Column-Stores vs. Row-Stores How Different are they Really?

Daniel J. Abadi, Samuel Madden and Nabil Hachem, SIGMOD 2008

Presented by: Mehvish Saleem

Contents

- Structural difference between column and row store
- Contributions
  - Emulation of column-store in row store
  - Column-oriented optimizations esp. Invisible join
  - Column-store performance contributors
- Experiments and Results
- Conclusion

What is Column-Store?

- Row-store stores data in the disk tuple by tuple
- Column-store stores data in the disk column by column

Emulation of column-store in row-store

- Vertical partitioning
- Using index-only plans
- Using materialized views

Vertical Partitioning

- Full Vertical partitioning of each relation
  - Each column = 1 Physical table
  - Can be achieved by adding integer position column to every table
- Join on position for multi-column fetch

- Problems:
  - "Position" - Space and disk bandwidth
  - Header for every tuple – further space wastage

Index-only plans

- Add B+Tree index for every Table column
- Plans never access the actual tuples on disk
- Headers are not stored, so per tuple overhead is less

- Problems:
  - Separate indices may require full index scan, which is slower
    - E.g.: SELECT AVG(salary) FROM emp WHERE age > 40

Materialized Views

- Create ‘optimal' set of MVs for given query workload
- Objective:
  - Provide just the required data
  - Avoid overheads
  - Performs better
- Expected to perform better than other two approach

- Problems:
  - Require knowledge of query workloads in advance

Column-oriented execution

- Compression
- Late Materialization
- Block Iteration
- Invisible join

Compression

- Perform better on data with low information entropy
- Column-store data super compressible
- Improves performance
  - Less space
  - Less time spent in I/O

Late Materialization

- Delay Tuple Construction
- Might avoid constructing it altogether
- Eg: SELECT R.a FROM R WHERE R.c = 5 AND R.b = 10
  - Output of each predicate is a bit string
  - Perform Bitwise AND
  - Use final position list to extract R.a

Block Iteration

- Iterate over blocks of tuples rather than tuple-at-a-time
- Like batch processing
- If column is fixed width, it can be operated as an array
- Exploits potential for parallelism

Invisible Join

Star Schema Benchmark

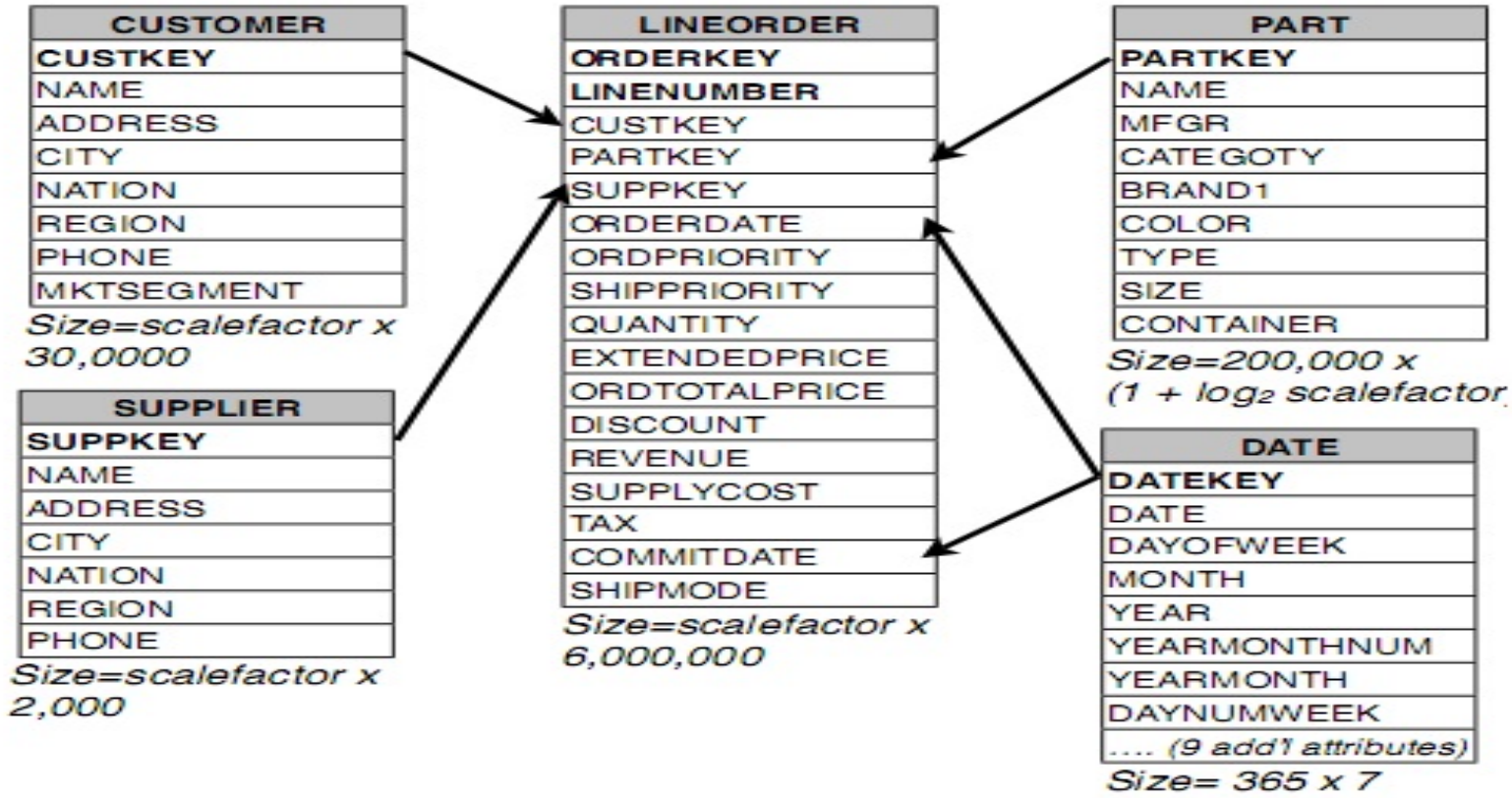


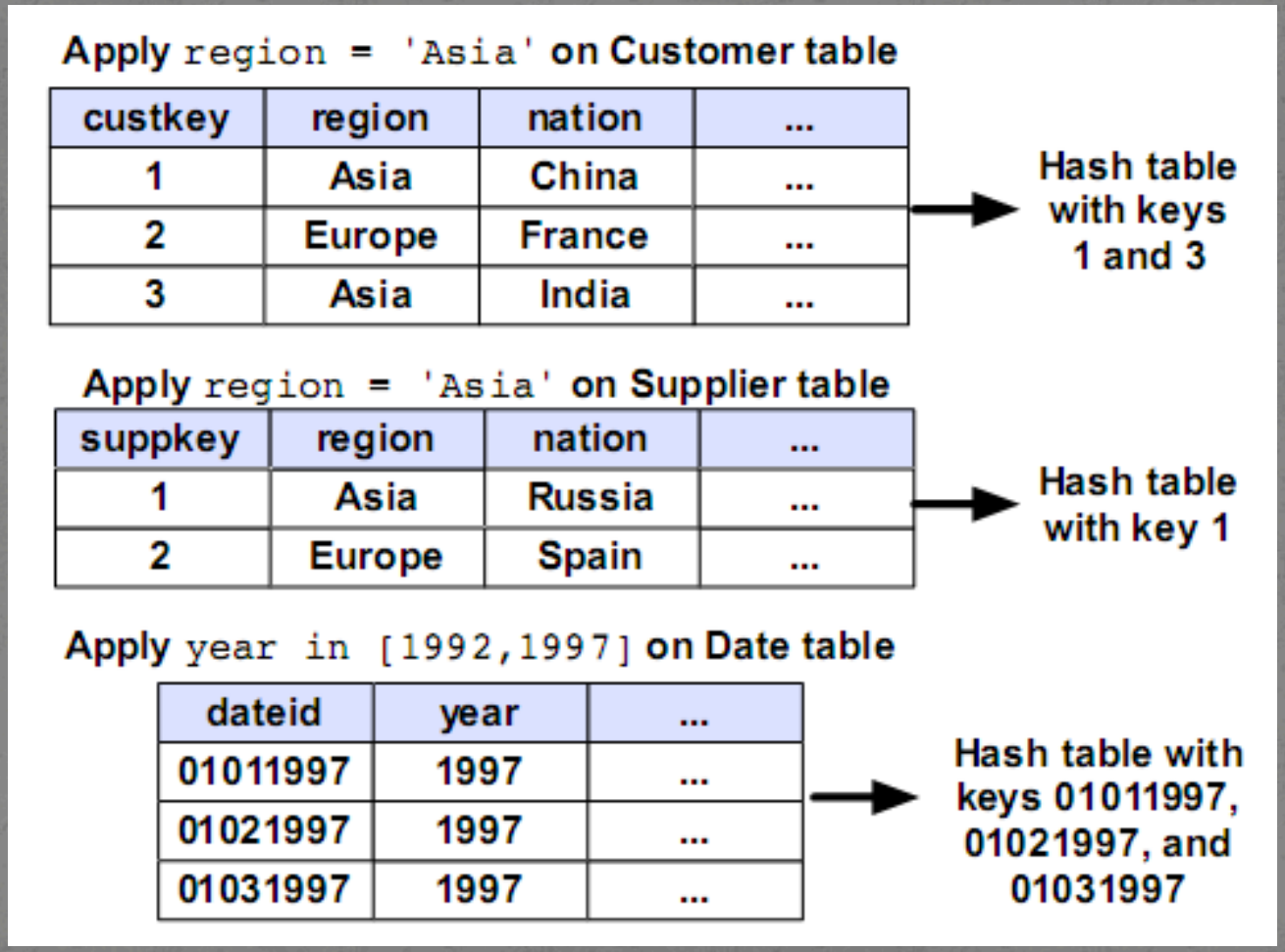
Figure 1: Schema of the SSBM Benchmark

Sample Query

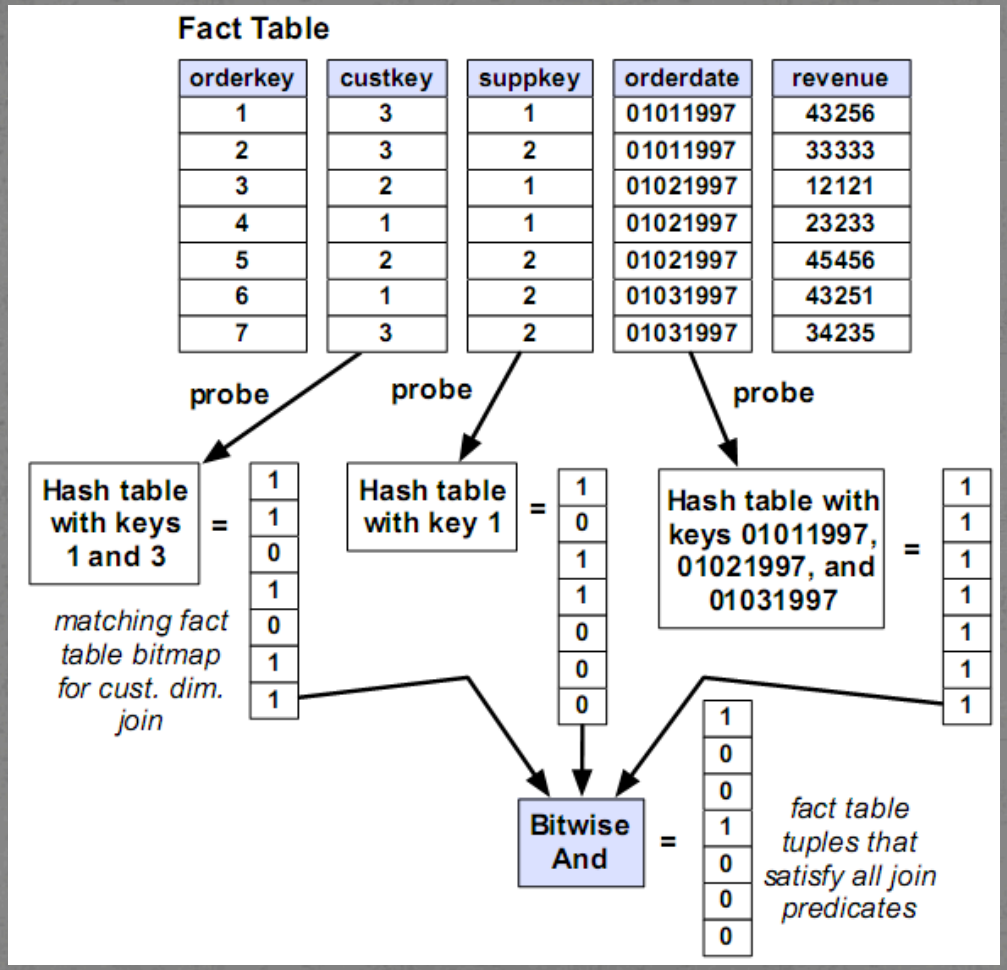
```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey
  AND lo.suppkey = s.suppkey
  AND lo.orderdate = d.datekey
  AND c.region =  ASIA
  AND s.region =  ASIA
  AND d.year >= 1992 and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

Find total revenue from Asian customers who purchase a product supplied by an Asian supplier between 1992 and 1997 grouped by nation of the customer, supplier and year of transaction

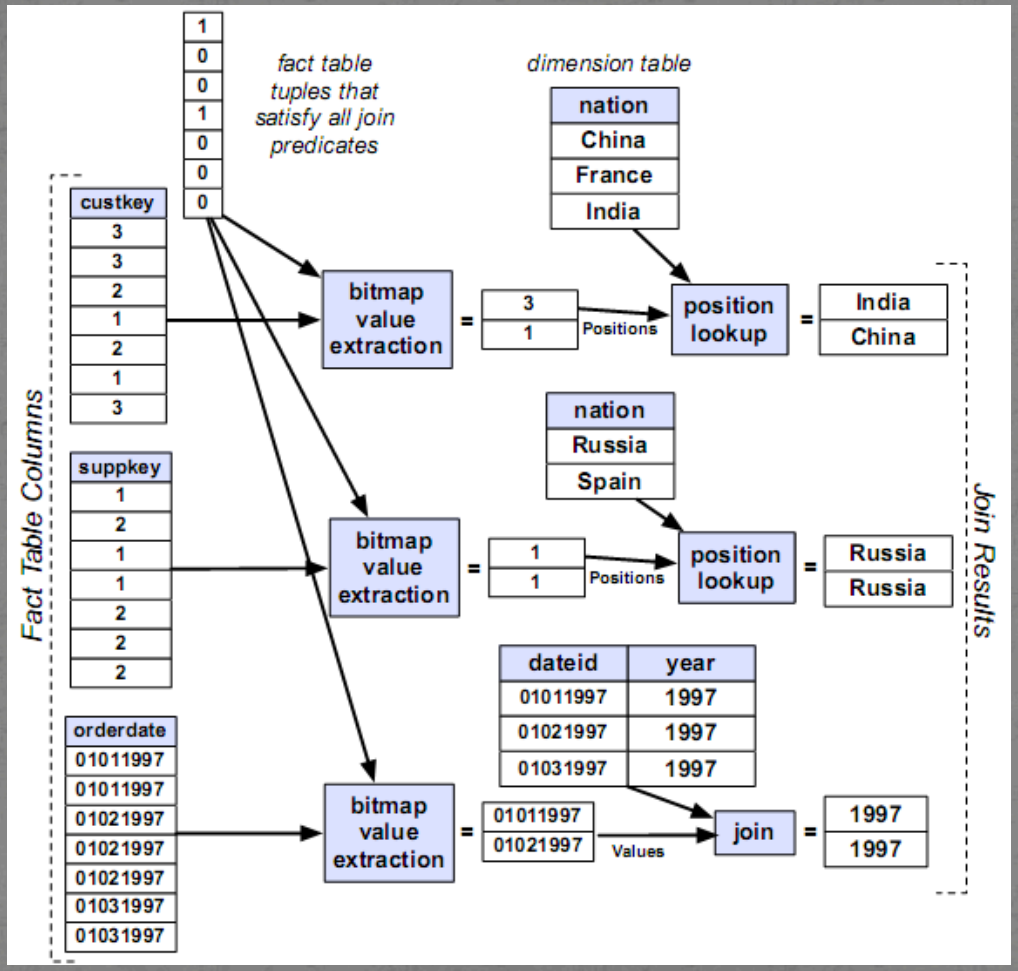
Phase I


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

Phase II


Fact Table
orderkey
custkey
suppkey
orderdate
revenue
1 3 1 01011997 43256
2 3 2 01011997 33333
3 2 1 01021997 12121
4 1 1 01021997 23233
5 2 2 01021997 45456
6 1 2 01031997 43251
7 3 2 01031997 34235
probe
probe
probe
Hash table with keys 1 and 3
=
1 1 0 1 0 1 1
matching fact table bitmap for cust. dim. join
Hash table with key 1
=
1 0 1 1 0 0 0
Hash table with keys 01011997, 01021997, and 01031997
=
1 1 1 1 1 1 1
Bitwise And
=
1 0 0 1 0 0 0
fact table tuples that satisfy all join predicates

Phase III

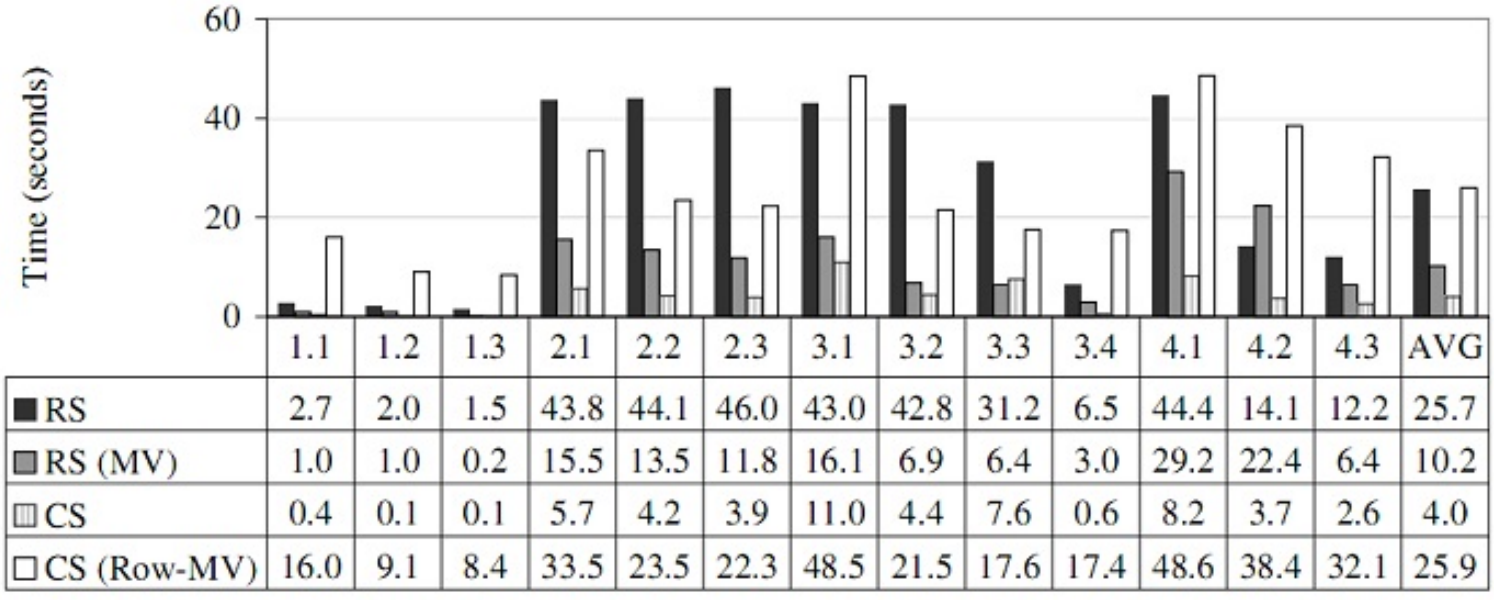

1
0
0
1
0
0
0
fact table tuples that satisfy all join predicates
dimension table
nation
China
France
India
Fact Table Columns
custkey
3
3
2
1
2
1
3
bitmap value extraction
=
3
1
Positions
position lookup
=
India
China
nation
Russia
Spain
suppkey
1
2
1
1
2
2
2
bitmap value extraction
=
1
1
Positions
position lookup
=
Russia
Russia
dateid
year
01011997
1997
01021997
1997
01031997
1997
orderdate
01011997
01011997
01021997
01021997
01021997
01031997
01031997
bitmap value extraction
=
01011997
01021997
Values
join
=
1997
1997
Join Results

Experiments

Purpose

- Comparing the performance of C-Store with column-store emulation in row-store
- Most significant optimization for column-stores
- Can unmodified row store obtain benefits of column-store?

C-Store VS System X

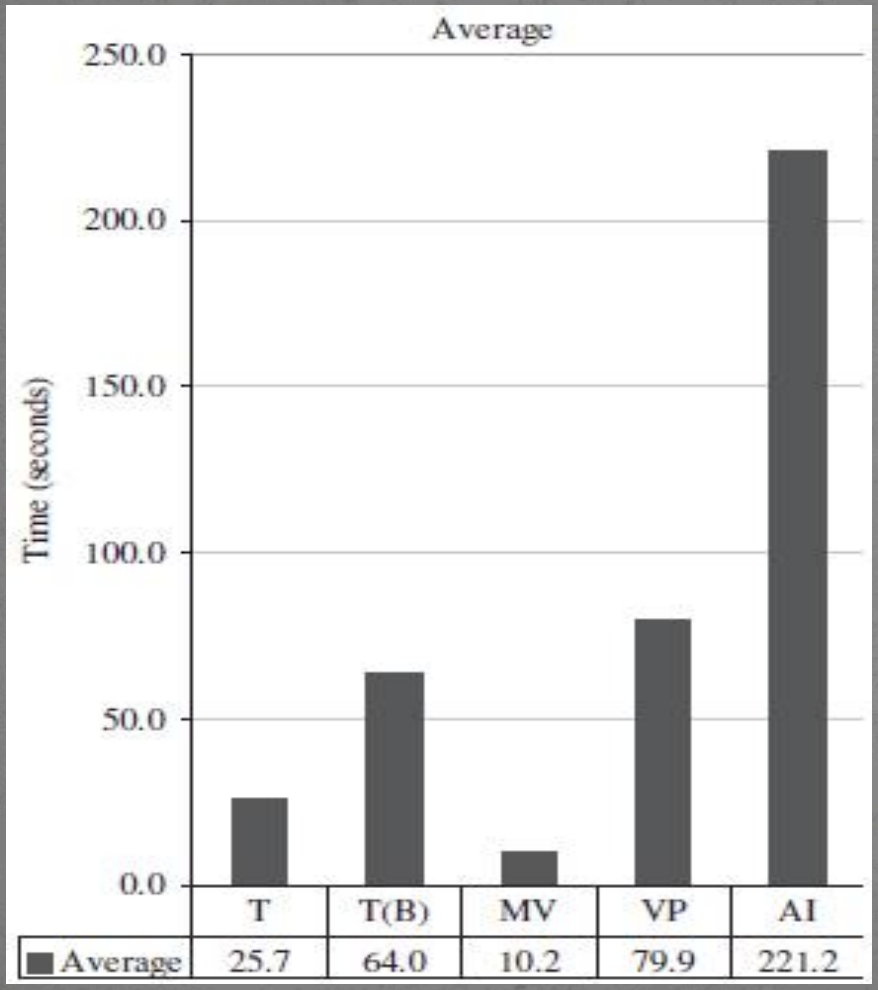


	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	AVG
RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

Baseline performance of C-Store "CS" and System X "RS", compared with materialized view cases on the same systems.

- RS: Base System X
- CS: Base C-Store
- RS (MV): System X with optimal collection of MVs
- CS (Row-MV): Column store constructed from RS(MV)

Different System X Configurations



- T: Traditional
- T(B): Traditional Bitmap
- MV: Materialized View
- VP: Vertical Partitioning
- AI: All Indexes

Results and Analysis

- MV performs best since they read minimal amount of data needed by a query
- Index only plans are the worst:
  - Expensive column joins on fact table
- Vertical partitioning:
  - Tuple overheads and reconstruction
- Column Store perform better than the best case of row store (4.0 sec VS 10.2 sec)

Conclusion

- Emulation of a column-store in a row-store does not yield good results
  - Tuple reconstruction costs
  - A pain to implement
- Most important optimizations for column-store: Compression and Late Materialization

Thank you!