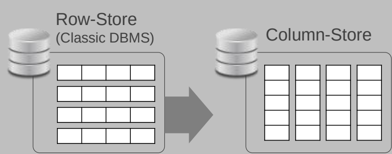
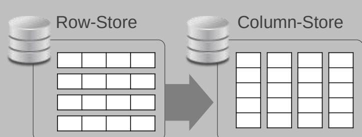
Column-Store: An Overview



Store one tuple at-a-time

Store one column at-a-time

Row-Store vs Column-Store



	Row-Store	Column-Store
Tuple Insertion:	+ Fast	− Requires multiple seeks
Queries:	− May access more data than needed	+ Read only relevant data
		Perfect for analytics!

Column-Store Optimizations

Operating on columns enables and/or is combined with the following optimizations:

- Compression
  Compress values per column
- Late Tuple Materialization
  Construct tuples as late as possible
- Block Iteration
  Pass blocks of values between operators
- Invisible Join

Column-Store Optimizations

Compression

Compress column data

Column-Store Optimizations > Compression

Why Compression

Advantages of compression in general:

- Lower storage space requirements
  Minor
- Better I/O performance
  Read fewer data (from disk, SSD, or RAM), gain from cache locality
- Better query processing performance
  Typically when operating directly on compressed data

Column-Store Optimizations > Compression

Why Column Store

Compress column data:

- One column at a time
  Data in a column more similar than data across columns

Name	Phone	City	State
Fred Flintstone	858-123-4567	San Diego	CA
Barney Rubble	619-000-0000	San Diego	CA
Maggie Simpson	415-999-2222	San Francisco	CA
James Bond	212-007-0000	New York	NY

Column-Store Optimizations > Compression

Compression Example

- Run-length encoding
  Replace list of identical values by pair (value, count)

State
CA
CA
CA
NY

→

State
CA x3
NY x1

Column-Store Optimizations > Compression

Compression Example

- Run-length encoding
  Replace list of identical values by run pair (value, count)

  ⊕ Good for sorted columns

State
CA
NY
CA
NY
CA

→

State
CA
NY
CA
NY
CA

Non-sorted

State
CA
CA
CA
NY
NY

→

State
CA x3
NY x2

Sorted

Column-Store Optimizations > Compression

Compression Example

- Run-length encoding
  Replace list of identical values by pair (value, count)

  ⊕ Enables query processing on compressed data directly

e.g., Select persons in CA

State
CA x3
NY x1

~~uncompress~~ →

State
CA
CA
CA
NY

$\sigma_{St = \text{“CA”}}$ →

State
CA
CA
CA

Column-Store Optimizations > Compression

Compression Example

- Run-length encoding
  Replace list of identical values by pair (value, count)

  ⊕ Enables query processing on compressed data directly

  e.g., Select persons in CA

State
CA x3
NY x1

$\xrightarrow{\sigma_{St = \text{“CA”}}}$

State
CA x3

Column-Store Optimizations > Compression

Other Compression Algos

- Dictionary Encoding
  Replace frequent patterns with smaller fixed length codes:
  eg, instead of string values “Dasgupta” → 0, “Freund” → 1, “Papakonstantinou” → 2
  Commonly used in row-stores also, since it enables fixed length fields, therefore random access.
- Bit-Vector Encoding
  Create for each possible value a bit vector with 1s in the positions containing the value: Useful for small domains.
  (Covered in the indexing section.)
- Heavyweight, Variable-Length Compression Schemes
  e.g., Huffman: Excellent compression ratio but (1) no random access (2) possibly poor decompression CPU performance
  Currently not used – they are good for selected workloads

Column-Store Optimizations

Late Tuple Materialization

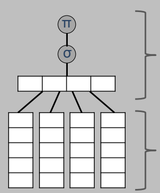
Create tuples as late in the query plan as possible

Column-Store Optimizations > Late Tuple Mat.

Early Tuple Materialization

Naïve method of query evaluation in a column store:

- Read relevant columns & create tuples
- Run query at a tuple-level as usual



Query Processing: **Rows**

Storage: **Columns**

Column-Store Optimizations > Late Tuple Mat.

Early Tuple Materialization

e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"

π_{Name}

$\sigma_{\text{City = "SD"} \wedge \text{State="CA"}}$

Create tuples

Name	Phone	City	State
Flintstone	858	SD	CA
Barney	619	SD	CA
Simpson	415	SF	CA
Bond	212	NY	NY

Column-Store Optimizations > Late Tuple Mat.

Early Tuple Materialization

e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"

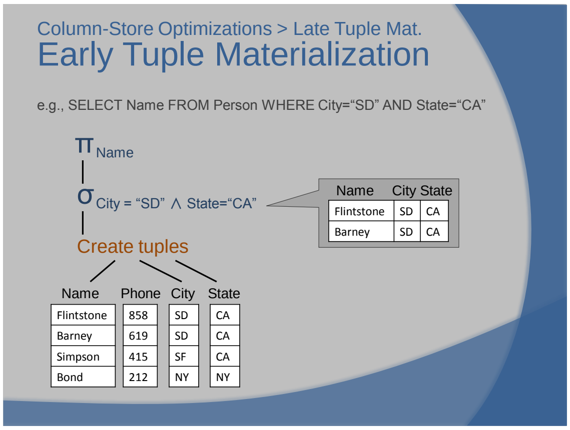
π_{Name}

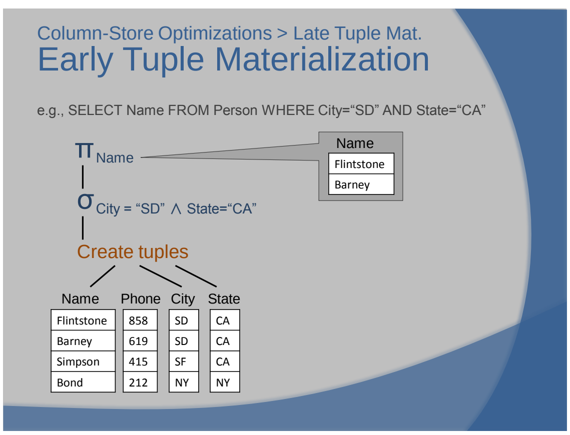
$\sigma_{\text{City = "SD"} \wedge \text{State="CA"}}$

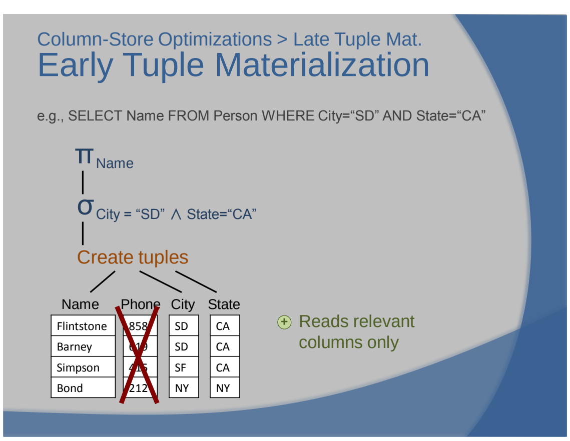
Create tuples

Name	City	State
Flintstone	SD	CA
Barney	SD	CA
Simpson	SF	CA
Bond	NY	NY

Name	Phone	City	State
Flintstone	858	SD	CA
Barney	619	SD	CA
Simpson	415	SF	CA
Bond	212	NY	NY


Column-Store Optimizations > Late Tuple Mat.
Early Tuple Materialization
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
π Name
σ City = "SD" ∧ State="CA"
Create tuples
Name City State
Flintstone SD CA
Barney SD CA
Name Phone City State
Flintstone 858 SD CA
Barney 619 SD CA
Simpson 415 SF CA
Bond 212 NY NY

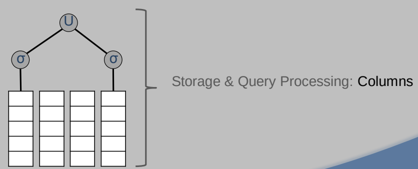

Column-Store Optimizations > Late Tuple Mat.
Early Tuple Materialization
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
π Name
Name
Flintstone
Barney
σ City = "SD" ∧ State="CA"
Create tuples
Name Phone City State
Flintstone 858 SD CA
Barney 619 SD CA
Simpson 415 SF CA
Bond 212 NY NY


Column-Store Optimizations > Late Tuple Mat.
Early Tuple Materialization
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
π Name
σ City = "SD" ∧ State="CA"
Create tuples
Name Phone City State
Flintstone 858 SD CA
Barney 619 SD CA
Simpson 415 SF CA
Bond 212 NY NY
Reads relevant columns only

Column-Store Optimizations > Late Tuple Mat.

Late Tuple Materialization

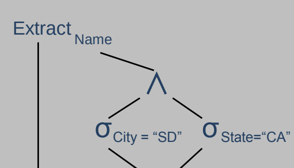
- Create tuples as late in the plan as possible



Column-Store Optimizations > Late Tuple Mat.

Late Tuple Materialization

e.g., SELECT Name FROM Person WHERE City=“SD” AND State=“CA”

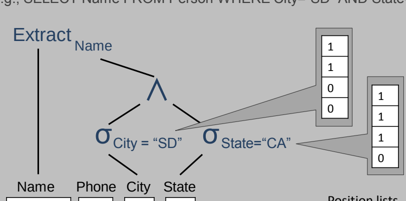


Name	Phone	City	State
Flintstone	858	SD	CA
Barney	619	SD	CA
Simpson	415	SF	CA
Bond	212	NY	NY

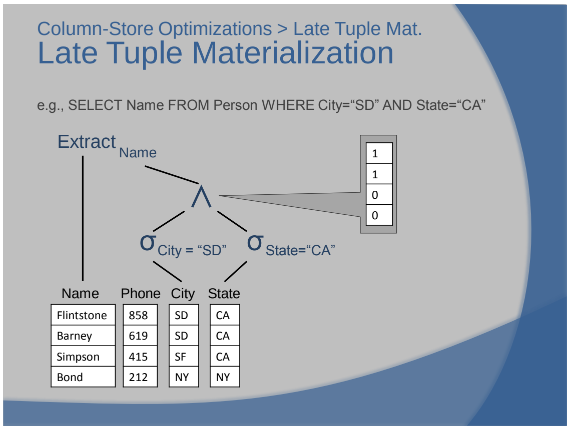
Column-Store Optimizations > Late Tuple Mat.

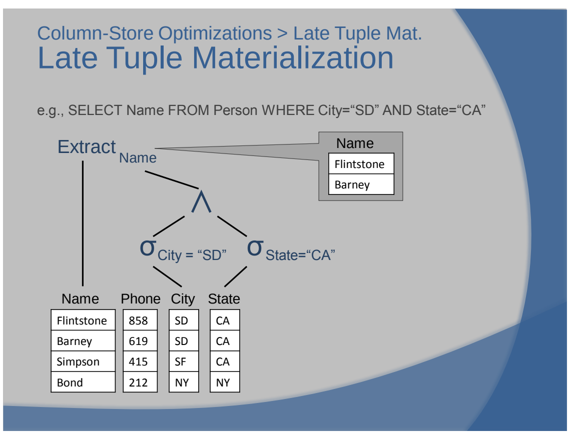
Late Tuple Materialization

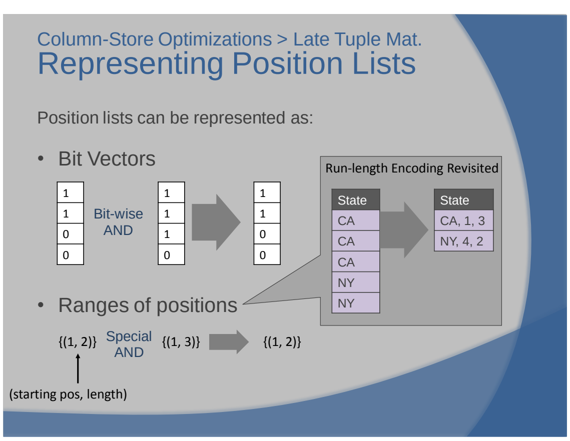
e.g., SELECT Name FROM Person WHERE City=“SD” AND State=“CA”



Name	Phone	City	State
Flintstone	858	SD	CA
Barney	619	SD	CA
Simpson	415	SF	CA
Bond	212	NY	NY


Column-Store Optimizations > Late Tuple Mat.
Late Tuple Materialization
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
Extract Name
∧
σ City = "SD"
σ State="CA"
1
1
0
0
Name
Phone
City
State
Flintstone
858
SD
CA
Barney
619
SD
CA
Simpson
415
SF
CA
Bond
212
NY
NY


Column-Store Optimizations > Late Tuple Mat.
Late Tuple Materialization
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
Extract Name
Name
Flintstone
Barney
∧
σ City = "SD"
σ State="CA"
Name
Phone
City
State
Flintstone
858
SD
CA
Barney
619
SD
CA
Simpson
415
SF
CA
Bond
212
NY
NY


Column-Store Optimizations > Late Tuple Mat.
Representing Position Lists
Position lists can be represented as:
• Bit Vectors
1
1
0
0
Bit-wise AND
1
1
1
0
1
1
0
0
Run-length Encoding Revisited
State
CA
CA
CA
NY
NY
State
CA, 1, 3
NY, 4, 2
• Ranges of positions
{(1, 2)}
Special AND
{(1, 3)}
{(1, 2)}
(starting pos, length)

Column-Store Optimizations > Late Tuple Mat.

Late Materialization Benefits

- Avoid materializing certain tuples
  since they may be filtered out before being materialized
  (Reminds of pushing selections down.)
- Avoid data decompression
  which has to be done when a tuple is materialized
- Leverage improved cache locality
  which exists when operating on a single column
- Leverage optimizations for fixed-width attributes
  which would not be possible if operating on the tuple level, since a tuple with at least one variable-width attribute becomes variable-width

Column-Store Optimizations

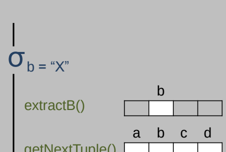
Block Iteration

Pass blocks of values between operators

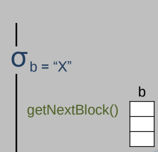
Column-Store Optimizations

Block Iteration

- Row Store
  - Pass single tuples between operators
  - Extract attribute value through function calls



- Column Store
  - Pass blocks of values between operators
  - No need for attribute extraction



Column-Store Optimizations

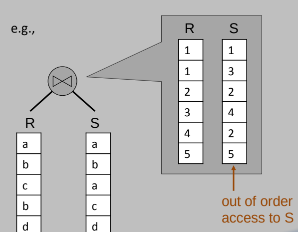
Invisible Join

Join Optimization for Star Schemas

Column Store Optimizations > Invisible Join

Joins & Late Tuple Mat.

- Late Tuple Materialization requires out of order access when we join tables



Column Store Optimizations

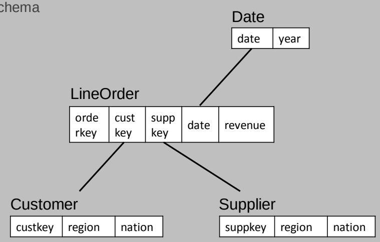
Invisible Join

- Optimization for joins on Star Schemas
  - Make sure that the fact table is accessed in order
  - Reduce the amount of data that are accessed out of order from the dimension tables
  - Reminiscent of bitmap intersection technique

Column Store Optimizations

Invisible Join Example

Schema



Example taken from VLDB 09 Tutorial by Harizopoulos, Abadi, Boncz

Column Store Optimizations

Invisible Join Example

Query

```
SELECT c_nation, s_nation, d_year, sum(lo_revenue) as revenue
FROM customer, lineorder, supplier, date
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;
```

Column Store Optimizations

Invisible Join Example

Data

Lineorder

order key	cust key	supp key	date	revenue
1	3	1	010197	43256
2	3	2	010197	33333
3	4	3	010297	12121
4	1	1	010297	23233
5	4	2	010297	45456
6	1	2	010397	43251
7	3	2	010397	34235

Supplier

suppkey	region	nation
1	Asia	Russia
2	Europe	Spain
3	Asia	Japan

Customer

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

Date

date	year
010197	1997
010297	1997
010397	1997

Column Store Optimizations

Invisible Join Example

Step 1: Apply selections on dimension tables

Supplier

suppkey	region	nation
1	Asia	Russia
2	Europe	Spain
3	Asia	Japan

$\sigma_{\text{region = “Asia”}}$ → Hashtable {1, 3}

Customer

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

$\sigma_{\text{region = “Asia”}}$ → Hashtable {1, 2, 3}

Date

date	year
010197	1997
010297	1997
010397	1997

$\sigma_{\text{year in [1992, 1997]}}$ → Hashtable {*}

Column Store Optimizations

Invisible Join Example

Step 2: Find fact table tuples satisfying all selections on dimension tables

supp key		cust key		date	
1	1	3	1	010197	1
2	0	3	1	010197	1
3	1	4	0	010297	1
1	1	1	1	010297	1
2	0	4	0	010297	1
2	0	1	1	010397	1
2	0	3	1	010397	1

\+ Hashtable {1, 3}

\+ Hashtable {1, 2, 3}

\+ Hashtable {010197, 010297, 010397}

Column Store Optimizations

Invisible Join Example

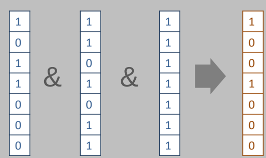
Step 2: Find fact table tuples satisfying *all* selections on dimension tables

1	1	1
0	1	1
1	0	1
1	1	1
0	0	1
0	1	1
0	1	1

Column Store Optimizations

Invisible Join Example

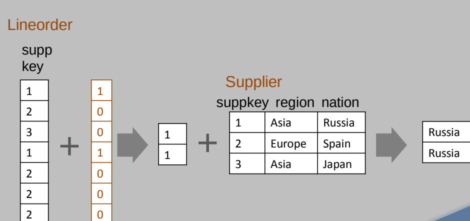
Step 2: Find fact table tuples satisfying *all* selections on dimension tables



Column Store Optimizations

Invisible Join Example

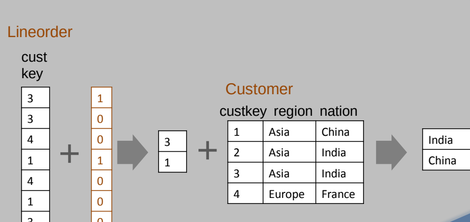
Step 3: Extract data from dimension tables



Column Store Optimizations

Invisible Join Example

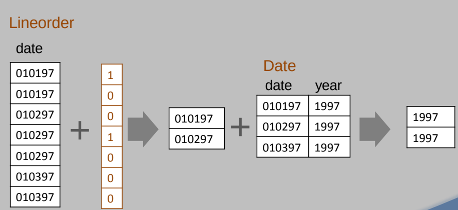
Step 3: Extract data from dimension tables



Column Store Optimizations

Invisible Join Example

Step 3: Extract data from dimension tables



Columm-Store Optimizations

Optimizations Summary

- Compression
- Late Tuple Materialization
- Block Iteration
- Invisible Join

Column-Store Optimizations

Comparing Optimizations

What is the speedup of each column-store optimization?

- Compression — 2x(avg)/10x(sorted)
- Late Tuple Materialization — 3x
- Block Iteration — 1.05-1.5x
- Invisible Join — 1.5-1.75x

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

Column-Store vs Row-Store

- How better is a column-store than a row-store?
  Heated debate: (Exaggerated) claims of performance up to 16,200x
- Can we simulate it in a row-store and get the performance benefits or does the row-store have to be internally modified?
  Another heated debate: Many papers on the topic
- Can we create a hybrid that will accommodate both transactional and analytics workloads?

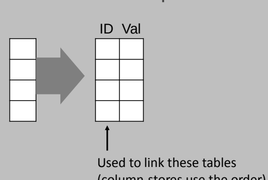
Column-Store Simulation

A column-store can be simulated in a row-store through:

- Vertical Partitioning
  Create one table per column
- Index-only Plans
  Create one index per column & use only indexes
- Materialized Views
  Create views of interest for given workload
- C-Table

Column-Store Simulation
Vertical Partitioning

- Create one table per column



Column-Store Simulation

Vertical Partitioning

e.g.,

Name	Phone	City	State
Flintstone	858	SD	CA
Barney	619	SD	CA
Simpson	415	SF	CA
Bond	212	NY	NY

ID	Name
1	Flintstone
2	Barney
3	Simpson
4	Bond

ID	Phone
1	858
2	619
3	415
4	212

ID	City
1	SD
2	SD
3	SF
4	NY

ID	State
1	CA
2	CA
3	CA
4	NY

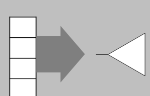
⊖ Overhead from storing tuple-ID

⊖ Excessive joins

Column-Store Simulation

Index-only Plans

- Create one index per column & use only indexes
  Data are kept as rows but are never accessed

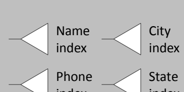


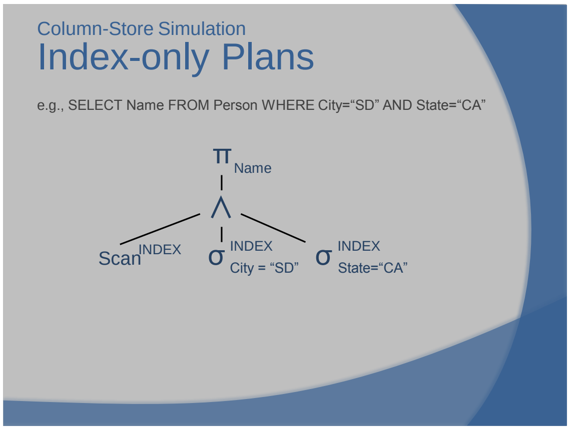
Column-Store Simulation

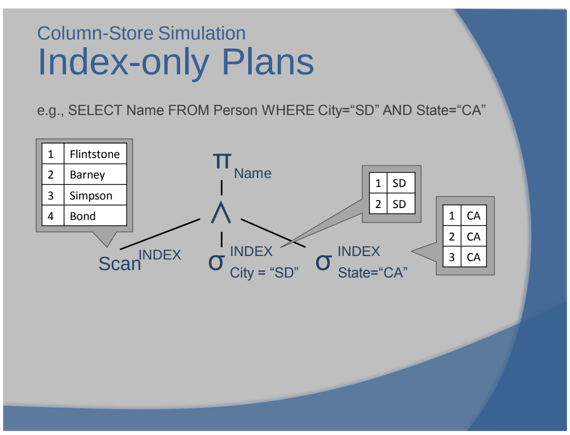
Index-only Plans

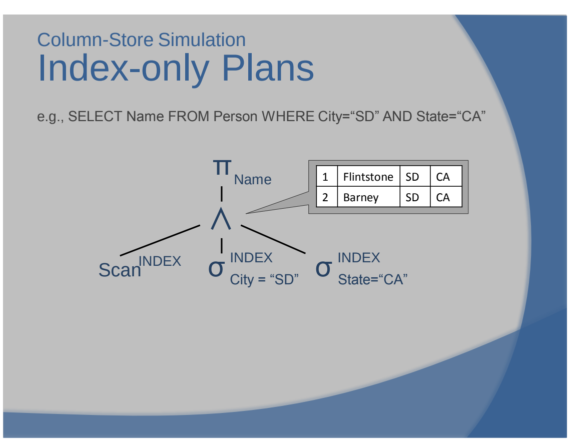
e.g.,

Name	Phone	City	State
Flintstone	858	SD	CA
Barney	619	SD	CA
Simpson	415	SF	CA
Bond	212	NY	NY




Column-Store Simulation
Index-only Plans
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
π Name
⋀
Scan INDEX
σ INDEX City = "SD"
σ INDEX State="CA"

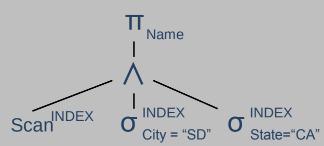

Column-Store Simulation
Index-only Plans
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
1 Flintstone
2 Barney
3 Simpson
4 Bond
π Name
⋀
1 SD
2 SD
1 CA
2 CA
3 CA
Scan INDEX
σ INDEX City = "SD"
σ INDEX State="CA"


Column-Store Simulation
Index-only Plans
e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"
π Name
1 Flintstone SD CA
2 Barney SD CA
⋀
Scan INDEX
σ INDEX City = "SD"
σ INDEX State="CA"

Column-Store Simulation

Index-only Plans

e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"



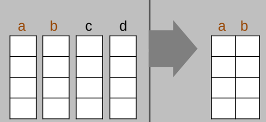
- ⊕ Avoids table access
- ⊖ Have to scan index for attributes wo predicate
  Create indexes with composite keys!

Column-Store Simulation

Materialized Views

- Create optimal view(s) for each query
  For each relation involved in the query, create a view including only columns needed to answer the query

$Q = \pi_a \sigma_{b='x'}(R)$



Column-Store Simulation

Materialized Views

e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"

Name	~~Phone~~	City	State
Flintstone	~~858~~	SD	CA
Barney	~~61[illegible]~~	SD	CA
Simpson	~~4[illegible]~~	SF	CA
Bond	~~212~~	NY	NY

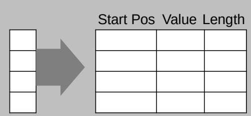
Name	City	State
Flintstone	SD	CA
Barney	SD	CA
Simpson	SF	CA
Bond	NY	NY

- ⊖ Requires workload knowledge

Column-Store Simulation

C-Table

- Extend vertical partitioning with run-length encoding
- Rewrite queries to operate on the compressed data



Column-Store Simulation

C-Table

e.g.,

State
CA
CA
CA
NY

Start Pos	State	Length
1	CA	3
4	NY	1

⊕ Pretty close to performance of native column-store

Back to our question:

Column-Store vs Row-Store

- Column-Stores have a definite advantage on analytic workflows…
- …but Row-Stores can be improved by taking lessons from them

Implementations

Open Source

monetdb C-Store

Commercial

SYBASE | An SAP Company INFOBRIGHT

Microsoft SQL Server 2012 VERTICA An HP Company

PARACCEL