Column-Stores vs. Row-Stores

How Different are they Really?

Arul Bharathi

Authors

Daniel J.Abadi

Samuel R. Madden

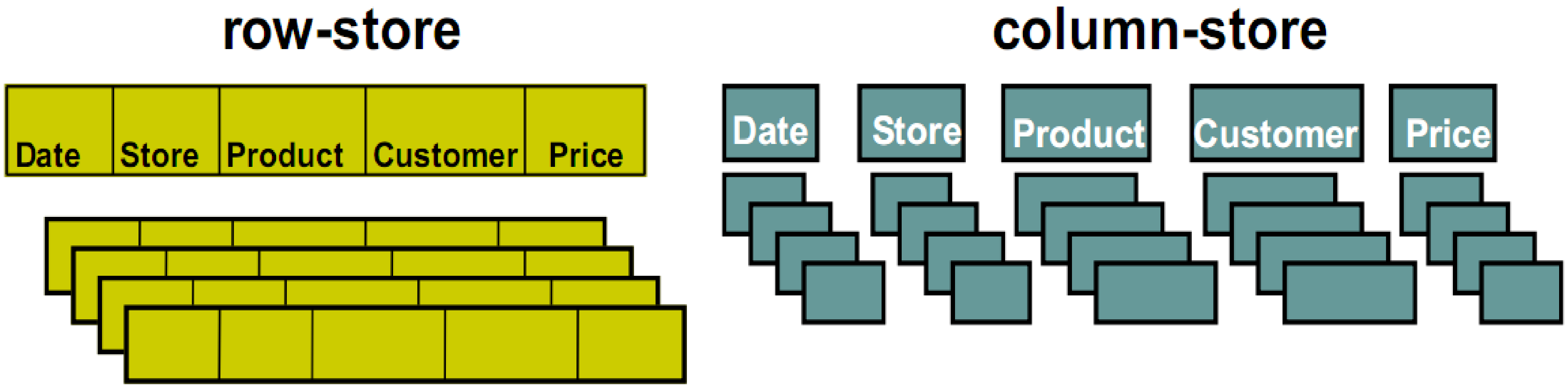
Nabil Hachem

Contents

- Introduction
- Row Oriented Execution
- Column Oriented Execution
- Column-Store Simulation in a Row-Store
- Column-Store Performance
- Conclusion

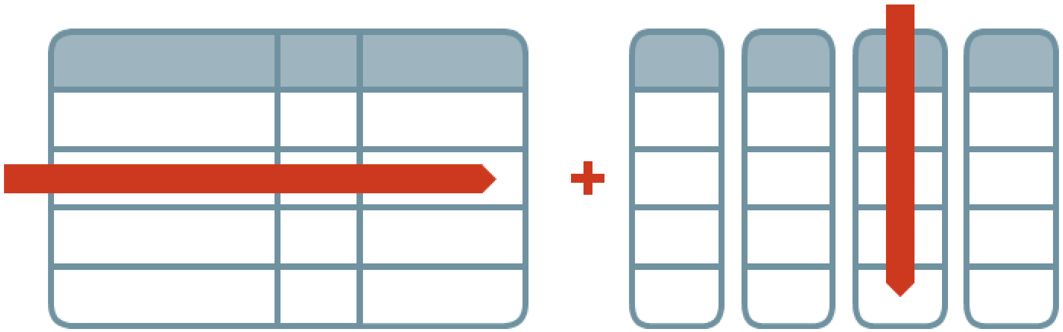
1.
Introduction

Row-Store & Column-Store



- In row-store the data is stored in form of tuples
- In column-store, the data is stored in memory by columns
- Column-store are more I/O efficient for read-only
- There are assumptions that column-store concept can be emulated in row-store

Questions addressed by this paper



- Are the performance gain due to something fundamental about the way column-oriented DBMSs are internally architecture, or would such gains also be possible in a conventional system that used a more column-oriented physical design?
- Which of the many column-database specific optimizations proposed in the literature are most responsible for the significant performance advantage of column-stores over row-stores on warehouse workloads?

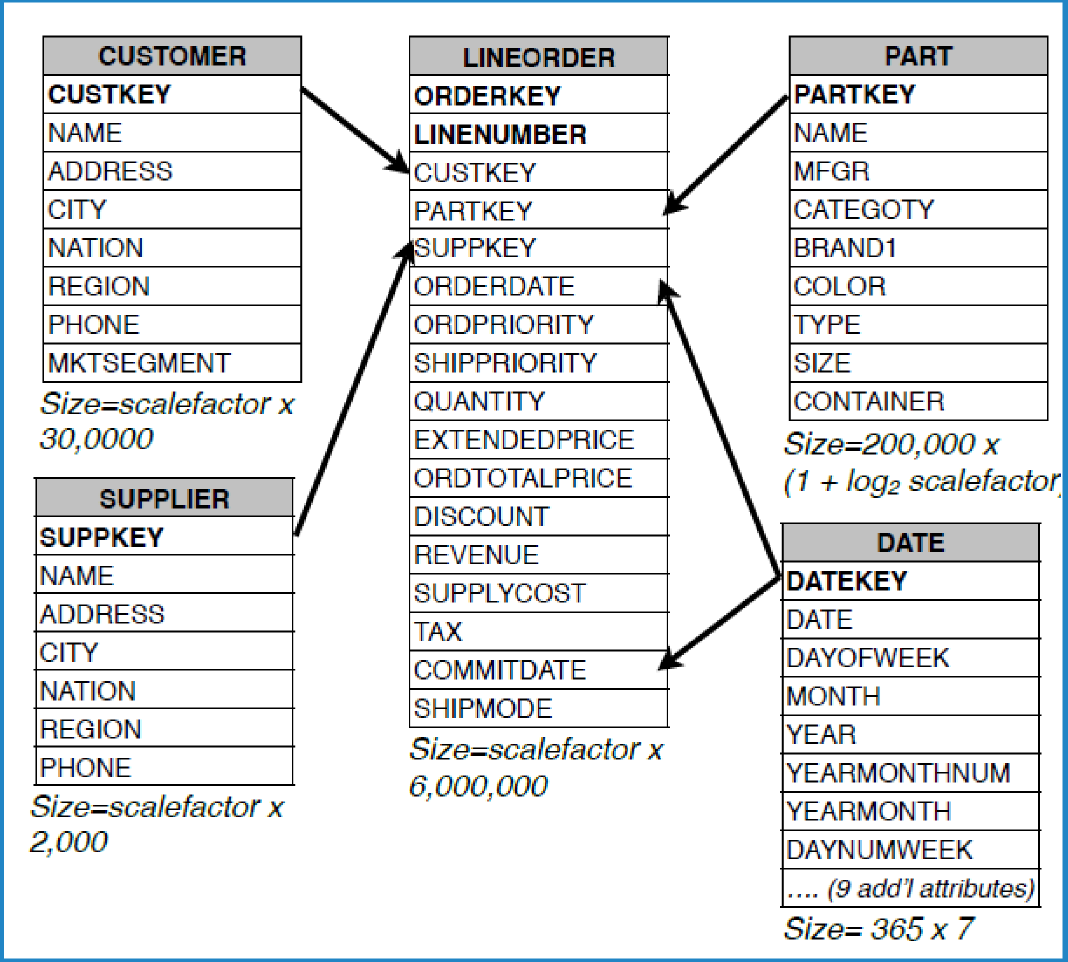
Contributions of this paper

- To show that trying to emulate a column-store in a row-store does not yield good performance results and that a variety of good techniques do little to improve the situation
- Propose a new technique for improving join performance in column stores called invisible joins, demonstrating that the execution of invisible join can perform better than selection and extraction from denormalized table.
- Break-down the sources of column database performance on warehouse workloads

Star Schema Benchmark

SSBM has 13 queries divided into four categories

- Flight 1 contains 3 queries
- Flight 2 contains 3 queries
- Flight 3 contains 4 queries
- Flight 4 contains 3 queries

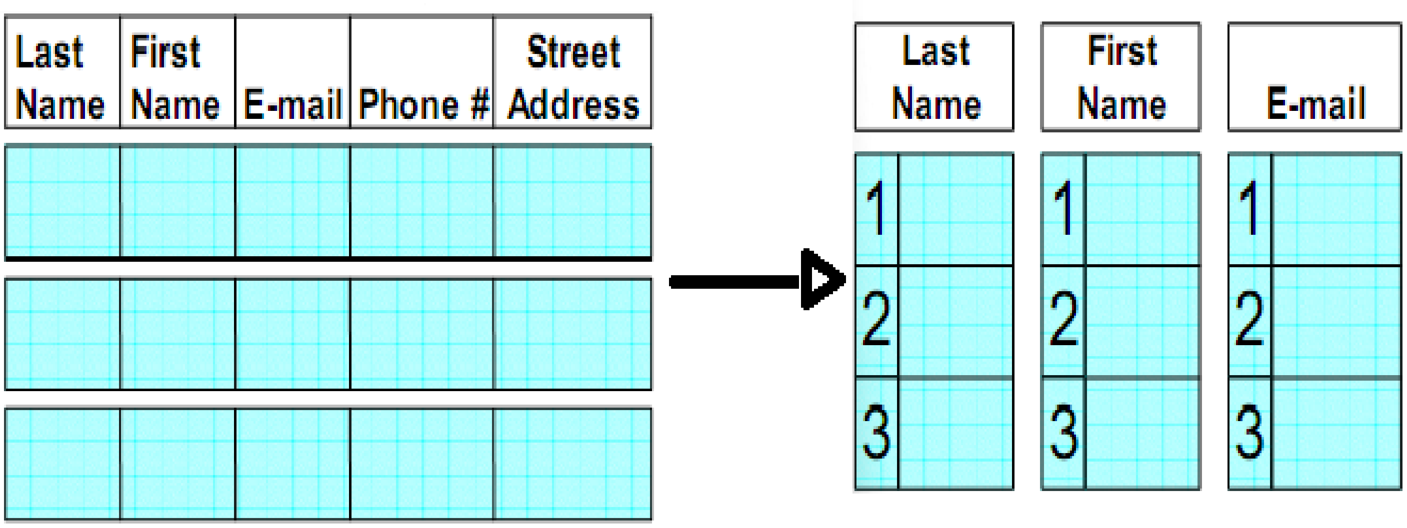


2.

Row oriented execution

Vertical Partitioning

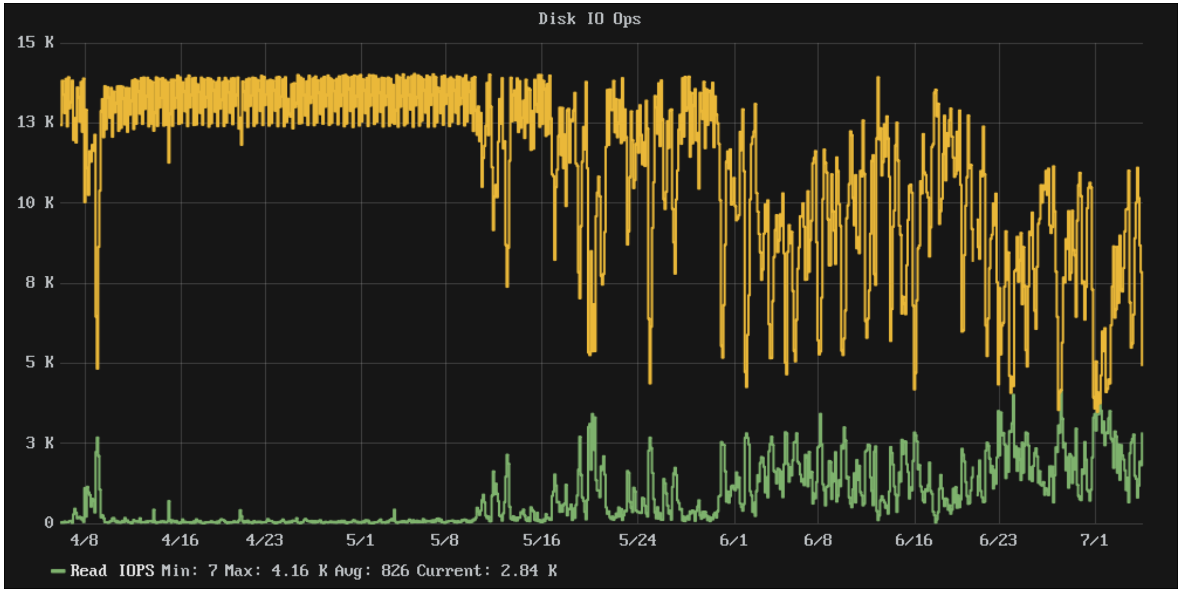
- Each column is made into one physical table
- An integer position column is added to each table
- Integer position is better than adding primary keys
- Join is made on this position column for a multicolumn fetch
- Header for every tuple will cause further space wastage



Vertical Partitioning

Problems with Vertical Partitioning

- It requires position attribute to be stored in every column, causing space and disk bandwidth
- Most row-stores store a relatively large header on every tuple, wasting further space.



Index only plans

- Adding B+ Tree index for every Table
- Never access the actual tuples on disk
- Has a low tuple-overhead than vertical partitioning

Problems:

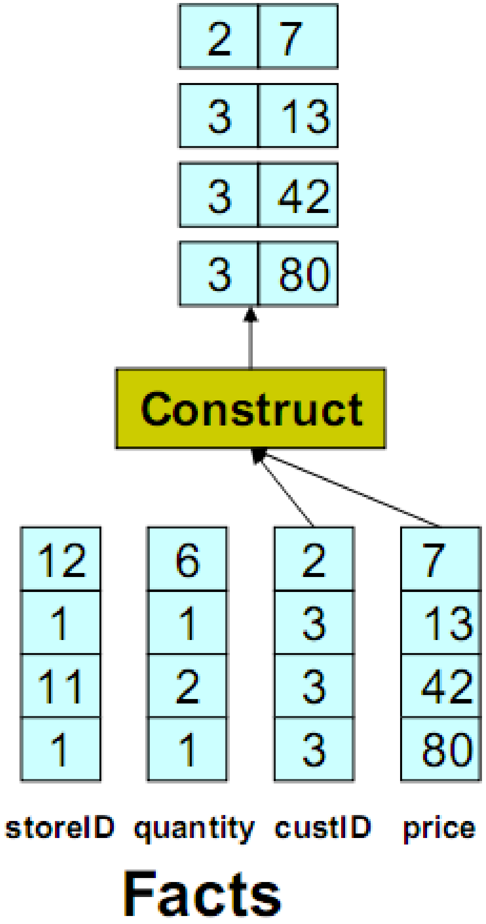
- Separate indices may require full scan, which is slower
- Composite Index might help
- Giant Hash joins will ruin the performance

Materialized Views

- Creating views for every flight
- Optimal set of MVs will have only columns needed for that flight
- Better than the other two process

Problems:

- Practical only in limited situations
- Requires knowledge of query workloads in advance



SELECT custID
FROM FACTS
WHERE price > 20

3.
Column oriented execution

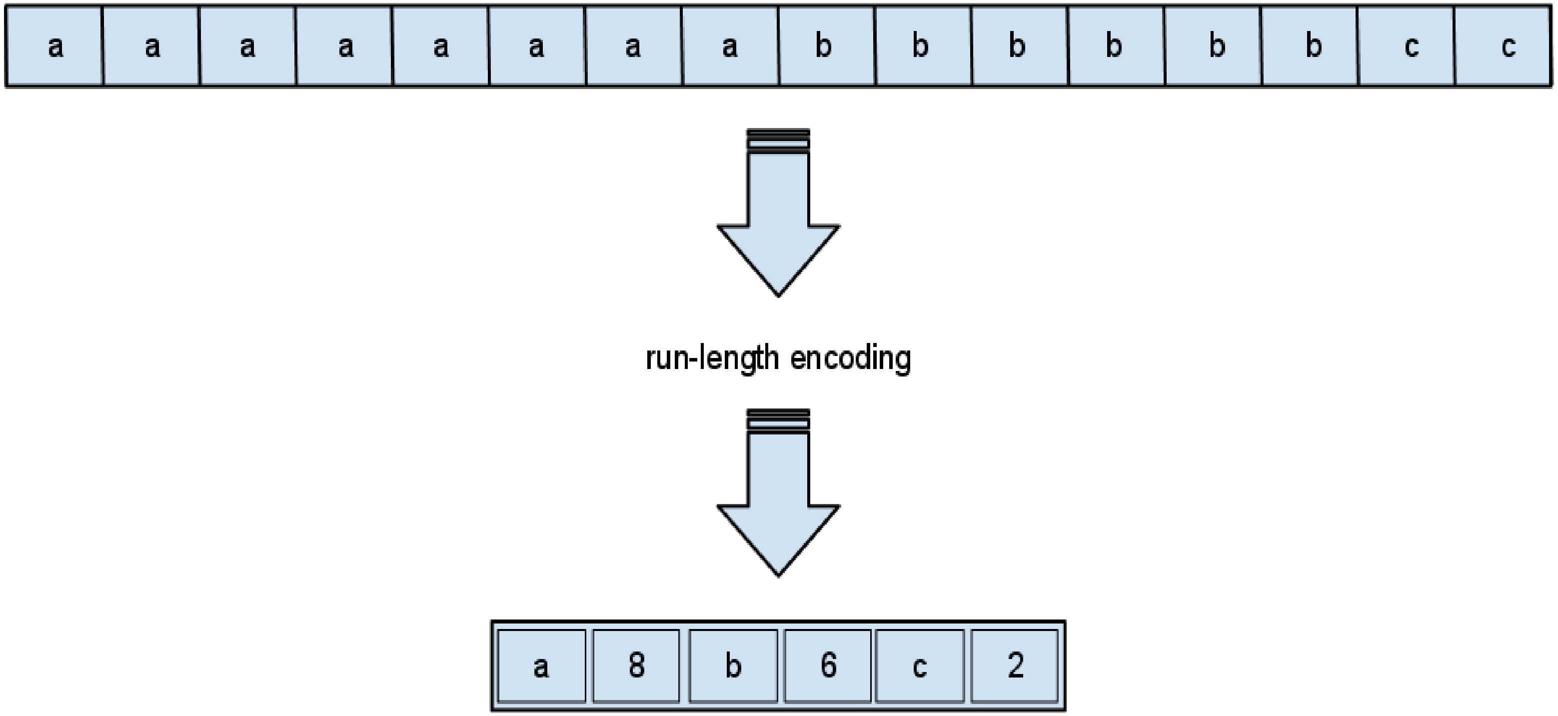
Compression

- Keeping data in compressed format as it is operated
- Data stored in columns is more compressible than data stored in rows
- Compression algorithms perform well on data with low entropy

Pros:

- Disk Space is saved
- CPU cost is low
- Best performance can be achieved using light weight compression schemes

- For schemes like run-length encoding, operating directly on compressed data results in the ability of a query executor to perform same operation on multiple column values at once
- Reduces CPU costs further



Late Materialization

- Majority of query results are entity-at-a-time and not column-at-a-time
- So in most query plans, data from multiple columns must be combined together into ‘rows’ of information about an entity
- Naïve column stores read in only those columns relevant for a particular query, construct tuples from their component attributes and executes normal row store operators.
- In Column stores with late materialization, the predicates are applied to the column for each attribute separately
- List of positions of values that passed the predicates are produced

Advantages of Late Materialization

- Avoids construction of complete tuples
- Direct operation on compressed data
- Cache performance is improved

Block Iteration

- Operators operate on blocks of tuples at once
- Iterate over blocks rather than tuples like batch processing
- If column is fixed width, it can be operated as an array
- Minimizes per-tuple overhead
- Exploits potential for parallelism as loop pipelining techniques can be used

Invisible Join

- Traditional Execution of Queries
  - Restrict set of tuple in the fact table using selection predicates on dimension table
  - Perform aggregation on the restricted fact table
  - Group other dimensional table attributes
  - Create joins between facts and dimension tables for each selection predicate and aggregate grouping

```
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;
```

Invisible Join vs Traditional Joins

- Traditional plan lacks all the advantages described previously of late materialized join
- In late materialized join group by columns need to be extracted in out-of-position order
- Invisible Join is a late materialized join but it minimizes the values that need to be extracted out of order.
- It rewrites joins into predicates on the foreign key columns in the fact table
- These predicates evaluated either by hash-lookup or between-predicate rewriting

Invisible Join – 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

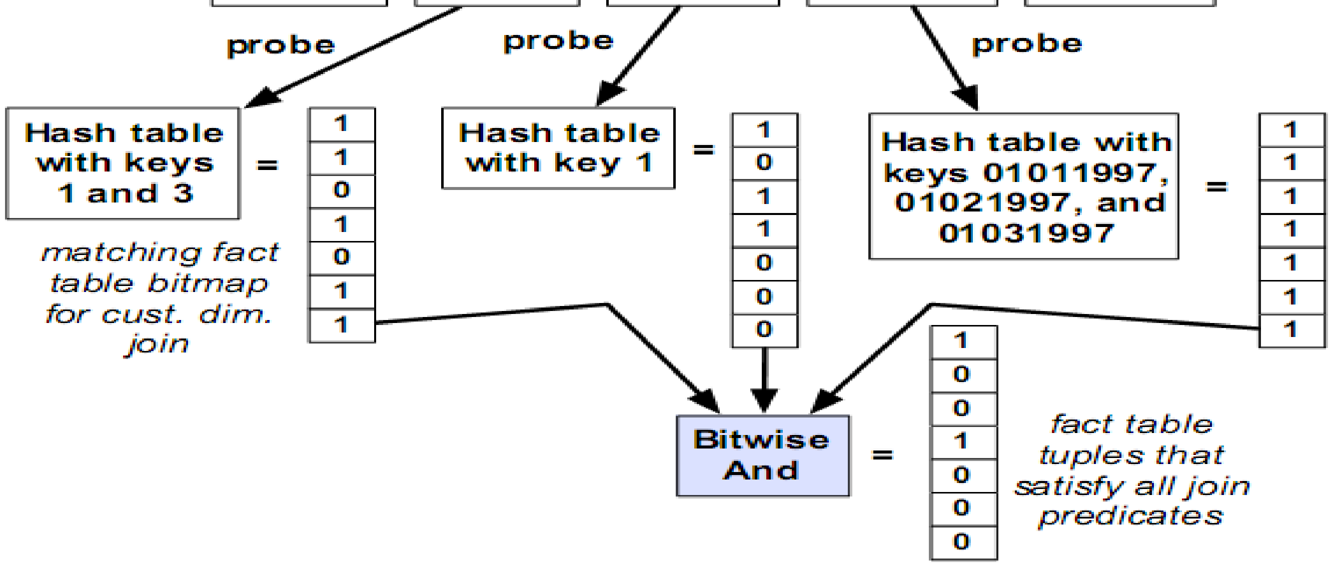
dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

Hash table with keys 01011997, 01021997, and 01031997

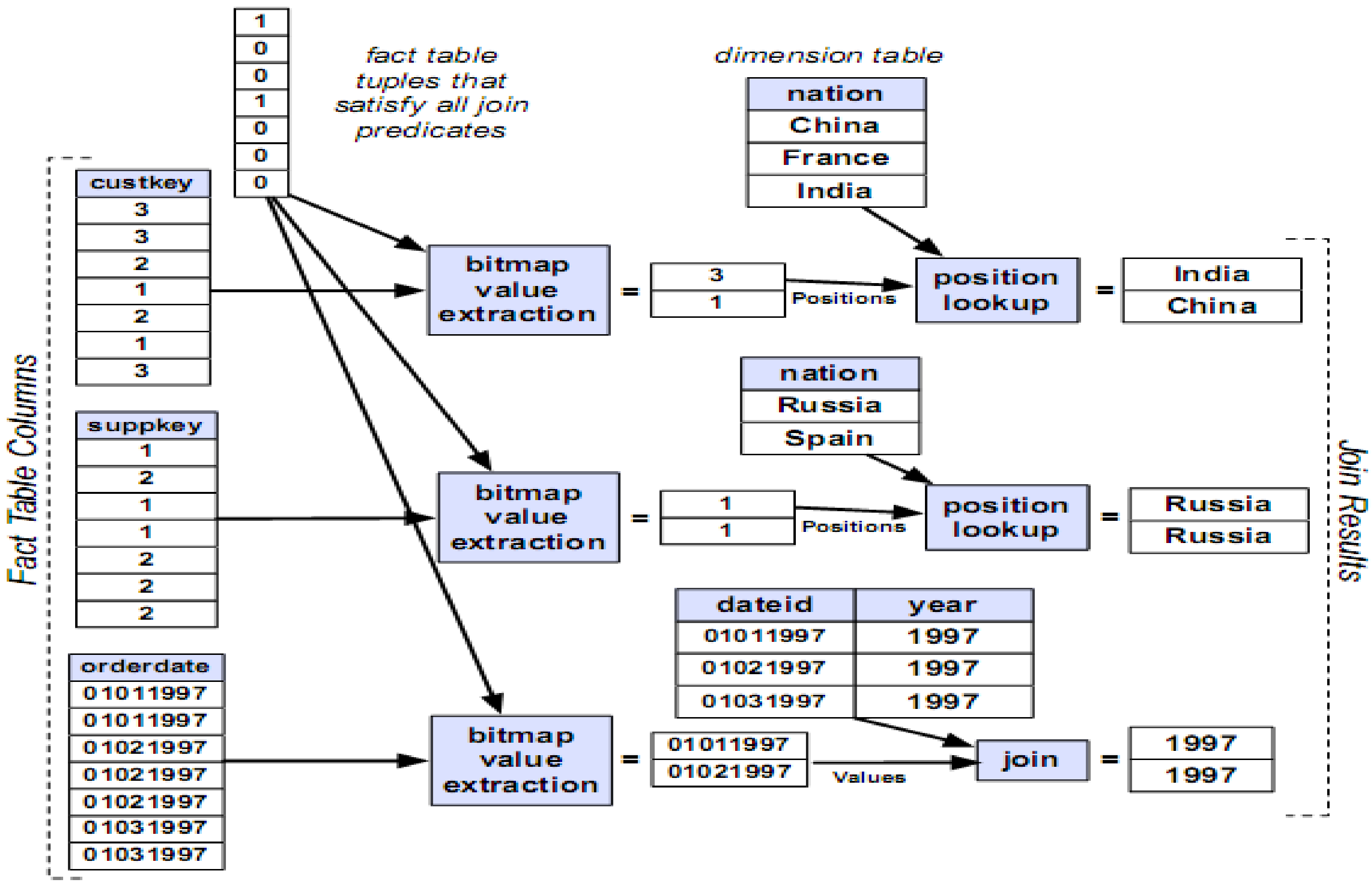
Invisible Join – Phase 2

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



Invisible Join – Phase 3



Between-Predicate Rewriting

- Range Predicates are used instead of hash lookups in Phase 1
- Effective and useful when contiguous set of keys are valid
- Dictionary encoding for key reassignment if not contiguous
- Usually gives a significant performance gain
- Does not need query optimizer to detect optimizations
- The code that evaluates predicates against the dimension tables is capable of detecting whether the result set is contiguous, then this technique is applied

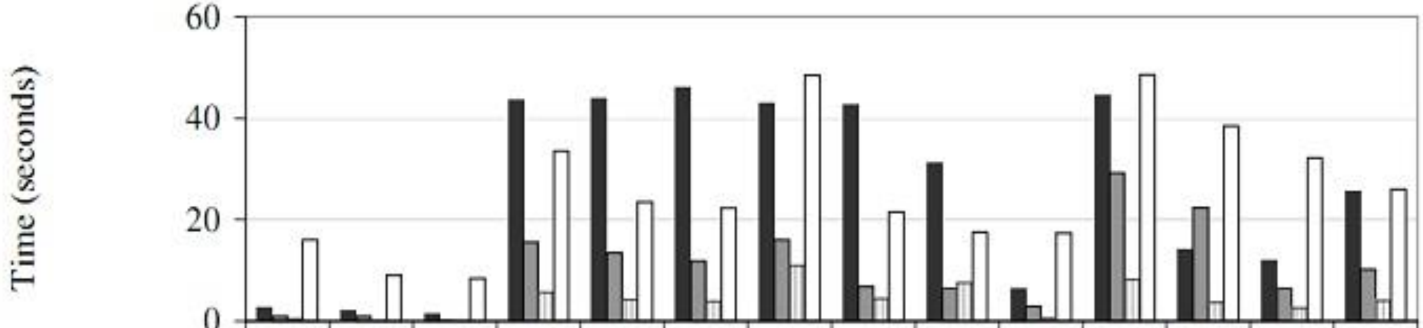
EXPERIMENTS

- To Emulate a column store in a row store with the baseline performance of C-store
- To try whether it is possible for a row-store to achieve the benefits of a column store database
- To try different optimization technique in column-store

EXPERIMENT SETUP

- 2.8GHz Dual Core Pentium(R) workstation
- 3 GB RAM
- RedHat Enterprise Linux 5
- 4 disk array mapped as a single logical volume
- Reported numbers are average of several run
- System X – Commercial Row Oriented Database

C-Store Vs Commercial Row Oriented DB



	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.

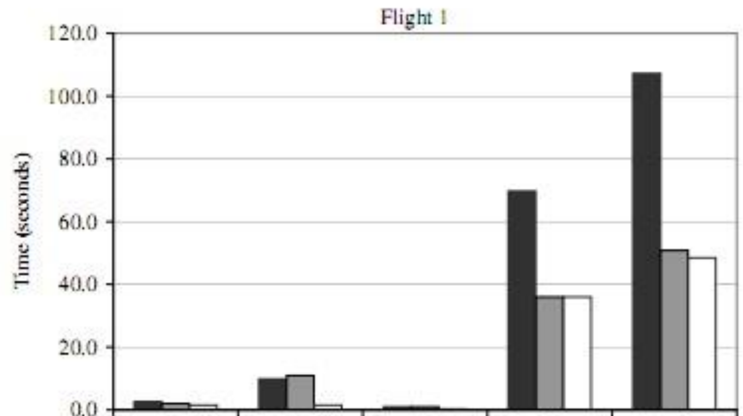
Results

- C-Store is better than System X by a factor of 6 in base case
- Beats System X by a factor of 3 when System X uses materialized view
- C-Store(MV) performs poor than RS(MV)
- C-Store has multiple known performance bottleneck
- No support for partitioning, multithreading

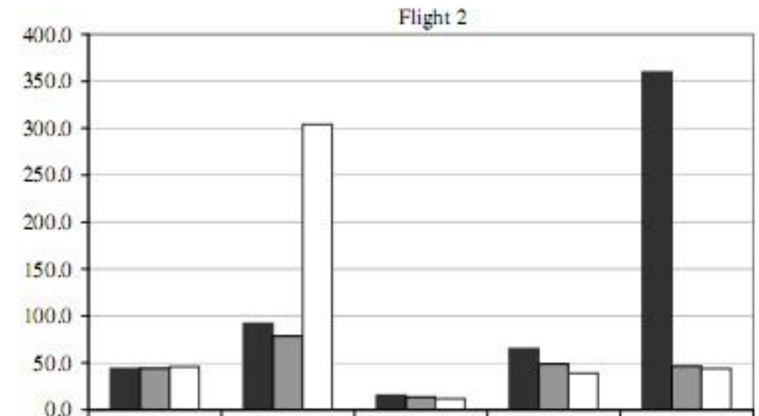
Column Store Simulation in a Row Store

- A Row Store database is designed in various ways to simulate a column store database
- Partitioning the row-store database is the base case
- Star join and Bloom Filters will be used
- Bloom Filters are probabilistic data structures used for optimization
- 5 different configuration
  - Traditional row oriented representation with bitmap and bloom filter
  - Traditional (bitmap): Plans are biased to use bitmaps
  - Vertical Partitioning: Each column is a relation
  - Index-Only: B+ Tree on each column
  - Materialized Views: Optimal set of views for every query

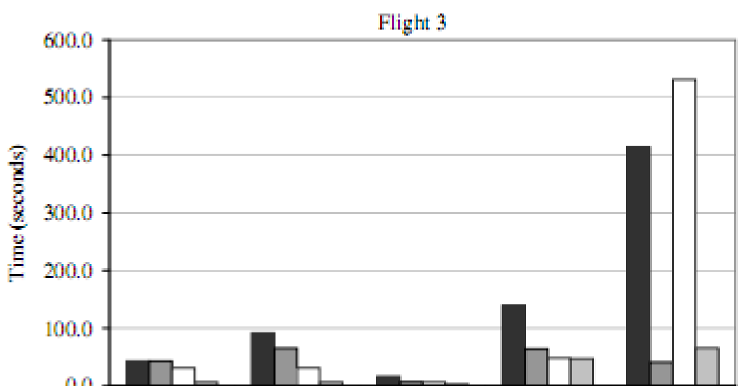
Column Store Simulation in a Row Store



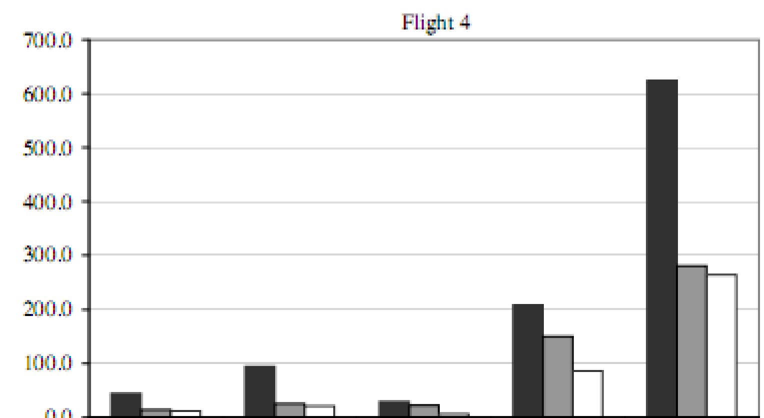
	T	T(B)	MV	VP	AI
Q1.1	2.7	9.9	1.0	69.7	107.2
Q1.2	2.0	11.0	1.0	36.0	50.8
Q1.3	1.5	1.5	0.2	36.0	48.5



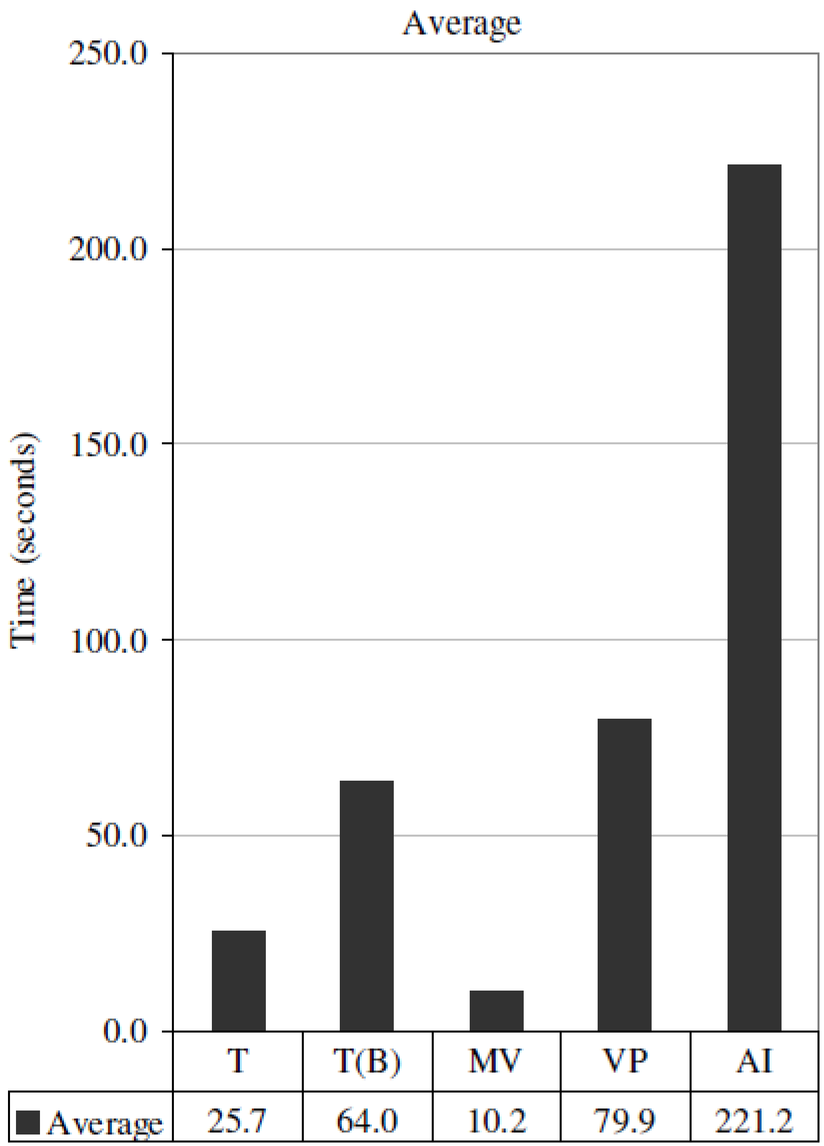
	T	T(B)	MV	VP	AI
Q2.1	43.8	91.9	15.5	65.1	359.8
Q2.2	44.1	78.4	13.5	48.8	46.4
Q2.3	46.0	304.1	11.8	39.0	43.9



	T	T(B)	MV	VP	AI
Q3.1	43.0	91.4	16.1	139.1	413.8
Q3.2	42.8	65.3	6.9	63.9	40.7
Q3.3	31.2	31.2	6.4	48.2	531.4
Q3.4	6.5	6.5	3.0	47.0	65.5



	T	T(B)	MV	VP	AI
Q4.1	44.4	94.4	29.2	208.6	623.9
Q4.2	14.1	25.3	22.4	150.4	280.1
Q4.3	12.2	21.2	6.4	86.3	263.9


Average
250.0
200.0
150.0
100.0
50.0
0.0
Time (seconds)
T
T(B)
MV
VP
AI
Average
25.7
64.0
10.2
79.9
221.2

Column Store Simulation in a Row Store

- Best Choice - Materialized view
- Worst Choice - Index only plans
- Traditional Method – Better Performance

Column Store Simulation in a Row Store

- Tuple Overheads
  - LineOrder Table – 60 million tuples, 17 columns
  - Compressed data
  - 8 bytes of over head per row
  - 4 bytes of record-id

	1 Column	Whole Table
Row Store	0.7- 1.1 GB	4 GB
Column Store	240 MB	2.3 GB

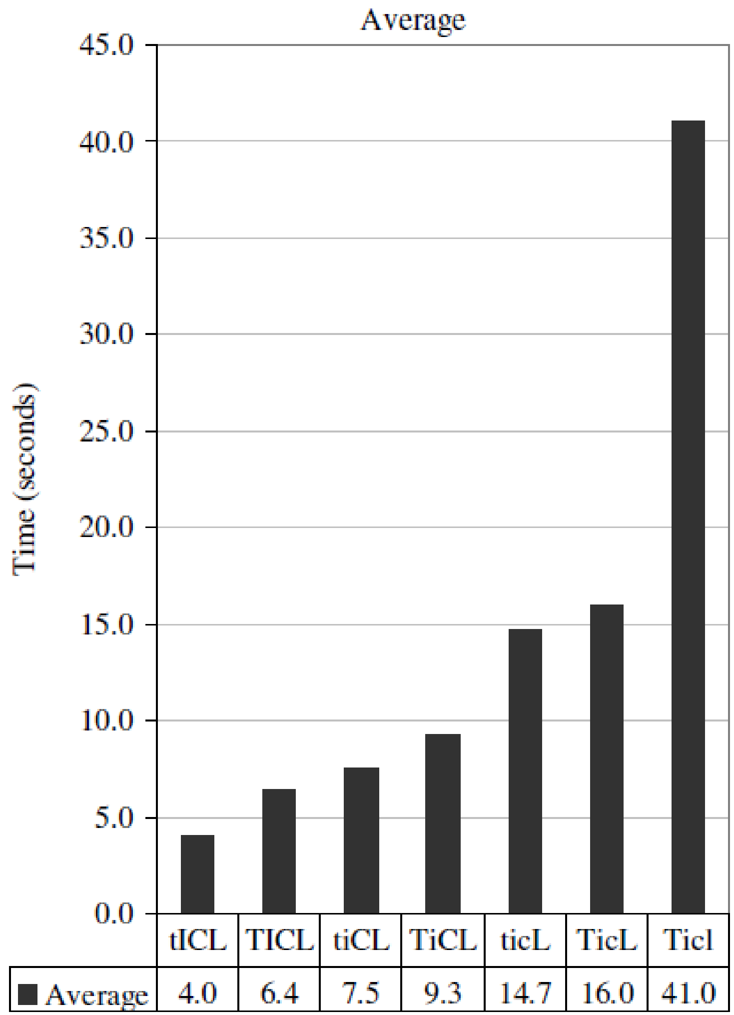
Column Store Simulation in a Row Store - Analysis

```
SELECT sum(lo.revenue), d.year, p.brand1
FROM lineorder AS lo, dwdate AS d,
     part AS p, supplier AS s
WHERE lo.orderdate = d.datekey
  AND lo.partkey = p.partkey
  AND lo.suppkey = s.suppkey
  AND p.category = MFGR#12
  AND s.region = AMERICA
GROUP BY d.year, p.brand1
ORDER BY d.year, p.brand1
```

Method	Time
Traditional	43
Vertical Partitioning	65
Index-only plans	360

Column Store Performance

- Column store is always better than the best of row store
- Block processing improves the performance by a factor 5 to 50 percent
- Late Materialization increases performance
- Invisible join improves by 50 to 75 percent



- 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;

Column Store Performance – Readings & Analysis

- Most significant two optimizations are compression and late materialization
- When all these optimizations are removed then the column store is same as the row store
- Denormalization is not useful for column stores

Conclusion

- We can emulate column store in row store using the following aspects but performance cant match that of column store’s
  - Vertical partioning
  - Index only plan
- High per-tuple overheads, high tuple reconstruction cost
- Reasons for performance in Column Store
  - Late materialization
  - Compression
  - Block iteration
  - Invisible join

Thanks!

Any questions?