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

SK Lee & Lucas Yoon

Row-Stores

Customer

ID	Name	Country	City
28495072	Jack Hamilton	USA	Seattle
78239842	Sarah Wilson	Finland	Helsinki
19389562	Jason Huang	China	Shanghai

```
INSERT INTO CUSTOMER(ID, Name, Country, City)
VALUES(37583719, 'Sam Kim', 'South Korea', 'Seoul')
```

Customer

ID	Name	Country	City
28495072	Jack Hamilton	USA	Seattle
78239842	Sarah Wilson	Finland	Helsinki
19389562	Jason Huang	China	Shanghai
37583719	Sam Kim	South Korea	Seoul

Column-Stores

Customer

ID	Name	Country	City
28495072	Jack Hamilton	USA	Seattle
78239842	Sarah Wilson	Finland	Helsinki
19389562	Jason Huang	China	Shanghai

Customer

ID	Name
28495072	Jack Hamilton
78239842	Sarah Wilson
19389562	Jason Huang

ID	Country
28495072	USA
78239842	Finland
19389562	China

ID	City
28495072	Seattle
78239842	Helsinki
19389562	Shanghai

Index

What is the problem this paper is solving?

1. Misconception
2. Lack of Systematic Comparison
3. Column Store Optimizations
4. Row Store Optimizations
5. Architectural Advantages

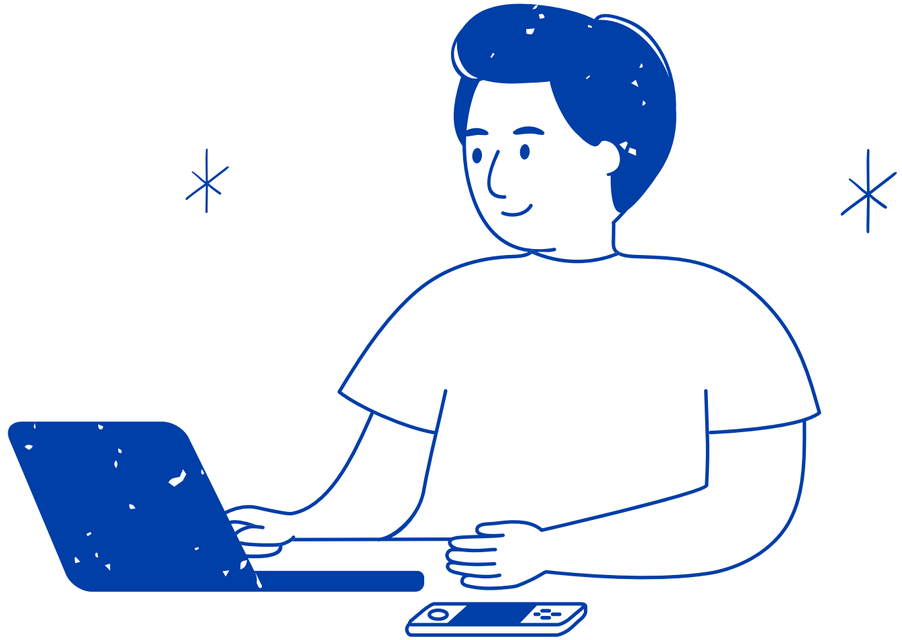
Why is it Challenging?

Challenge 1

- Architectural Differences

Challenge 2

- Balancing Performance



Why is it important?

Informed System Design

Hybrid Solutions

Future Development

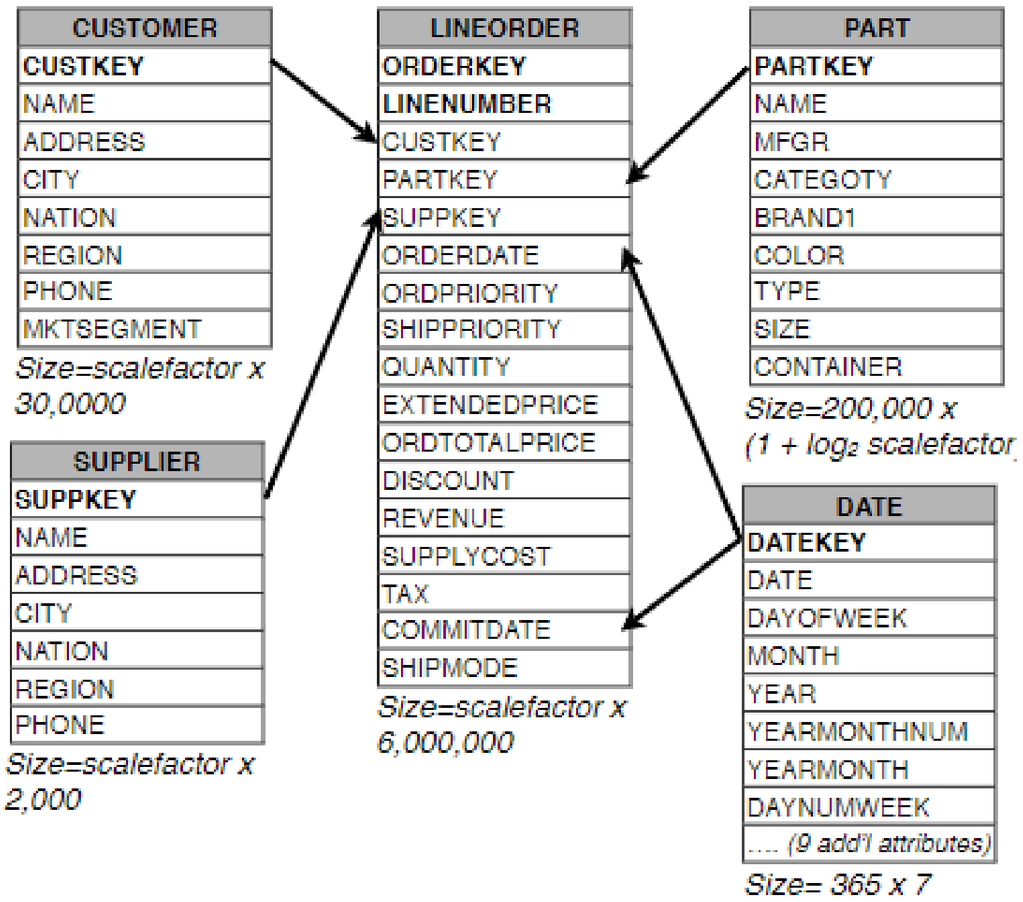
Solution Description

Compare Performance

Dissect Optimizations

Highlight Implications

Star Schema Benchmark



Flight 1: 1 dimension + DISCOUNT & QUANTITY

Flight 2: 2 dimensions & calculate revenue of product + region

Flight 3: 3 dimensions & analyze revenue in REGION + time

Flight 4: 3 dimensions & profit by YEAR, NATION, REGION

Row-Stores

Unnecessary reads due to row-by-row
Easy read and write
Expensive and low compression rate
Best suited for transaction system

Column-Stores

Read only relevant data
Slower read and write
Efficient when going through entire data set and high compression rate
Best suited for analytical system

Row-Stores

Customer

ID	Name	Country	City
28495072	Jack Hamilton	USA	Seattle
78239842	Sarah Wilson	Finland	Helsinki
19389562	Jason Huang	China	Shanghai

```
INSERT INTO CUSTOMER(ID, Name, Country, City)
VALUES(37583719, 'Sam Kim', 'South Korea', 'Seoul')
```

Customer

ID	Name	Country	City
28495072	Jack Hamilton	USA	Seattle
78239842	Sarah Wilson	Finland	Helsinki
19389562	Jason Huang	China	Shanghai
37583719	Sam Kim	South Korea	Seoul

Column-Stores

Customer

ID	Name	Country	City
28495072	Jack Hamilton	USA	Seattle
78239842	Sarah Wilson	Finland	Helsinki
19389562	Jason Huang	China	Shanghai

Customer

ID	Name
28495072	Jack Hamilton
78239842	Sarah Wilson
19389562	Jason Huang

ID	Country
28495072	USA
78239842	Finland
19389562	China

ID	City
28495072	Seattle
78239842	Helsinki
19389562	Shanghai

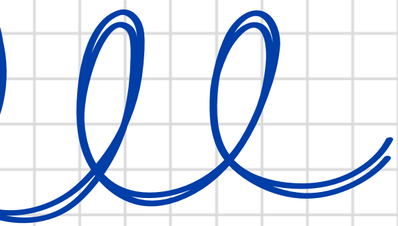
Row-orientation execution

1. Vertical partitioning
2. Index-only plans
3. Materialized views

Vertical partitioning

Employees Table

Name	Age	department
Bob	29	Marketing
Hugh	30	Marketing
John	32	Business



Bob | 1
Hugh | 2
John | 3

NameTable

29 | 1
30 | 2
32 | 3

AgeTable

Marketing | 1
Marketing | 2
Business | 3

DeptTable

Vertical partitioning

Employees Table

NameTable	AgeTable	DeptTable
Bob | 1	29 | 1	Marketing | 1
Hugh | 2	30 | 2	Marketing | 2
John | 3	32 | 3	Business | 3

```
SELECT *
FROM Employees
WHERE name = “Bob”
```

```
SELECT name
FROM Employees
WHERE name = “Bob”
```

Vertical partitioning

Pros:

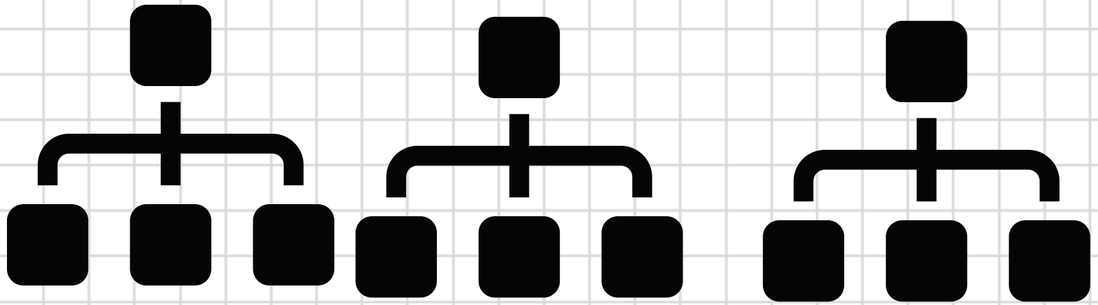
1. **Can emulate column stores**

Cons:

1. **It will need a join when different column data needs to be used**
2. **It requires the position attribute which wastes disc space.**

Index-only plans

Name	Age	department
Bob	29	Marketing
Hugh	30	Marketing
John	32	Business



Index-only plans

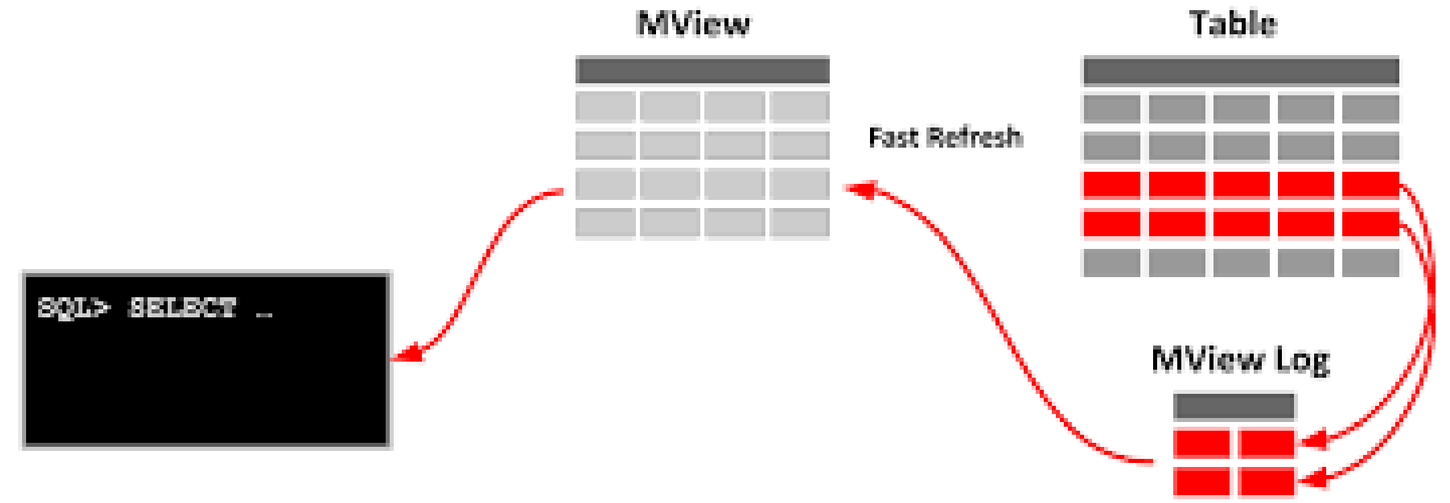
Pros:

1. **Minimizes space and performance overhead associated with vertical partitioning**

Cons:

1. **Expensive writes**
2. **Indexes still take up space**

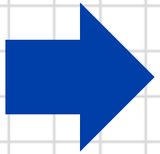
Materialized views



Materialized views

Name	Age	department
Bob	29	Marketing
Hugh	30	Marketing
John	32	Business

Save view



Age	department
29	Marketing
30	Marketing
32	Business

```
SELECT AVG(Age)
FROM Employees
WHERE department = “Marketing”
```

Materialized Views

Pros:

1. **Guarantees improved performance**

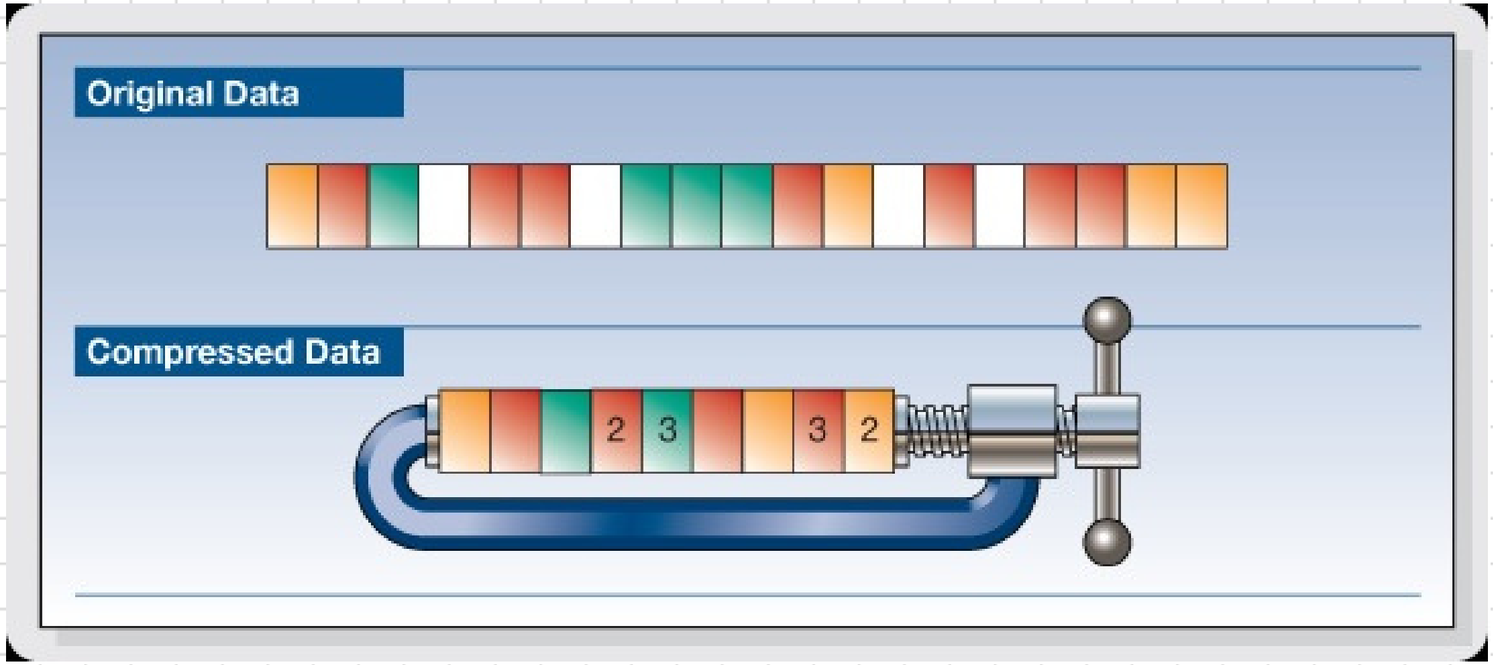
Cons:

1. **The query workload has to be known in advance to take advantage**

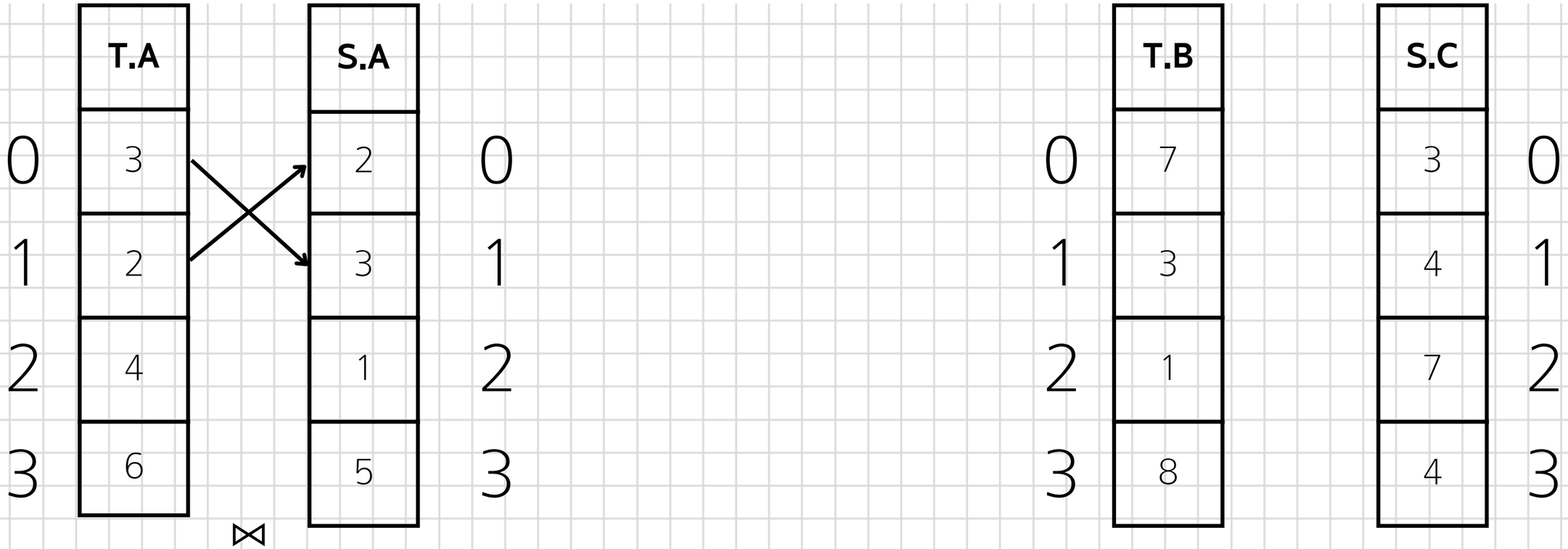
Column-orientation execution

1. Compression
2. Late materialization
3. Block iteration
4. Invisible join

Compression



Late materialization

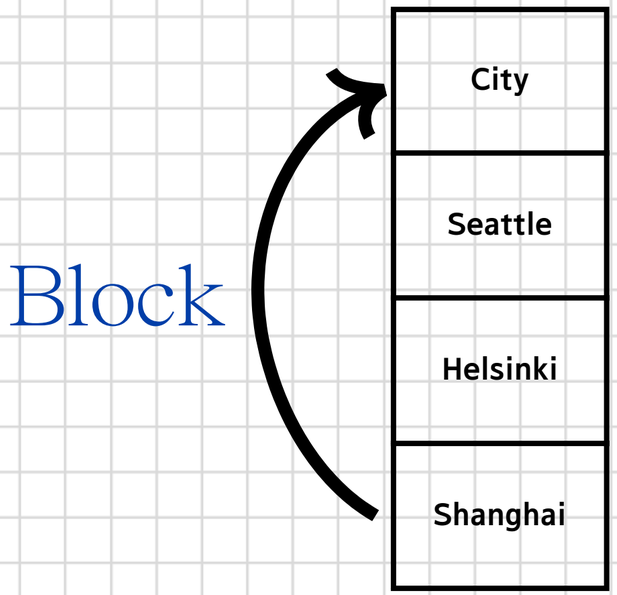


Pos(T.A)	Pos(S.A)
0	1
1	0

T.B	S.C
7	4
3	3

Block iteration

ID	Name	Country	City
28495072	Jack Hamilton	USA	Seattle
78239842	Sarah Wilson	Finland	Helsinki
19389562	Jason Huang	China	Shanghai



Invisible join

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

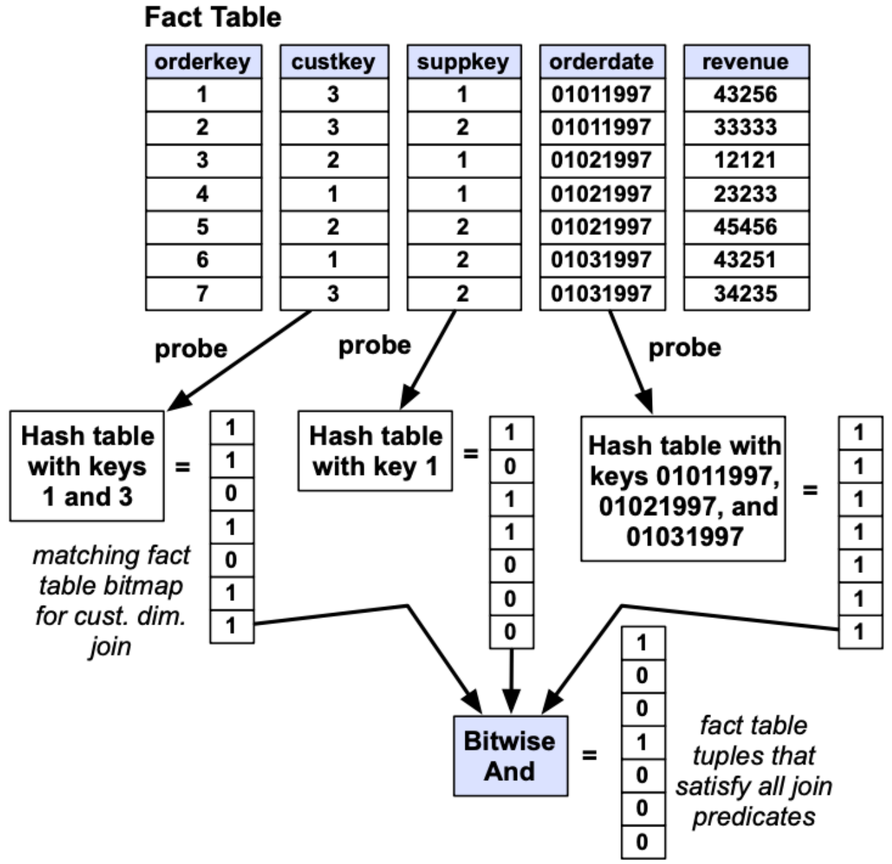
suppkey	region	nation	...
1	Asia	Russia	...
2	Europe	Spain	...

→ Hash table with key 1

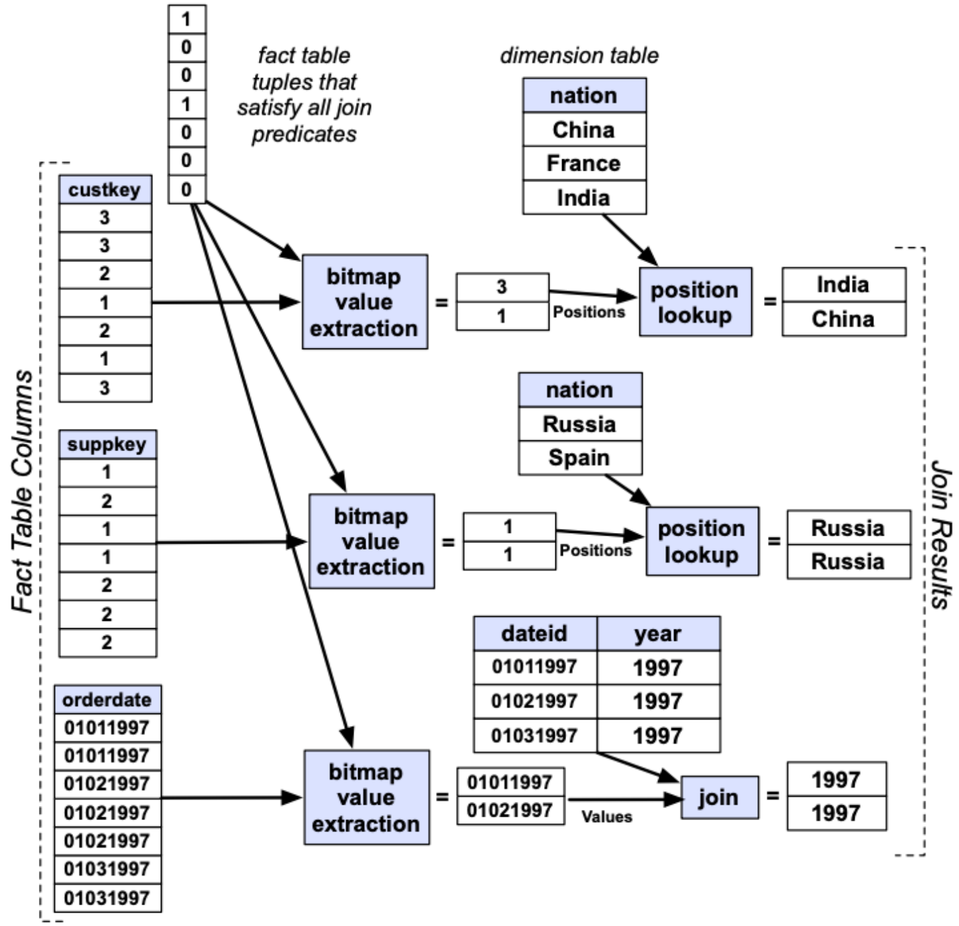
Apply `year in [1992,1997]` on Date table

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

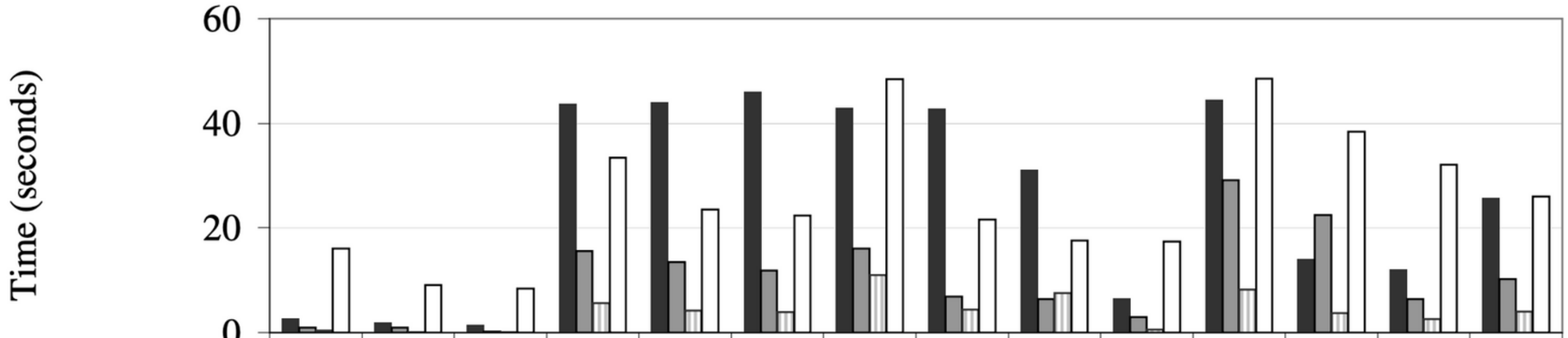
→ Hash table with keys 01011997, 01021997, and 01031997



Invisible join



Experiments



	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

Figure 5: Baseline performance of C-Store "CS" and System X "RS", compared with materialized view cases on the same systems.

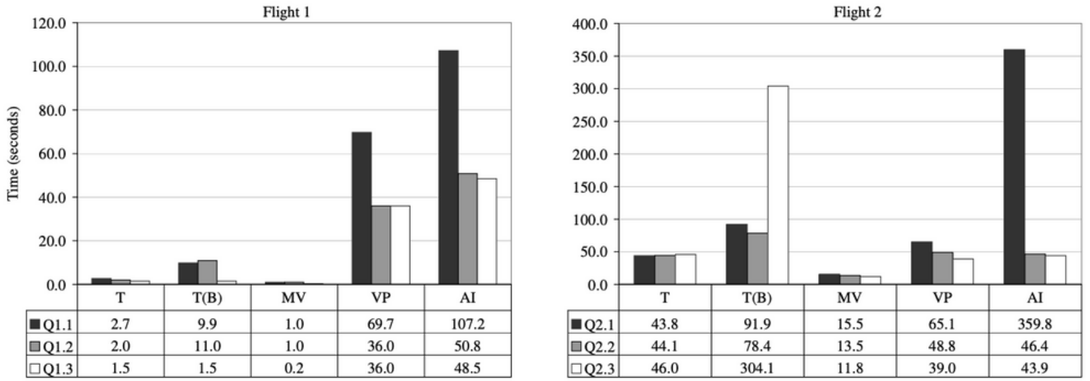
RS: Traditional Row Store (System X)

RS (MV): Row Store with Materialized Views

CS: Traditional Column Store (C-Store)

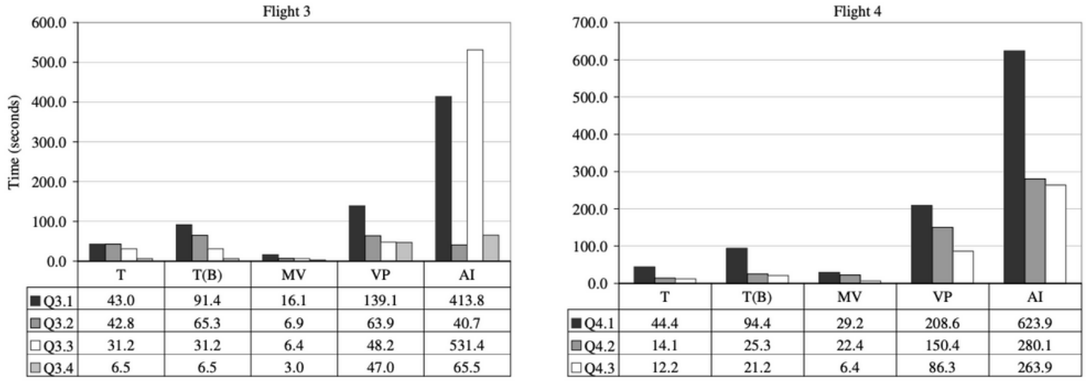
CS (Row-MV): Row Store with Materialized Views (C-Store)

Experiments



	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

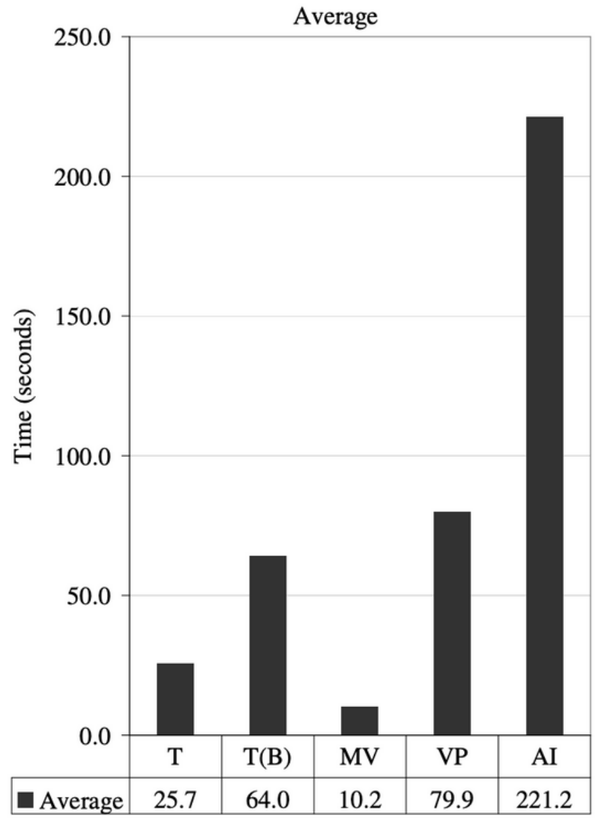
	T	T(B)	MV	VP	AI
Q2.1	43.8	91.9	15.5	65.1	359.8
Q2.2	44.1	78.4	13.5	48.8	46.4
Q2.3	46.0	304.1	11.8	39.0	43.9



	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

(a)

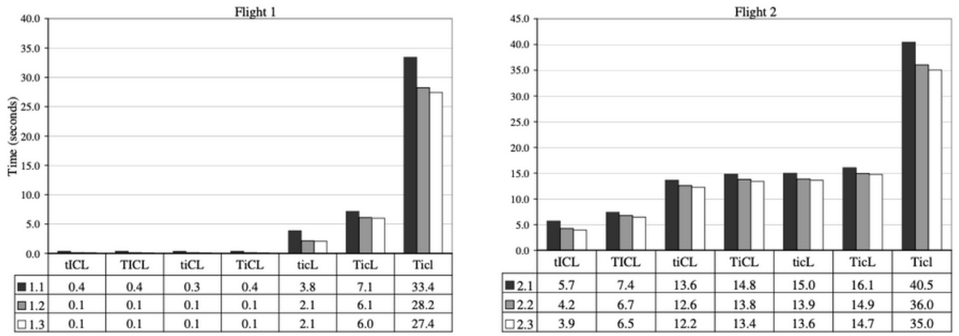


	T	T(B)	MV	VP	AI
Average	25.7	64.0	10.2	79.9	221.2

(b)

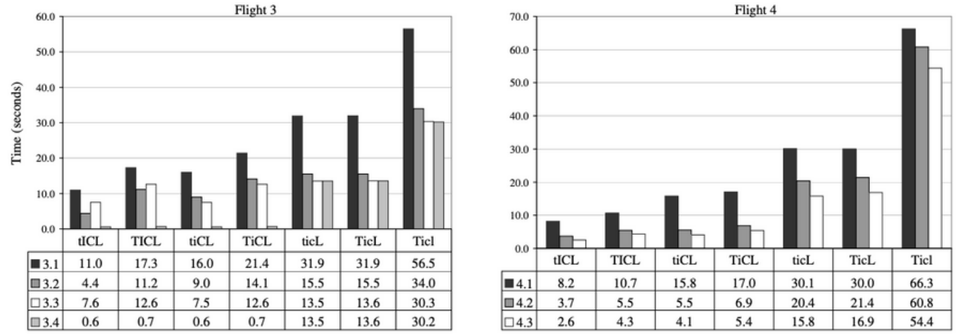
Figure 6: (a) Performance numbers for different variants of the row-store by query ight. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all queries.

Experiments



	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
1.1	0.4	0.4	0.3	0.4	3.8	7.1	33.4
1.2	0.1	0.1	0.1	0.1	2.1	6.1	28.2
1.3	0.1	0.1	0.1	0.1	2.1	6.0	27.4

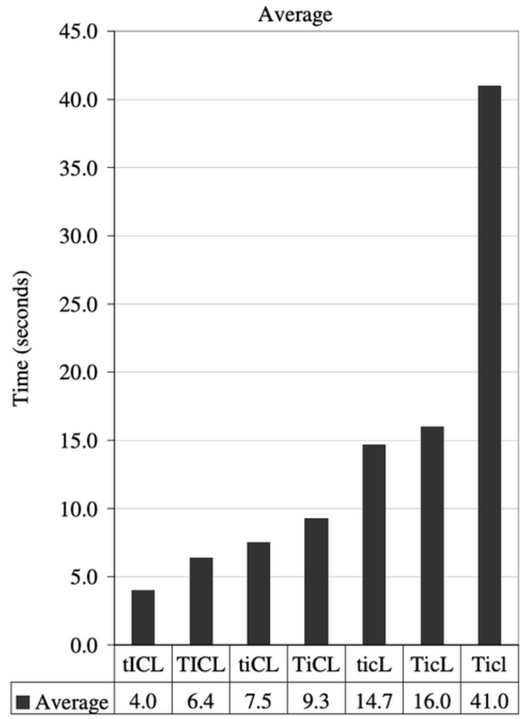
	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
2.1	5.7	7.4	13.6	14.8	15.0	16.1	40.5
2.2	4.2	6.7	12.6	13.8	13.9	14.9	36.0
2.3	3.9	6.5	12.2	13.4	13.6	14.7	35.0



	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
3.1	11.0	17.3	16.0	21.4	31.9	31.9	56.5
3.2	4.4	11.2	9.0	14.1	15.5	15.5	34.0
3.3	7.6	12.6	7.5	12.6	13.5	13.6	30.3
3.4	0.6	0.7	0.6	0.7	13.5	13.6	30.2

	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
4.1	8.2	10.7	15.8	17.0	30.1	30.0	66.3
4.2	3.7	5.5	5.5	6.9	20.4	21.4	60.8
4.3	2.6	4.3	4.1	5.4	15.8	16.9	54.4

(a)

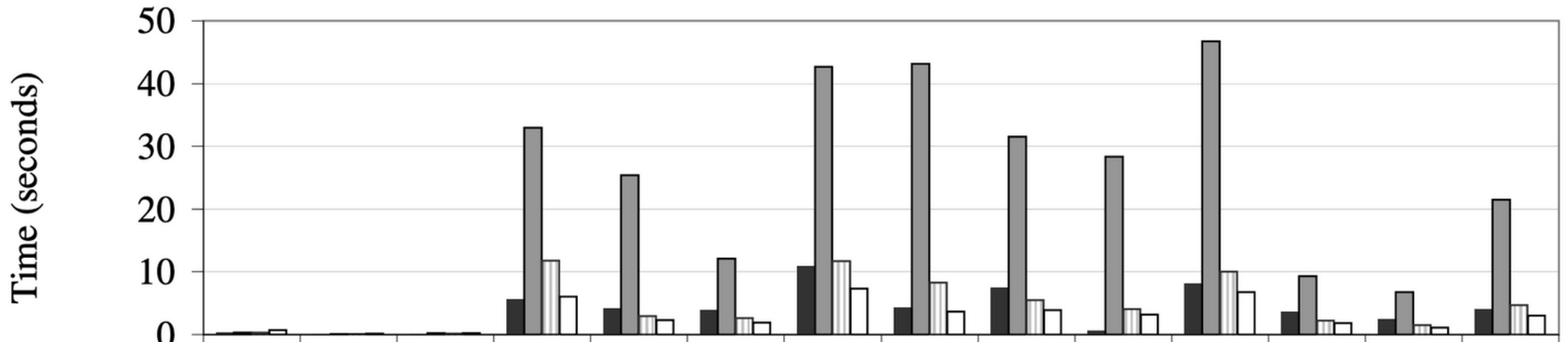


	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
Average	4.0	6.4	7.5	9.3	14.7	16.0	41.0

(b)

Figure 7: (a) Performance numbers for C-Store by query ight with various optimizations removed. The four letter code indicates the C-Store con guration: 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. (b) Average performance numbers for C-Store across all queries.

Experiments



	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
Base	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
PJ, No C	0.4	0.1	0.2	32.9	25.4	12.1	42.7	43.1	31.6	28.4	46.8	9.3	6.8	21.5
PJ, Int C	0.3	0.1	0.1	11.8	3.0	2.6	11.7	8.3	5.5	4.1	10.0	2.2	1.5	4.7
PJ, Max C	0.7	0.2	0.2	6.1	2.3	1.9	7.3	3.6	3.9	3.2	6.8	1.8	1.1	3.0

Figure 8: Comparison of performance of baseline C-Store on the original SSBM schema with a denormalized version of the schema. Denormalized columns are either not compressed ("PJ, No C"), dictionary compressed into integers ("PJ, Int C"), or compressed as much as possible ("PJ, Max C").

Related Work

Monet DB
Monet DB/X100

- Pioneers of the design of modern column-oriented database system
- Contributed to superior CPU and cache performance and reduce I/O compared to row-store

C-Store

- Optimization for direct operation on compressed data
- Dramatic performance improvements on warehouse workloads

Fractured Mirrors

- Hybrid of row/column-stores
- Updates using row-store and reads using column-store

Shore

- Halverson et al worked with Shore to compare against unmodified version
- “Super tuples” optimization made vertically partitioned database competitive to column-stores

What Did We Learn?

Differences in Architecture

Need for Changes in Row-Stores

Future Directions

In Class Discussion

How can we further improve a row-store & column-store?

In Class Discussion

Do you think row-store can follow the performance rate of column-store in the future?

Q&A