is it and how does it work?

- Optimize join predicates
- Bitmaps



What makes it effective for Star Schema?

- Joins with the fact table

Conclusion

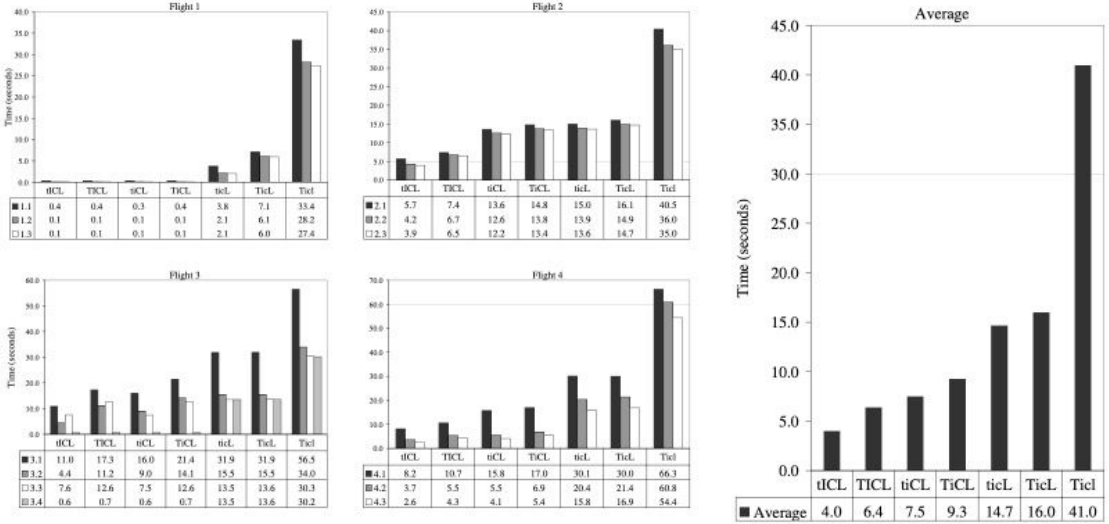


Figure 7: (a) Performance numbers for C-Store by query flight with various optimizations removed. The four letter code indicates the C-Store configuration: T=tuple-at-a-time processing, t=block processing; I=invisible join enabled, i=disabled; C=compression enabled, c=disabled; L=late materialization enabled, l=disabled. (b) Average performance numbers for C-Store across all queries.

Topic Review

Column Store and Row Stores

Vertical Partitioning

Index Only Plans

Materialized Views

Tuple Reconstruction

Late/Early Materialization

Block Iteration

Compression

Hash Joins

Index Joins

Sort-Merge Joins

Invisible Joins

Star Schema

Pros & Cons

Pros

- Dissecting the reasons for why column store is more efficient
- Exploring beyond just the topic of the paper (i.e. Invisible Joins)
- Good job in fully fleshing out their experiments

Pros & Cons

Cons

- Did not clearly define the difference between ‘Materialized Views’ and ‘Late Materialization’
- They chalked up a lot of performance problems to limitations in ability to tune their database
- Exploring the cost/benefit analysis of using two separate databases (one column, one row)

Final Thoughts

- This paper definitely does a good job around helping the reader understand the fundamental differences between column and row stores. As well as the certain optimizations that column stores have that row-stores cannot accomplish due to their underlying design.
- Overall this paper does a good job of highlighting the differences and helping us understand why we can’t emulate row-stores as column stores