

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

Volodymyr Piven

Wilhelm-Schickard-Institut für Informatik
Eberhard-Karls-Universität Tübingen

21. Januar 2011

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



To Buy or not To Buy

Soll man sich wirklich Column-Stores kaufen?

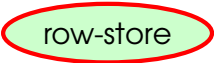


To Buy or not To Buy

Key Questions

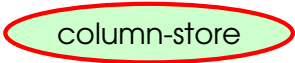
1. Emulate of column-store in a row-store
2. Unmodified row-store vs. column-oriented design
3. Optimizations of column-stores
4. Invisible join vs. denormalized fact table

Comparing of row-store with column-store



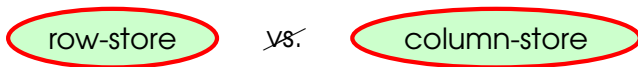
row-store

vs.

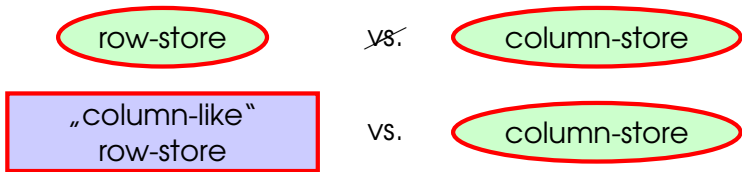


column-store

Comparing of row-store with column-store



Comparing of row-store with column-store



Comparing of row-store with column-store

row-store

~~vs.~~

column-store

„column-like“
row-store

vs.

column-store

row-store

vs.

„row-like“
column-store

First Phase

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	...

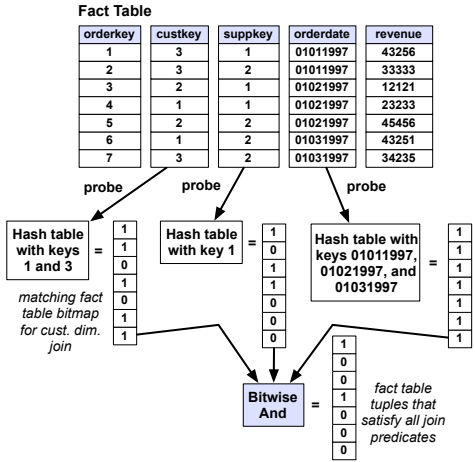
Hash table
with key 1

Apply `year in [1992,1997]` on Date table

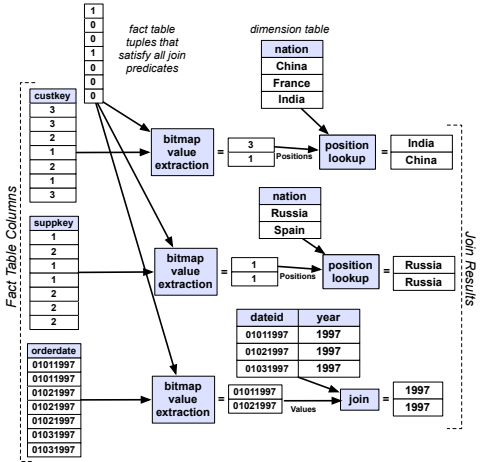
dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

Hash table with
keys 01011997,
01021997, and
01031997

Second Phase



Third Phase

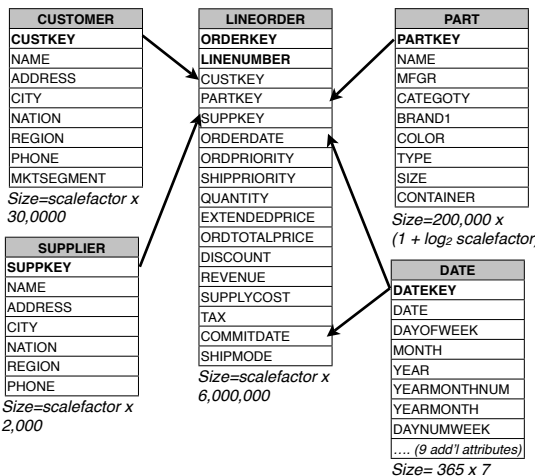


Versuchaufbau

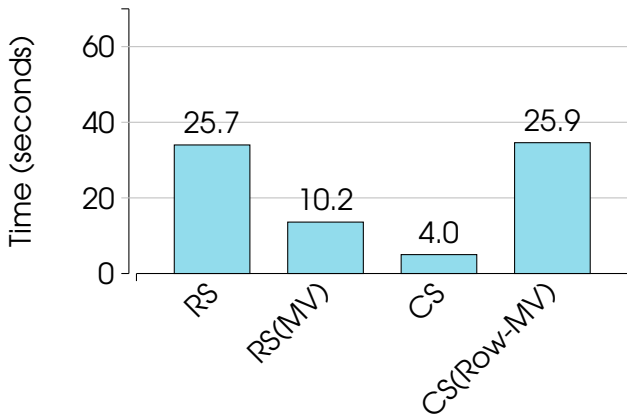
- 2.8 GHz single processor
- Dual core Pentium® D
- 3 GB of RAM
- RedHat Enterprise Linux 5



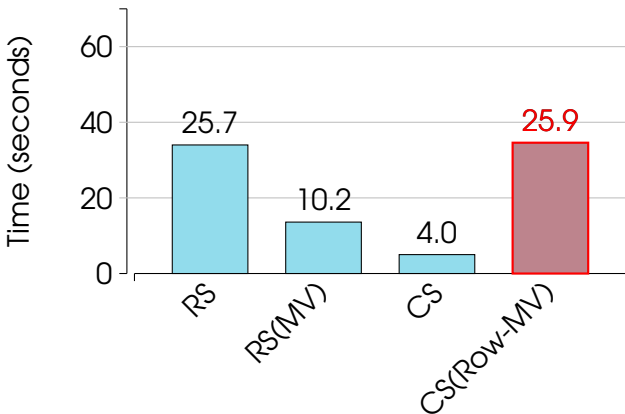
Schema of the SSBM Benchmark



Baseline performance of C-Store and System X



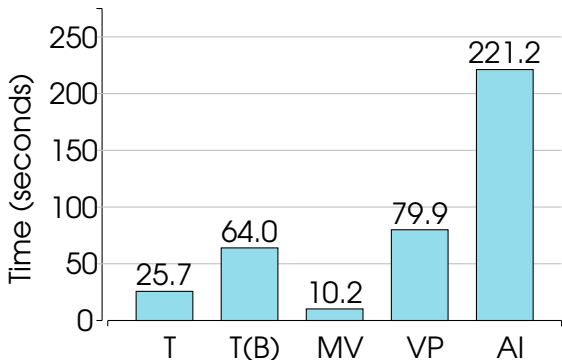
Baseline performance of C-Store and System X



Row-Store Execution

- Vertical Partitioning
 - ▶ Each attribute is a two-column table (values, position)
- Index-All
 - ▶ Unclustered B⁺-Treeindex for every column of every table
- Materialized View
 - ▶ Optimal set of materialized views for every query

Average performance across all queries



T	Traditional
T(B)	Traditional (bitmap)
MV	Materialized View
VP	Vertical Partitioning
AI	Index-only

Reasons

- Tuple Overheads

- VP Single column-table requires 0.7 - 1.1 GByte (compressed)

- T Entire 17 column is 6 GByte (decompressed) or 4 GByte (compressed)

- Column Joins

- ▶ Hash-join is slow
 - ▶ Perhaps the best for Index-All

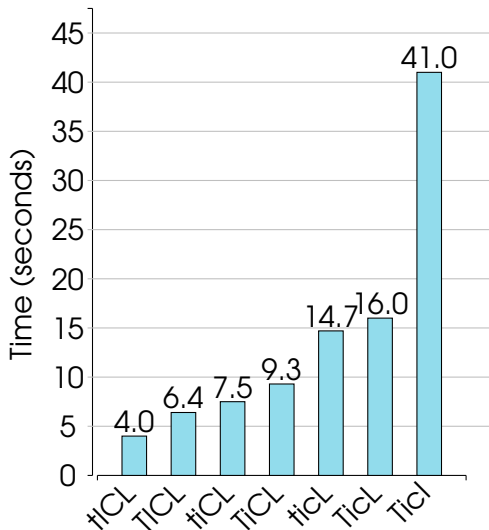
Column-Store Execution

- Compression
- Late Materialization
- Block Iteration
- Invisible Join

Column-Store Execution

- Compression
- Late Materialization
- Block Iteration
- Invisible Join

Average performance across all queries



T	Tuple
†	Block
I	Invisible Join
C	Compression
L	Late Materialization

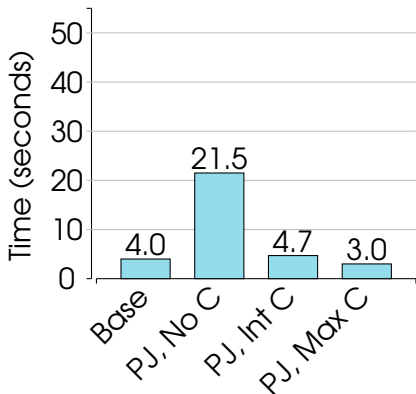
Analysis

- Block: 5% - 50%
- Compression: almost factor 2 averagely
- Late materialization: factor 3
- Invisible Join: 50% - 75%

Analysis

- Block: 5% - 50%
- Compression: almost factor 2 averagely
- Late materialization: factor 3
- Invisible Join: 50% - 75%
 - ▶ Optimization for star schemas

Average performance across all queries



No C	Not compressed
Int C	Dictionary compressed into integers
Max C	Compressed as much as possible

Summary

- Significant optimizations:
 - ▶ Compression
 - ▶ Late materialization
- Without optimizations column store acts just like a row store
- Invisible join makes denormalizationis useless