



# *Column Stores vs Row Stores : How Different Are They Really*

*Daniel J. Abadi, Samuel R. Madden, Nabil Hachem*

Kameswara Venkatesh Emani  
Guide: Prof. S. Sudarshan

CS 632  
Course Seminar  
Department of CSE  
IIT Bombay

March 4, 2014



## Overview of the talk

---

- Introduction to Column Stores
- Emulating a Column Store In a Row Store
  - Vertical partitioning
  - Index only plans
  - Materialized views
- Optimizations for column stores
  - Compression
  - Late materialization
  - Block Iteration
  - Invisible Join
- Experiments
- Conclusions



## Section 1 - Introduction to Column Stores

---

Section content references: [2, 6, 5, 3]

Image references inline



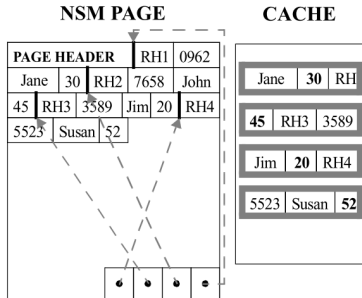
## Motivation

---

- Consider a table  $t$  which has 10 columns  $a, b, \dots, j$
- Query on  $t$ : `SELECT a FROM t WHERE b = X AND c = Y`
- Traditional database systems
  - store records contiguously with all fields from same record occurring together (N-ary storage model)
  - extract all tuples satisfying the predicates
  - return the column  $a$  from those tuples
- Since we need only the column  $a$  - reading/operating on other columns is unnecessary
- Can we improve on this?

## PAX - Partition Attributes Across

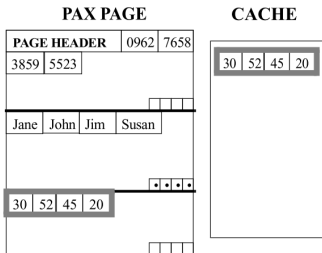
- Addresses a similar problem - poor cache performance due to irrelevant surrounding fields
  - Cache utilization and performance very important
  - In-page data placement key to high cache performance
  - N-ary storage model performs poorly
- Example below: assume cache block size is smaller than record size - leads to lot of cache misses



<sup>0</sup>Image reference: [6]

## PAX - Partition Attributes Across

- PAX solution - group together all values of a particular attribute within each page into a *mini-page*
- To reconstruct a record, perform a *mini-join* among the mini-pages within each page
- Exhibits superior cache performance
- However, does not reduce IO
- **Takeaway:** Organizing data by columns rather than by rows improves locality which is good for read intensive scenarios



<sup>0</sup>Image reference: [6]

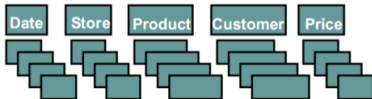
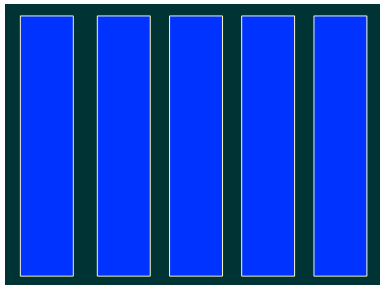


# What is a Column Store?

Row store



Column Store



<sup>0</sup>Image references: [3, 5]



## Column Store vs Row Store - Example

---

<b>EmpID</b>	<b>LastName</b>	<b>FirstName</b>	<b>Salary</b>
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

<b>Row Store</b>	<b>Column Store</b>
1,Smith,Joe,40000; 2,Jones,Mary,50000; 3,Johnson,Cathy,44000;	1,2,3; Smith,Jones,Johnson; Joe,Mary,Cathy; 40000,50000,44000;





## Why Column Stores?

---

- Row oriented databases are optimized for writes
  - Single disk write suffices to push all fields of a record onto disk
- Systems for ad-hoc querying should be **read optimized**
- Common example from data warehousing
  - Load new data in bulk periodically
  - Long period of ad-hoc querying
- For such read intensive workloads, column store architecture is more suitable
  - Fetch only required columns for query
  - Better cache effects
  - Better compression due to similar attributes within a column

We present the C-Store data model as a representative data model for column stores.

- Standard relational logical data model
- EMP(name, salary, age, dept)  
DEPT(dname, floor)
- Table - collection of projections
- Projection - set of columns, sorted

```
EMP1 (name, age | age)
EMP2 (dept, age, DEPT.floor | DEPT.floor)
EMP3 (name, salary | salary)
DEPT1 (dname, floor | floor)
```

- Horizontally partitioned into segments with segment identifier (sId)
- Column-to-row correspondence identified by position in the segment (storage key or sKey)

---

<sup>0</sup>Image reference: [2]



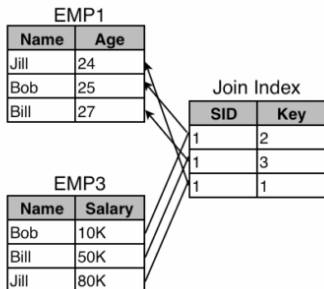
## Column Stores - Data Model

---

- How to put together various columns into a tuple?
  - Can use storage keys
  - What if data sorted on a different attribute?
  - Sorting again and merging is inefficient
- Solution : Use **Join indexes**
- Let T1 and T2 be two projections on table T
- M segments in T1, N segments in T2
- Join index from T1 to T2 contains M tables
  - Each row is of the form  
(s: SID in T2, k: Storage Key in Segment s)
  - In other words, for each sKey in a particular partition of T1, it specifies where (which partition, which sKey) the corresponding row is located in T2

### Join Indexes Example - Join Index from EMP3 to EMP1

```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```



<sup>0</sup>Image reference: [2]



## Compression

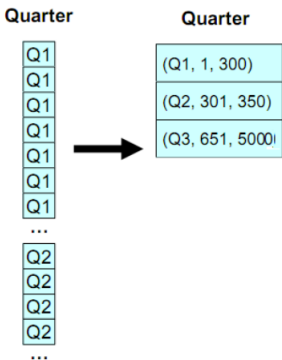
---

- Column-organized data can be compressed well due to good locality
- Trades IO for CPU
- Compression Schemes:
  - Run Length encoding
  - Dictionary encoding
  - Bit-vector encoding
  - Null suppression
  - Heavy-weight schemes
- Choice of compression depends on
  - Characteristics of data
  - Decompression trade-off (more compressed necessarily not the better)

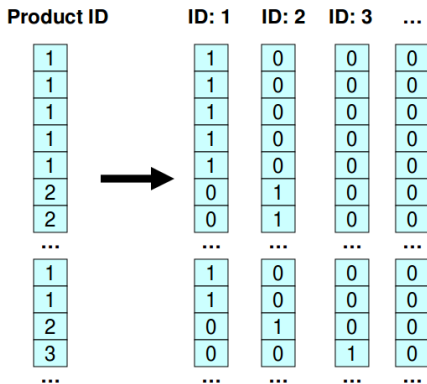


## Compression Examples

### Run Length encoding



### Bit-vector encoding



<sup>0</sup>Images from: [5]

- **Decompress**: Converts compressed columns into uncompressed representation
- **Select** : Same as relational select  $\sigma$  but produces bit-string
- **Mask** : Takes a bit-string B and a projection Cs, emits only those values from Cs whose corresponding bit in B is 1
- **Project** : Equivalent to relational project  $\pi$
- **Sort** : Sort a projection on some columns
- **Aggregation Operators** : SQL-like aggregates
- **Concat** : Combines two projections which are sorted in same order
- **Permute** : Re-order the projection according to a join index
- **Join** : Join two projections on a predicate
- **Bit-string operators** : BAnd (bitwise-AND), BOr (bitwise-OR), BNot (bitwise-NOT)



## Row vs Column - Summary of Differences

---

<b>Row Store</b>	<b>Column Store</b>
All fields in a record are stored contiguously	Each field is stored separately
Byte/word alignment of records	Dense packing
Compression/encoding discouraged	Make good use of compression
Needs to read all attributes to process any query	Only the necessary attributes can be read
Very good for OLTP queries	Not good for OLTP but performs much better for analytics workloads

And more ...





## The Questions

---

- Are row stores and column stores *really* different?
- We would like to answer the following questions
  - Is there a fundamental difference in architecture of column oriented databases?
  - Or can we use a more “column oriented” design in a traditional row store and achieve the same benefits?
  - What specific optimizations set column stores apart (on warehouse workloads)?



## Section 2 - Emulating a column store using a row store

---


Section content references: [1]

Image references inline

## Vertical Partitioning

- Partition each relation on all its columns
- How to match columns corresponding to the same row?
  - Store primary key - can be expensive
  - Use a 'position' attribute
  - Rewrite queries to join on position attribute to get multiple columns from same table

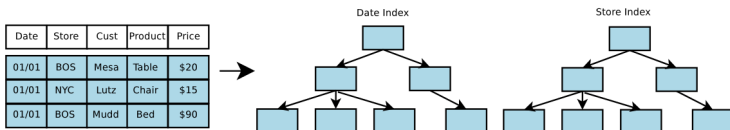
Date	Store	Cust	Product	Price
01/01	BOS	Mesa	Table	\$20
01/01	NYC	Lutz	Chair	\$15
01/01	BOS	Mudd	Bed	\$90



Date		Store		Cust		Product		Price	
POS	VALUE	POS	VALUE	POS	VALUE	POS	VALUE	POS	VALUE
1	01/01	1	BOS	1	Mesa	1	Table	1	\$20
2	01/01	2	NYC	2	Lutz	2	Chair	2	\$15
3	01/01	3	BOS	3	Mudd	3	Bed	3	\$90

<sup>0</sup>Image from [5]

- Disadvantages of Vertical Partitioning
  - Need to store position with every column
  - Large header for each tuple
  - Leads to space wastage
- Alternative- Use indexes

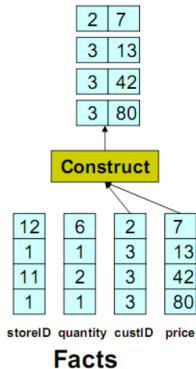


- Base relation stores using standard relational design
- B+ tree index on every column of table
- Tuple header not stored - so overhead is less
- No need to access actual tuples on disk
  - Based on the query predicate, index is consulted and (*record\_id*, *column\_value*) pairs are returned
  - These are then merged in memory

- Base relation stores using standard relational design
- B+ tree index on every column of table
- Tuple header not stored - so overhead is less
- No need to access actual tuples on disk
  - Based on the query predicate, index is consulted and  $(record\_id, column\_value)$  pairs are returned
  - These are then merged in memory
- Disadvantages
  - If there is no predicate on the column, requires full scan of the index to read all tuples
  - Example: `SELECT AVG(salary) FROM EMP WHERE age > 40`
  - With separate indexes, first find record\_id's satisfying predicate on age
  - Then join this with full column set for salary
  - Can answer directly if there is an index on  $(age, salary)$

- Idea: Access only the required data from disk
- Create views with only those columns that are necessary
- No pre-joins done
- Workload needs to be known in advance - limited applicability

SELECT F.custId FROM fact AS F  
WHERE F.price > 20





## Section 3 - Optimizations for Column Stores

---

Section content references: [1]

Image references inline





### Recap

- Run Length encoding
- Dictionary encoding
- Bit-vector encoding
- Null suppression
- Heavy-weight schemes



## *Late Materialization*

---

- Realization: We may use column store, but ultimately, we need to construct tuples while returning results
  - Standard API's - JDBC, ODBC - rely on this model
- Naive column store implementation
  - Store data as columns on disk
  - Retrieve columns, stitch them into rows
  - Operate using row-oriented operators
  - 'Early Materialization' - does not make full use of column oriented design

- Use 'late materialization' instead
  - Construct tuples as late as possible
- Example: Remember our first query?  
SELECT a FROM t WHERE  $b = X$  AND  $c = Y$
- Early materialization would operate on many columns - unnecessary
- Late materialization
  - Get record id's (as bit string) which pass the predicate  $b = X$
  - Get record id's (as bit string) which pass the predicate  $c = Y$
  - Perform bit-wise AND to get result bit-string
  - Get corresponding values (1's in the result) from column  $a$

- Use 'late materialization' instead
  - Construct tuples as late as possible
- Example: Remember our first query?  
SELECT a FROM t WHERE  $b = X$  AND  $c = Y$
- Early materialization would operate on many columns - unnecessary
- Late materialization
  - Get record id's (as bit string) which pass the predicate  $b = X$
  - Get record id's (as bit string) which pass the predicate  $c = Y$
  - Perform bit-wise AND to get result bit-string
  - Get corresponding values (1's in the result) from column a
- Advantages
  - Constructs only those tuples that are necessary since many tuples filtered by each predicate. Also, only the required columns used
  - Can use operators to operate on compressed data for longer duration - hence better performance
  - Cache performance better with column oriented data - not polluted with surrounding irrelevant attributes (PAX)
  - Block iteration performs better for fixed length attributes



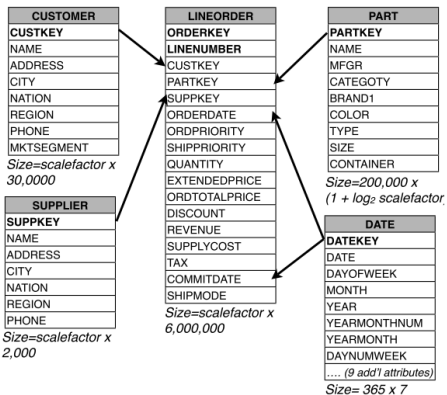
## *Block Iteration*

---

- Traditional row oriented systems incur per-tuple processing overhead
  - 1-2 function calls for each tuple to get needed data
- Overhead lesser when dealing with blocks of tuples at once
- Applies naturally to column oriented databases
  - Most column stores send blocks of values in single function call
  - Columns tend to be fixed-width — can exploit array referencing and parallelism

# Star Schema

- **Fact table** - many columns; typically some data values and many foreign keys to dimension tables
- **Dimension tables** - small tables, a few columns



<sup>0</sup> Image reference: [1]

### Motivation

- Commonly encountered query pattern in star schema
  - Restrict tuples from fact table using predicate on dimension table
  - Aggregate on fact table, grouping by columns in dimension table

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey
      AND lo.supkey = s.supkey
      AND lo.orderdate = d.datekey
      AND c.region = ASIA
      AND s.region = ASIA
      AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

---

<sup>0</sup>Image reference: [1]



## Traditional Approach

---

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, ddate AS d
WHERE lo.custkey = c.custkey
      AND lo.suppkey = s.suppkey
      AND lo.orderdate = d.datekey
      AND c.region = ASIA
      AND s.region = ASIA
      AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

- Pipelining based on selectivity
- Join *lineorder* and *customer* and filter *lineorder* tuples using *c.region*
- Append *c.nation* to *lineorder* during the join
- Pipeline to join on *supplier* and filter using predicate, add required fields
- Pipeline to join on *ddate* and filter using predicate, add required fields
- Perform aggregation

---

<sup>0</sup>Image reference: [1]





## Late Materialization Approach

---

- Apply predicate on *customer* table and obtain matching customer keys
- Join with customer key from *lineorder*
  - Results in two sets of positions indicating the tuples matched for join
  - Only one set of positions (typically those of fact table) sorted
  - Customer key positions out of order

$$\begin{array}{|c|} \hline 42 \\ \hline 36 \\ \hline 42 \\ \hline 44 \\ \hline 38 \\ \hline \end{array} \bowtie \begin{array}{|c|} \hline 38 \\ \hline 42 \\ \hline 46 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 2 \\ \hline 5 & 1 \\ \hline \end{array}$$

- Extract required fields from *customer* table based on key
- Similar join with other tables

Out of order retrieval can have significant cost

- Minimize out of order extractions using **Invisible Join**

---

<sup>0</sup>Image reference: [4]

### Phase 1

- Rewrite joins as selection predicates on fact table columns
- Apply predicate to dimension tables to obtain list of satisfying keys
- Hash the (potentially small) dimension table

### Phase 2

- Check fact table foreign key against this hash and construct satisfying bitmap for fact table
- After doing this for each such dimension table, take AND of bitmaps to extract final list of satisfying fact table keys

### Phase 3

- Now, use the foreign keys and hashed dimension tables to extract the required column values
- Number of values to be extracted is minimized because selectivity for entire query has been used
- If the dimension table keys are sorted and contiguous starting from 1, foreign key is just an array index. So lookup in the hashed table is extremely fast.

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, ddate AS d
WHERE lo.custkey = c.custkey
      AND lo.suppkey = s.suppkey
      AND lo.orderdate = d.datekey
      AND c.region = ASIA
      AND s.region = ASIA
      AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

## Phase 1

Apply region = 'Asia' on Customer table

custkey	region	nation	...
1	Asia	China	...
2	Europe	France	...
3	Asia	India	...

Hash table  
with keys  
1 and 3

Apply region = 'Asia' on Supplier table

suppkey	region	nation	...
1	Asia	Russia	...
2	Europe	Spain	...

Hash table  
with key 1

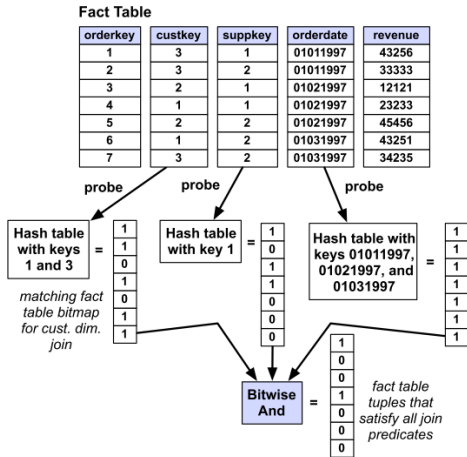
Apply year in [1992,1997] on Date table

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

Hash table with  
keys 01011997,  
01021997, and  
01031997

# Invisible Join

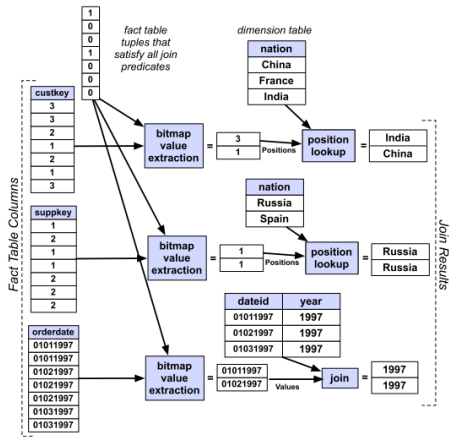
## Phase 2



<sup>0</sup>Image reference: [1]

# Invisible Join

## Phase 3



<sup>0</sup>Image reference: [1]

## Between predicate rewriting

- Remember we expressed join as a selection predicate on fact table?
- Very useful if resultant dimension table after applying predicate consists of contiguous keys (in other words, it represents a **range** of keys)
- If so, rewrite fact table predicate as a “between” predicate instead of “equal” predicate

Apply “region = ‘Asia’” On Customer Table

custkey	region	nation	...
1	ASIA	CHINA	...
2	ASIA	INDIA	...
3	ASIA	INDIA	...
4	EUROPE	FRANCE	...



~~Hash Table (or Bit-map)  
Containing Keys 1, 2 and 3~~

**Range [1-3]**

- Much simpler to check — no lookup, significant improvement

<sup>0</sup>Image reference: [5]

## Between predicate rewriting

- Remember we expressed join as a selection predicate on fact table?
- Very useful if resultant dimension table after applying predicate consists of contiguous keys (in other words, it represents a **range** of keys)
  - Data often sorted by increasingly finer granularity - continent then country then city, year then month then date etc.
  - Equality on any of the sorted columns results in a range
- If so, rewrite fact table predicate as a “between” predicate instead of “equal” predicate

Apply “region = ‘Asia’” On Customer Table

custkey	region	nation	...
1	ASIA	CHINA	...
2	ASIA	INDIA	...
3	ASIA	INDIA	...
4	EUROPE	FRANCE	...



~~Hash Table (or Bit-map)  
Containing Keys 1, 2 and 3~~

**Range [1-3]**

- Much simpler to check — no lookup, significant improvement

<sup>0</sup>Image reference: [5]



## Section 4 - Experiments

---

Section content references: [1]

Image references inline





**System X** : a traditional row store

**C-Store** : a column store

### Goals

- Compare performance of C-Store vs column store emulation on System X
- Identify which optimizations in C-store are most significant
- Infer from results if it is possible to successfully emulate a column store using a row store
  - What guidelines should one follow?
  - Which performance optimizations will be most fruitful?



### Machine

- 2.8 GHz single processor, dual core Pentium(R) D workstation
- 3 GB RAM
- Red Hat Enterprise Linux 5
- 4-disk array, managed as a single logical volume
- Reported numbers are average of several runs
  - Also, a “warm” buffer pool - 30% improvement for both systems

### Data

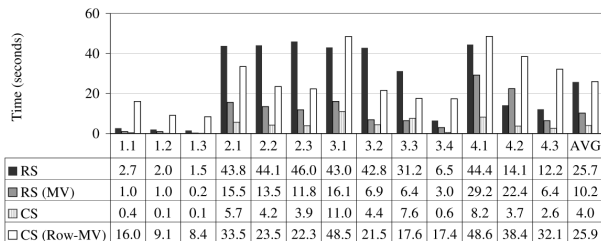
- Star Schema Benchmark (SSBM) - derived from TPCH
- Fact table: 17 columns, 60,000,000 rows
  - Table : LineOrder
- 4 dimension tables: largest - 80,000 rows
  - Tables: Customer, Supplier, Part, Date



### **Workload**

- 13 queries divided into four categories or “flights”
- Data warehousing queries
- Flight 1: Restriction on 1 dimension attribute + columns on the fact table
- Flight 2: Restriction on 2 dimension attributes
- Flight 3, 4: Restriction on 3 dimensions

## Baseline and Materialized View



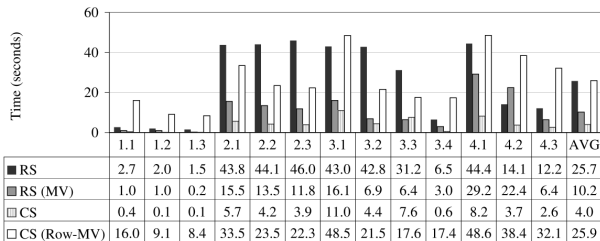
RS: Row store, RS(MV): Row Store with optimal set of materialized views, CS: column store, CS(Row-MV): Column store constructed from RS(MV)

- C-Store outperforms System X
- Factor of 6 in base case (CS vs RS)
- Factor of 3 with MV on System X (CS vs RS(MV))

Expected on warehouse workloads

<sup>0</sup>Image reference: [1]

## Baseline and Materialized View



RS: Row store, RS(MV): Row Store with optimal set of materialized views, CS: column store, CS(Row-MV): Column store constructed from RS(MV)

- CS(Row-MV) vs RS(MV)
  - Expected to be comparable
- System X outperforms by a factor of 2
- System X more fine tuned with advanced performance features
- Not a level ground for comparison

<sup>0</sup>Image reference: [1]



## Need for a fair comparison

■ RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
■ RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
□ CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
□ CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

RS: Row store, RS(MV): Row Store with optimal set of materialized views, CS: column store, CS(Row-MV): Column store constructed from RS(MV)

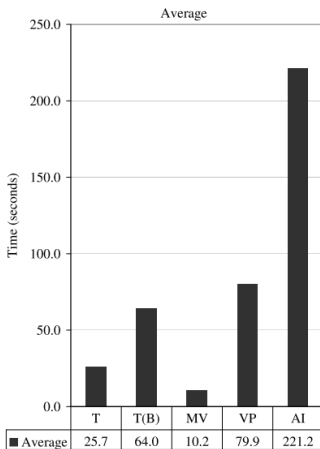
- Hidden factors might affect results when comparing two different systems
- Solution: Take one system at a time, and modify it
  - Simulate column store inside System X
  - Remove performance optimizations from C-Store until row store performance is achieved
- Inferences will be more reliable
- Example
  - CS vs CS(Row-MV) - factor of 6 difference
  - Although both read minimal set of columns
  - Thus, less IO not the only factor

<sup>0</sup>Image reference: [1]



### Configurations of System X used

- Traditional (T)
- Traditional (bitmap): biased to use bitmaps; might be inferior sometimes (T(B))
- Vertical Partitioning: Each column is a relation (VP)
- Index-Only: B+Tree on each column (AI)
- Materialized Views: Optimal set of views for every query (MV)



<sup>0</sup>Image reference: [1]

- Materialized views performed the best
- Index-only plans the worst
  - Expensive hash joins on fact table before it is filtered (to join columns)
  - System X cannot retain fact table record id's after joining with another table
- Vertical Partitioning
  - Comparable to MV when only a few columns were used
  - Tuple overheads affected performance significantly when more than 1/4th of the columns used
  - Scanning four columns in vertical partitioning approach took as long as scanning the entire table in traditional approach
  - 960 MB per column (vertical partitioning) vs 240 MB per column (C-store)



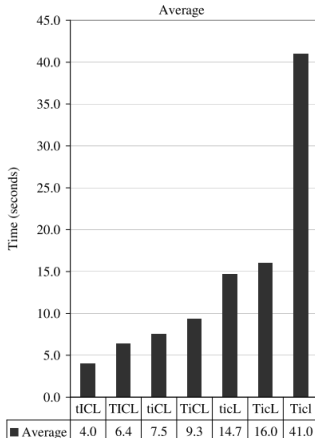


### Approach

- Start with column store
- Remove each optimization

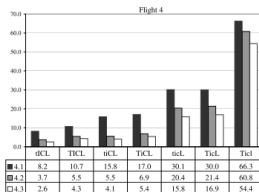
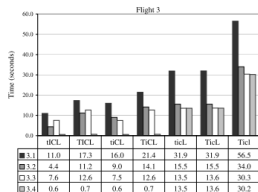
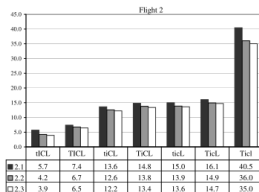
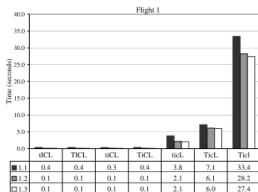
### Configuration

- 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



<sup>0</sup>Image reference: [1]

# Column Store Performance - By Flight



T=tuple-at-a-time processing, t=block processing; l=invisible join enabled, i=disabled; C=compression enabled, c=disabled; L=late materialization enabled, l=disabled

0 Image reference: [1]

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

T=tuple-at-a-time processing, t=block processing; I=invisible join enabled, i=disabled; C=compression enabled, c=disabled; L=late materialization enabled, l=disabled

### Analysis

- Late materialization - factor of 3 (most important)
- Block processing - 5% to 50% depending on whether compression has been removed
- Invisible joins - 50% to 75% (largely due to between-predicate rewriting)
- Compression - factor of 2
- Tuple construction is costly - adds a factor of almost 2

---

<sup>0</sup>Image reference: [1]



## Section 5 - Conclusions

---

Section content references: [1]

Image references inline



## Conclusions

---

- Column stores perform better than row stores for warehouse workloads
  - Various optimizations in column stores contribute to improved performance
- Column stores and row stores employ different design decisions
  - No fundamental hindrances for row stores to adopt some of the techniques from column stores
  - Example: Store tuple headers separately, use virtual record id's to join data etc.

Contd . . .



## Conclusions

---

- Emulating a column store inside row stores performs poorly
  - Tuple reconstruction costs
  - Per tuple overheads
- Some important system improvements necessary for row stores to successfully emulate column stores
- Can we build a complete row store that can transform into column store for warehouse workloads?
  - SAP HANA has a solution for this <sup>1</sup>

---

<sup>1</sup>Interested readers refer: <http://dl.acm.org/citation.cfm?doid=2213836.2213946> (also on course website)



## References

---



Daniel J. Abadi et. al, *Column-Stores vs. Row-Stores: How Different Are They Really?* SIGMOD, 2008.



Mike Stonebraker et. al, *C-Store: A Column-oriented DBMS* VLDB, 2005.



<http://www.vldb2005.org/program/slides/thu/s553-stonebraker.ppt>



Daniel J. Abadi et. al, *Materialization Strategies in a Column-Oriented DBMS*, ICDE, 2007.



Stavros Harizopoulos et. al, *Column-Oriented Database Systems*, VLDB, 2009.



Anastassia Ailamaki et. al, *Weaving Relations for Cache Performance*, VLDB, 2001.



Talk by Karthik S. R, <http://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/Talks/columnstore-karthik.odp>



Talk by Subhro Bhattacharyya and Souvik Pal,

[http://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/Talks/C-Store\\_Presentation.pdf](http://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/Talks/C-Store_Presentation.pdf)



Talk by Paresh Modak and Souman Mandal,

<http://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/Talks/Column-vs-Row.ppt>



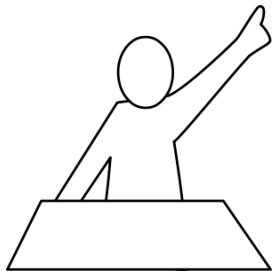
<http://openclipart.org/detail/123337/question-by-1mproulx>



*Thank You!*

---

Questions?



---

<sup>0</sup>Image reference: [10]