Column-Stores vs. Row-Stores: How Different Are They Really?

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

Presented by:

Souvik Pal
Subhro Bhattacharyya

Department of Computer Science
Indian Institute of Technology, Bombay

Outline

1. Introduction
2. Column-Stores vs. Row-Stores
   - Row-oriented execution
   - Column-oriented execution
3. Experiments
4. Conclusion

Column Stores

Figure 1: Column Stores [1].

- Row store: Data are stored in the disk tuple by tuple
- Column store: Data are stored in the disk column by column

Column Stores

- A relational DB shows its data as 2D tables of columns and rows

Example

EmpId	Lastname	Firstname	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

Table 1: Column store vs Row Store [2]

Column Stores

- A relational DB shows its data as 2D tables of columns and rows
- Row Store: serializes all values of a row together

Example

EmpId	Lastname	Firstname	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

Table 1: Column store vs Row Store [2]

Row Store

1,Smith,Joe,40000;
2,Jones,Mary,50000;
3,Johnson,Cathy,44000;

Column Stores

- A relational DB shows its data as 2D tables of columns and rows
- Row Store: serializes all values of a row together
- Column Store: serializes all values of a column together

Example

EmpId	Lastname	Firstname	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

Table 1: Column store vs Row Store [2]

Row Store

1,Smith,Joe,40000;
2,Jones,Mary,50000;
3,Johnson,Cathy,44000;

Column Store

1,2,3;
Smith,Jones,Johnson;
Joe,Mary,Cathy;
40000,50000,44000;

Column Stores

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

Table 2: Column store vs Row Store [1]

Column Stores

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

Table 2: Column store vs Row Store [1]

Column stores are suitable for read-mostly, read-intensive, large data repositories

- data warehouses
- decision support applications
- business intelligent applications

For performance comparison, the star schema bench mark is used(SSBM)

Outline

Outline

Simulating a Column-Store in a Row-Store

Column-store performance from a row-store?

- Vertical Partitioning
- Index-only plans
- Materialized Views

Vertical Partitioning

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

→

Date

POS	VALUE
1	01/01
2	01/01
3	01/01

Store

POS	VALUE
1	BOS
2	NYC
3	BOS

Cust

POS	VALUE
1	Mesa
2	Lutz
3	Mudd

Product

POS	VALUE
1	Table
2	Chair
3	Bed

Price

POS	VALUE
1	$20
2	$15
3	$90

Figure 2: Vertical Partitioning [1].

Features

- Full vertical partitioning of each relation.
- 1 physical table for each column.

Vertical Partitioning

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

→

Date

POS	VALUE
1	01/01
2	01/01
3	01/01

Store

POS	VALUE
1	BOS
2	NYC
3	BOS

Cust

POS	VALUE
1	Mesa
2	Lutz
3	Mudd

Product

POS	VALUE
1	Table
2	Chair
3	Bed

Price

POS	VALUE
1	$20
2	$15
3	$90

Figure 2: Vertical Partitioning [1].

Features

- Primary key of relation may be long and composite
- Integer valued "position" column for each table.
- Thus each table has 2 columns.
- Joins required on "position" attribute for multi-column fetch.

Vertical Partitioning

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

→

Date

POS	VALUE
1	01/01
2	01/01
3	01/01

Store

POS	VALUE
1	BOS
2	NYC
3	BOS

Cust

POS	VALUE
1	Mesa
2	Lutz
3	Mudd

Product

POS	VALUE
1	Table
2	Chair
3	Bed

Price

POS	VALUE
1	$20
2	$15
3	$90

Figure 2: Vertical Partitioning [1].

Problems

- “Position” attribute: stored for every column
  - wastes disk space and bandwidth
- large header per tuple
  - more space is wasted
- Joining tables for multi-column fetch
  - Hash Join slow
  - Index Join slower

Index-only plans

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

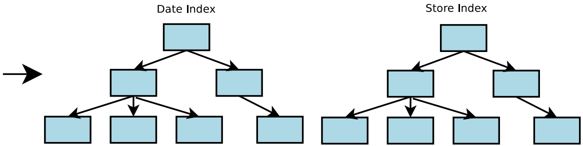


Figure 3: Index-only plans [1].

Features

- Unclustered B+ tree index on each table column
- Plans never access actual tuples on the disk
- Tuple headers not stored, so overhead is less

Index-only plans

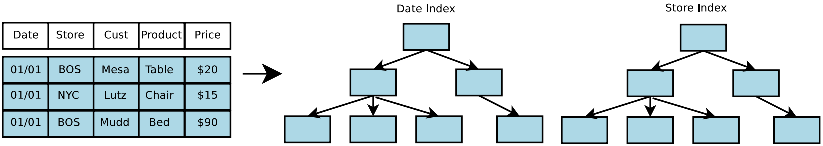


Figure 3: Index-only plans [1].

Features

- Indices stored as (record-id, value) pairs.
- All rids stored
- No duplicate values stored

Index-only plans

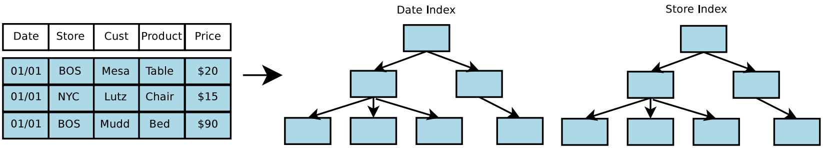


Figure 3: Index-only plans [1].

Problems

- Separate indices may require full index scan which is slow
- Solution: Composite indices required to answer queries directly

Example

SELECT AVG(SALARY) FROM EMP WHERE AGE>40

Materialized views

Features

- Optimal set of MVs created for given query
- Contains only those columns required to answer the query.
- Tuple headers are stored just once per tuple
- Provides just the required amount of data

Problems

- Query should be known in advance

Outline

Optimizations in Column-Oriented DBs

- Compression
- Late Materialization
- Block Iteration
- Invisible Join

Compression

Features

- Low information entropy in columns than rows
- Decompression performance more valuable than compression achievable

Advantages

- Low disk space
- Lesser I/O
- Performance increases if queries executed directly on compressed data

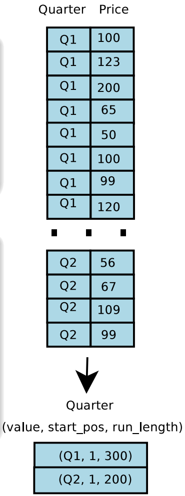


Figure 4: Compression [1].

Late Materialization

- Information about entities stored in different tables.
- Most queries access multiple attributes of an entity.

Naive column-store approach-Early Materialization

- Read necessary columns from disk
- Construct tuples from component attributes
- Perform normal row-store operations of these tuples
- Much of performance potential unused

Late Materialization

Features

- Keep data in columns and operate on column data until late into the query plan
- Intermediate "position" lists need to be created.
- Required for matching up operations performed on different columns.

Example

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

Late Materialization

Advantages

- Selection and Aggregation limits the number of tuples generated
- Compressed data need not be decompressed for creating tuples
- Better cache performance – PAX
- Block iteration works better on columns than on rows

Partition Attributes Across (PAX)

Features

- column interleaving
- minimal row reconstruction cost
- only relevant data in cache
- minimizes cache misses
- effective when applying querying on a particular attribute

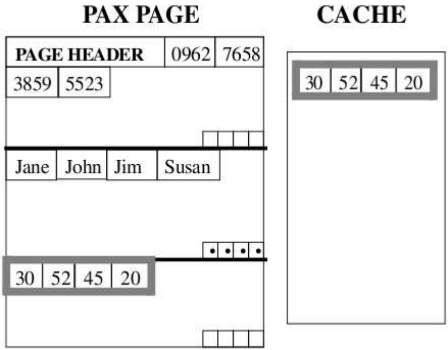


Figure 5: PAX [3].

Block Iteration

Features

- Operators operate on blocks of tuples at once
  - Iterate over blocks of tuples rather than a single tuple
  - Avoids multiple function calls on each tuple to extract data
  - Data is extracted from a batch of tuples
- Fixed length columns can be operated as arrays
  - Minimizes per-tuple overhead
  - Exploits potential for parallelism

Star Schema Benchmark

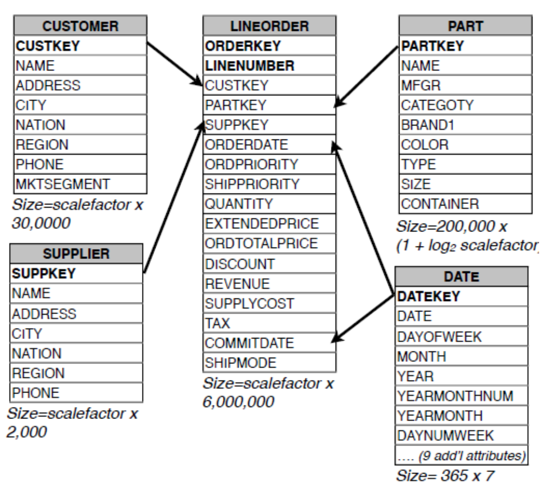


Figure 6: Star Schema Benchmark [4].

Invisible Join

Example

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 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 customers who live in ASIA
- and who purchase from an Asian supplier between 1992 and 1997
- grouped by nation of customer, nation of supplier and year of transaction

Invisible Join

Traditional Plan

Pipelines join in order of predicate selectivity.
Disadvantage: misses out on late materialization

Late materialized join: Disadvantage

After join the list of positions for dimension tables are unordered

Group by columns in dimension tables need to be extracted in out-of-position order.

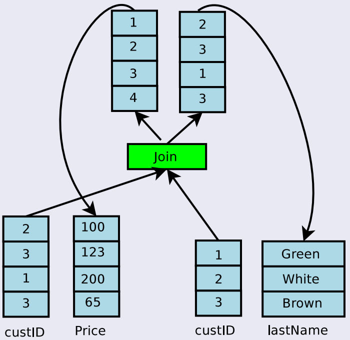


Figure 7: Late materialization [1].

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

Figure 8: Phase 1 [4].

Invisible Join

Phase 2

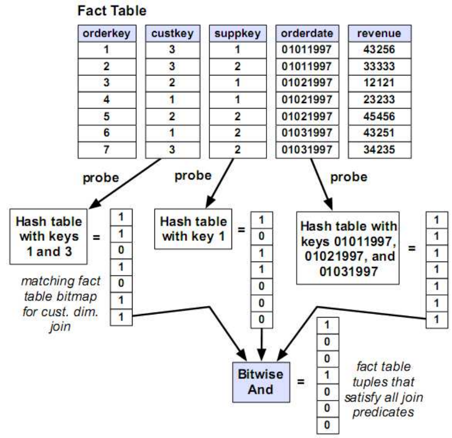


Figure 9: Phase 2 [4].

Invisible Join

Phase 3

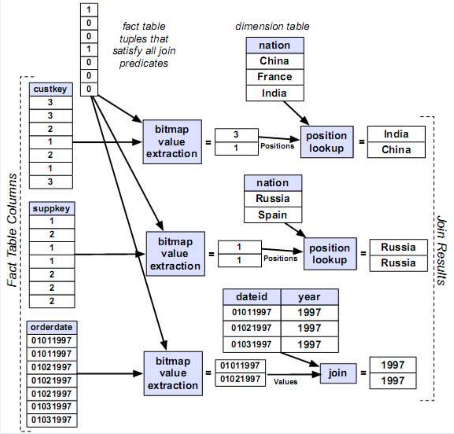


Figure 10: Phase 3 [4].

Invisible Join

Between-Predicate Rewriting

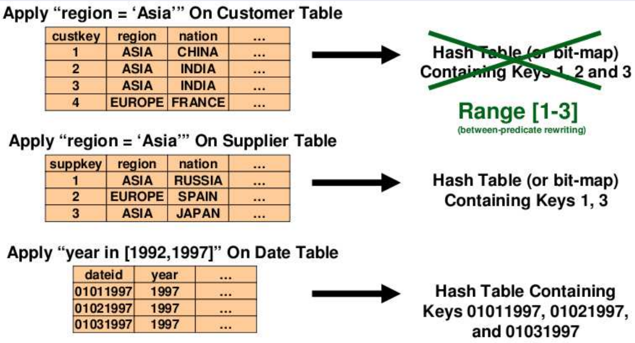


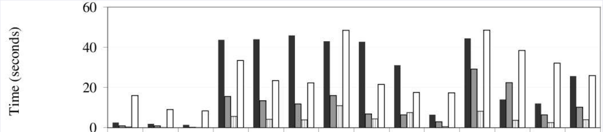
Figure 11: Between-Predicate Rewriting [1].

Outline

Goal

- Performance comparison of C-Store with R-Store
- Performance comparison of C-Store with column-store simulation on a R-Store
- Finding the best optimization for a column-store
- Comparison between invisible join and denormalized table

C-Store(CS) vs. System-X(RS)



	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

RS – Row Store

CS – Column store

RS(MV) – Row Store with optimal set of materialized views

CS (Row-MV) – Column store constructed from RS(MV)

Figure 12: C-Store(CS) and System-X(RS) [4].

C-Store(CS) vs. System-X(RS)

- First three rows as per expectation.
- For CS(Row-MV) materialized data is stored as strings in C-store.
- Expected that both RS(MV) and CS(Row-MV) will perform similarly
- However RS(MV) performs better
  - No support for multi-threading and partitioning in C-Store.
  - Disabling partitioning in RS(MV) halves performance
  - Difficult to compare across systems
- C-Store(CS) 6 times faster than CS(Row-MV)
  - Both read minimal amount of data from disk to answer a query
  - I/O savings- not the only reason for performance advantage

Column store simulation in Row store

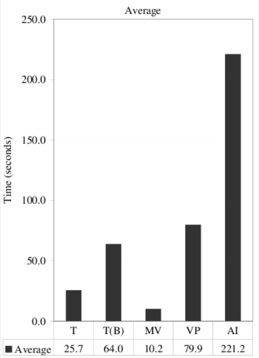
- Traditional
- Vertical Partitioning: Each column is a relation
- Index-only plans: B+Tree on each column
- Materialized Views: Optimal set of views for every query

Column store simulation in Row store

- MV < T < VP < AI (time taken)
- Without partitioning, T ≈ VP
- Vertical partitioning: Tuple Overhead

	1 Column	Whole Table
T		4 GB
VP	1.1 GB	
CS	240 MB	2.3 GB

- Index-only plans: Column Joins
  - Hash Join: takes a long time
  - Index Join: high index access overhead
  - Merge Join: unable to skip sort step



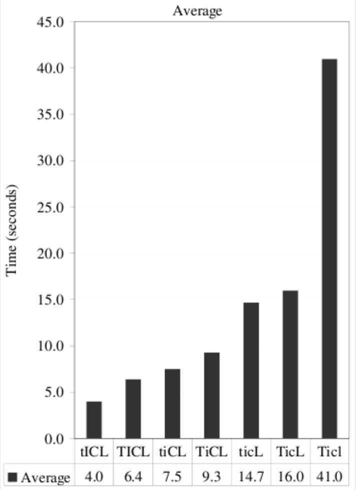
T – Traditional VP – vertical partitioning
T(B) – Traditional(bitmap)
MV – materialized views AI – All indexes

Figure 13: Column store simulation in Row store [4].

Breakdown of Column-Store Advantages

- Start with C-Store
- Remove optimizations one by one
- Finally emulate Row-Store

- Late materialization improves 3 times
- Compression improves 2 times
- Invisible Join improves 50%
- Block processing improves 5-50%



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

Outline

Conclusion

- C-Store emulation on R-Store is done by vertical partitioning, index plans
- Emulation does not yield good performance
- Reasons for low performance by emulation
  - High tuple reconstruction costs
  - High tuple overhead
- Reasons for high performance of C-Store
  - Late Materialization
  - Compression
  - Invisible Join

References

[1] S. Harizopoulos, D. Abadi, and P. Boncz, "Column-Oriented Database Systems," in *VLDB*, 2009.

[2] http://en.wikipedia.org/wiki/Column-oriented_DBMS.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving Relations for Cache Performance," in *VLDB*, 2001.

[4] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-Stores vs. Row-Stores: How Different Are They Really?" in *SIGMOD*, June 2008.