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

Authours: Daniel Abadi, Sammuel Madden, Nabil Hachem

Presented By: Aaron Sarson

Introduction

- Column-Store database systems have emerged in recent years
    - MonetDB
    - **C-Store**
- It is commonly understood that column-stores offer superior performance on I/O intensive tasks
    - However, literature fails to address if these performance gains can be achieved in row-store DBMS
- **RQ1.** This work investigates if row-store DBMS can achieve similar gains if the physical architecture emulates that of column-stores
- **RQ2.** The authours look to discover which features/attributes of column-stores DBMS contributes most to the performance advantage over row-stores

Row-Oriented Execution

Emulating Column-Stores in Row-Oriented DBMS

Authours outline three alternative physical designs:

- Vertical Partitioning
- Index-Only Plans
- Materialized Views

Vertical Partitioning

ID	A	B	C
1	X	X	X
2	X	X	X
3	X	X	X

Pos	A
1	X
2	X
3	X

Pos	B
1	X
2	X
3	X

Pos	C
1	X
2	X
3	X

Queries perform joins on the `Pos`tion attribute when retrieving multiple attributes of a single entity/row

Cons of this approach:

1. Position attribute on every column
2. Row-stores have large headers associated with each tuple

Wasted memory and/or bandwidth

Index-Only Plans

- Base relations are stored in standard row-store format
  - **Addition:** Unclustured B+ tree index on every column (**ALL** tables)
- Through this approach only access to indices is required, and not the actual data
  - Reduce I/O → No disk access
- **Cons of this approach:**
  1. Predicate-less columns, require index to be scanned to extract values
     - This is slower than scanning a heap file

Materialized Views

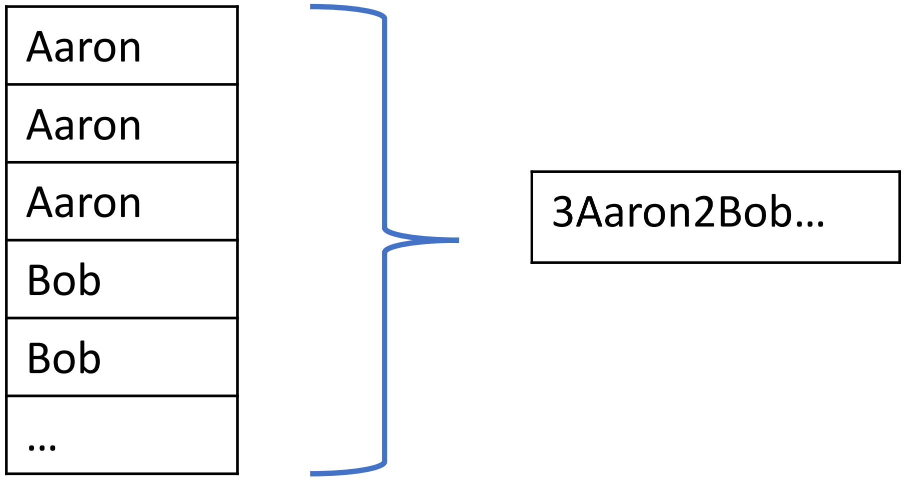
- “Optimal set of materialized views for every query flight”
  - optimal view contains only the required columns
- Pre-computed dataset
  - Allows access to just the data needed to answer a query
- **Advantages** of this approach:
  - No need to store record-ids (index only) or position (vertical partition)
  - Only stores tuple headers once

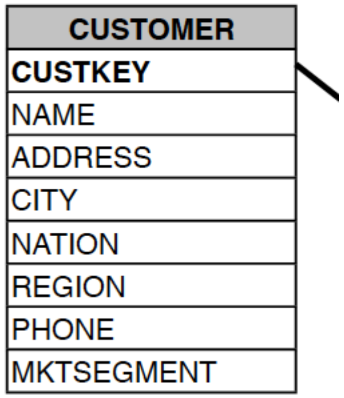
Column-Oriented Execution

Compression

“Intuitively, data stored in columns is more compressible than data stored in rows”

- Column-Oriented Databases → **low information entropy**
  - Compression algorithms perform better under this condition
- Data sorted on a particular column is super-compressible
  - Can be **run-length encoded**





Compression

$$Compression\ Ratio = \frac{Uncompressed}{Compressed}$$

- Produces a larger compression ratio
  - Memory Gains
    - Reducing number of disks
    - Power consumption
  - Performance Gains
    - Reduced I/O time → Smaller reads
    - If query executor can operator on compressed data performance can be improved further

3Aaron2Bob…

- Compression differences are largest in row vs column-stores when:
  1. Column data is sorted
  2. Repeating values are present (**runs**)

Late Materialization

- Column-stores have **entity** information distributed throughout a disk(s)
- Row-stores have **entity** information group together (single record)
- **Problem?**
  - Most queries access multiple attributes of an entity (i.e., name, address)
  - Many database output standards (i.e. JDBC, ODBC) work at an entity-at-a-time
- **Solution?**
  - At some point, query plans must combine data from multiple columns into rows representing an entity
    - Depending on when this is done → "Early Materialization" or "Late Materialization"

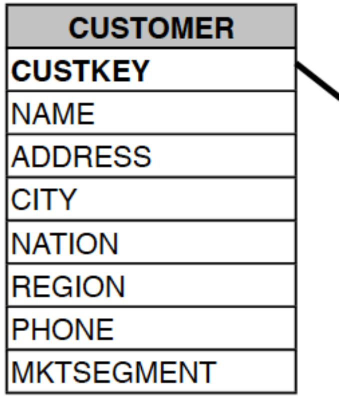
- **Early Materialization:**
  - Constructs entity using relevant columns and then applies row-store operators

Aaron	Sam	Jennifer	Lucy	Alex	Luke
0	1	2	3	4	5

Canada	England	Canada	France	Italy	Canada
0	1	2	3	4	5

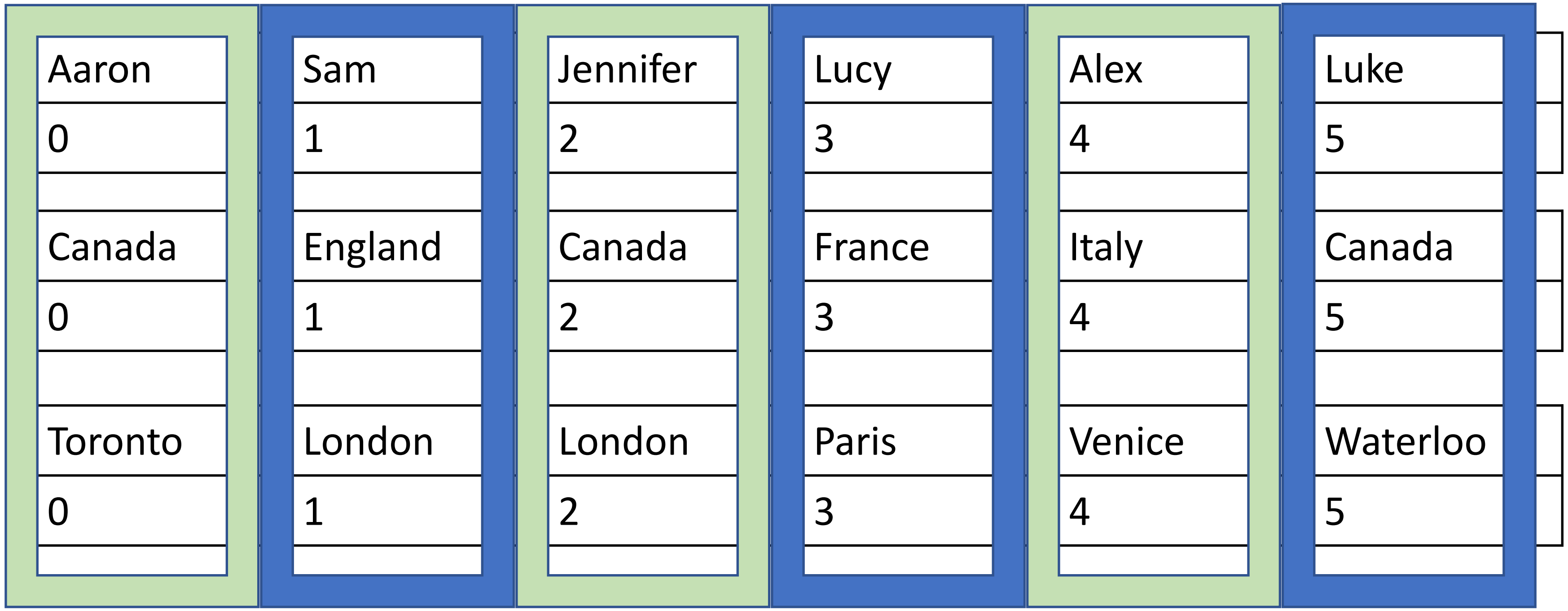
Toronto	London	London	Paris	Venice	Waterloo
0	1	2	3	4	5

SELECT Name, City FROM Customer WHERE Nation = “Canada”



CUSTOMER
CUSTKEY
NAME
ADDRESS
CITY
NATION
REGION
PHONE
MKTSEGMENT

- **Early Materialization:**
  - Constructs entity using relevant columns and then applies row-store operators



Aaron, **Canada**, Toronto

Sam, England, London

Jennifer, **Canada,** London

Lucy, France, Paris

Alex, Italy, Venice

Luke, **Canada**, Waterloo

SELECT Name, City FROM Customer WHERE Nation = “Canada”

Aaron, **Canada**, Toronto

Jennifer, **Canada,** London

Luke, **Canada**, Waterloo

- **Late Materialization:**
  - Operates on columns

Aaron	Sam	Jennifer	Lucy	Alex	Luke
0	1	2	3	4	5

Canada	England	Canada	France	Italy	Canada
0	1	2	3	4	5
			0	2	5

Toronto	London	London	Paris	Venice	Waterloo
0	1	2	3	4	5

SELECT Name, City FROM Customer WHERE Nation = “Canada”

CUSTOMER
CUSTKEY
NAME
ADDRESS
CITY
NATION
REGION
PHONE
MKTSEGMENT

- **Late Materialization:**
  - Operates on columns

Aaron	Sam	Jennifer	Lucy	Alex	Luke
0	1	2	3	4	5

2

AND

Canada	England	Canada	France	Italy	Canada
0	1	2	3	4	5
			0	2	5

Toronto	London	London	Paris	Venice	Waterloo
0	1	2	3	4	5
				1	2

SELECT Name FROM Customer WHERE Nation = “Canada”
AND City = “London”

CUSTOMER
CUSTKEY
NAME
ADDRESS
CITY
NATION
REGION
PHONE
MKTSEGMENT

- **Late Materialization:**
  - Operates on columns

Aaron	Sam	Jennifer	Lucy	Alex	Luke
0	1	2	3	4	5

2

AND

Canada	England	Canada	France	Italy	Canada
0	1	2	3	4	5
			0	2	5

Toronto	London	London	Paris	Venice	Waterloo
0	1	2	3	4	5
				1	2

SELECT Name FROM Customer WHERE Nation = "Canada"
AND City = "London"

CUSTOMER
CUSTKEY
NAME
ADDRESS
CITY
NATION
REGION
PHONE
MKTSEGMENT

Late Materialization - Advantages

1. Selection and aggregation operators tend to reduce the number of tuples which need to be constructed
   - Think of the number tuples we needed to construct in **early materialization**

2. Data compressed using column-oriented compression methods must be decompressed during the tuple construction process
   - **Early materialization** constructs many tuples at start
   - **Late materialization** constructs few tuples at end

3. Cache performance improved
   - Cache line is populated with related data (**High data locality** of column-stores)

A Typical Query Structure

```
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;
```

Restrict the set of tuples using selection predicates on 1+ dimension tables

Next, perform aggregation often grouping on other table attributes

Traditional Query Plan:

- Perform joins in order of predicate selectivity

```
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;
```

Assuming `c.region = ASIA` is the most selective

1. Join `customer` and `lineorder`
2. Filter `lineorder` → customers from `ASIA` remain
3. `nation` of these customers is added to `customer-order`

Traditional Query Plan:

```
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;
```

1. Join `supplier` and `lineorder`
2. Filter `lineorder` → suppliers from `ASIA` remain
3. `nation` of these supliers is added to `customer-order`

Traditional Query Plan:

```
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;
```

1. Join `dworder` and `lineorder`
2. Filter `lineorder` → customers who ordered between the `years` `1992` and `1997` remain
3. `year` of these customers ordered is added to `customer-order`

Results of joins to are finally `GROUP`ed and aggregated (i.e. `sum`)

Late Materialized Query Plan:

- Predicate is applied on column-store

```
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;
```

1. Filter `c.region` → customers from `ASIA` remain
2. `CUSTKEY` of these customers is extracted
3. These `CUSTKEY`s are joined with `CUSTKEY`s from the fact table.
   - Resulting in **2** position lists
     - 1 sorted (fact table) and 1 unsorted (dimension table)
     - Lists indicate which tuples pass the predicate (i.e. `c.region = ASIA`)

4. Extract values from out-of-order positions (i.e. `c.nation`) alongside the values from in-order set of positions for the fact table (i.e. `lo.suppkey`, `lo.orderdate`, and `lo.revenue`)

Late Materialized Query Plan:

- Predicate is applied on column-store

```
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;
```

1. Filter `s.region` → customers from `ASIA` remain
2. `SUPPKEY` of these suppliers is extracted
3. These `SUPPKEY`s are joined with `SUPPKEY`s from the fact table.
   - Resulting in **2** position lists
     - 1 sorted (fact table) and 1 unsorted (dimension table)
     - Lists indicate which tuples pass the predicate (i.e. `s.region = ASIA`)

4. Extract values from out-of-order positions (i.e. `s.nation`) alongside the values from in-order set of positions for the fact table (i.e. `lo.custkey`, `lo.orderdate`, and `lo.revenue`)

Repeat once more for `d.year` predicate

An Alternative Plan – Invisible Join

- Late materialized join that minimizes out-of-order value extraction
  - **How is this accomplished?**
    - Rewriting joins as predicates on foreign key columns in fact table

PHASE 01: Constructing Hash Tables

- Apply each predicate to dimension table → list of keys satisfying predicate
- Construct hash table

Apply `region = 'Asia'` on Customer table

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

Hash tabl
with keys
1 and 3

Apply `region = 'Asia'` on Supplier table

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

Hash tabl
with key 1

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

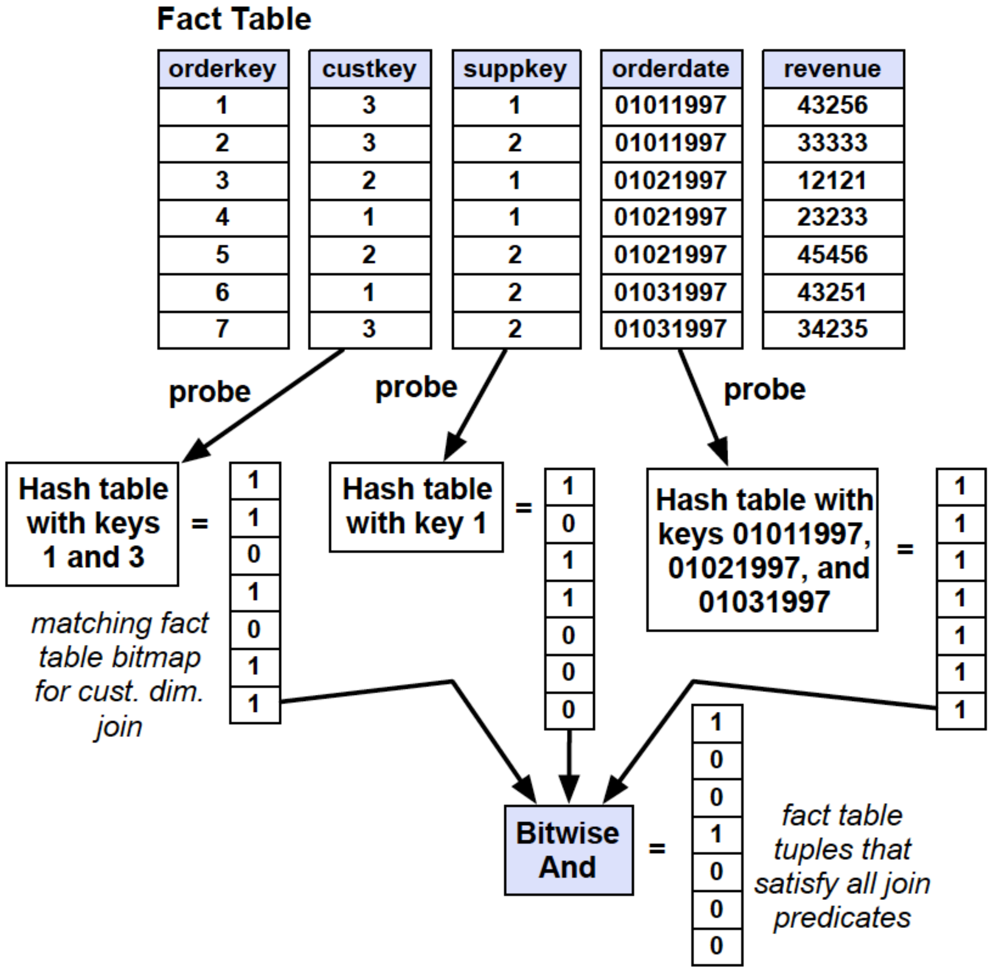
dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

Hash table with
keys 01011997,
01021997, and
01031997

An Alternative Plan – Invisible Join

PHASE 02: Extract Fact Table Records

- Use hash tables to locate records in fact table that satisfy predicate
- Probe hash table with each value in foreign key column
- Intersect positions lists → records which satisfy ALL predicates

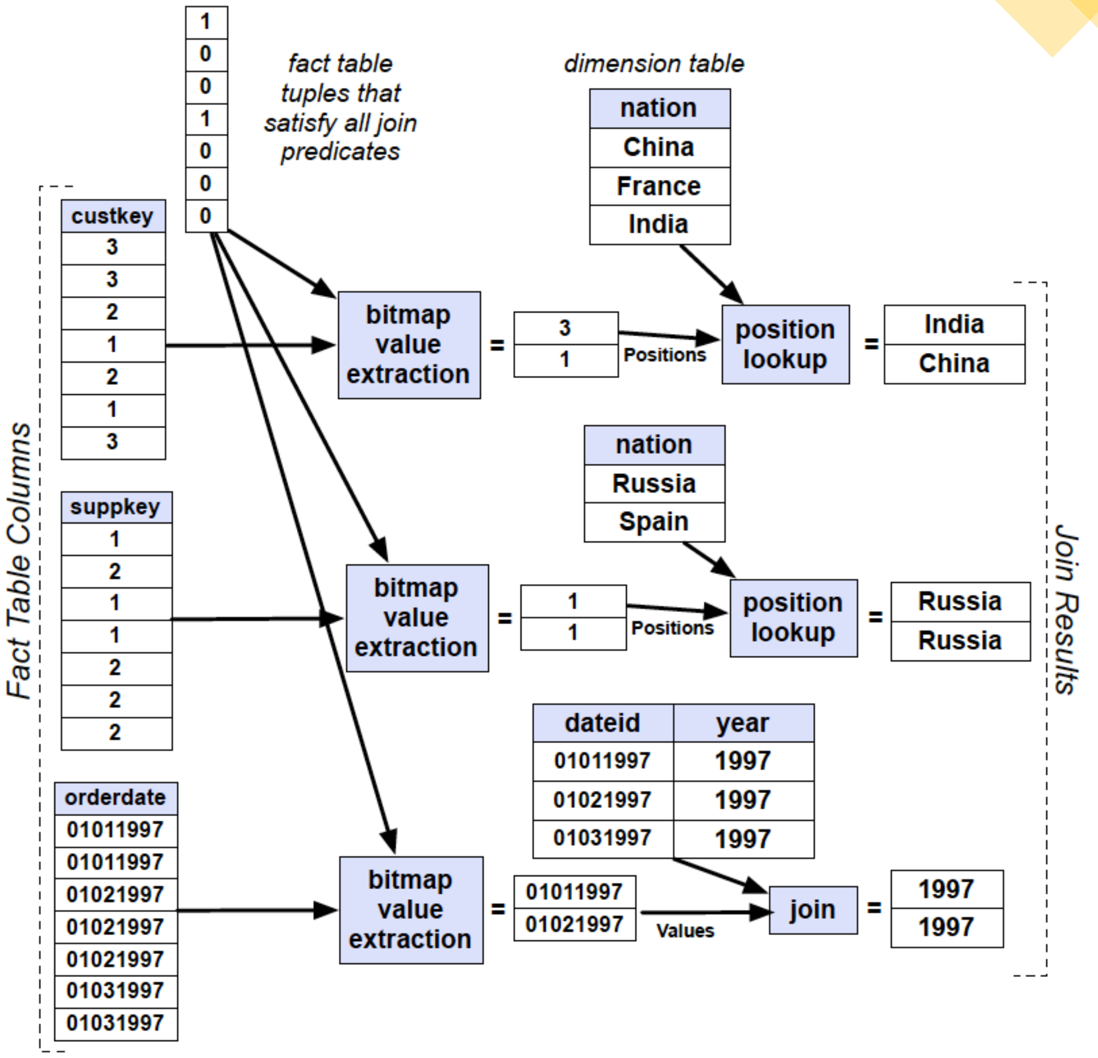


An Alternative Plan – Invisible Join

PHASE 03: Extract Dimension Table Records & Execute Query

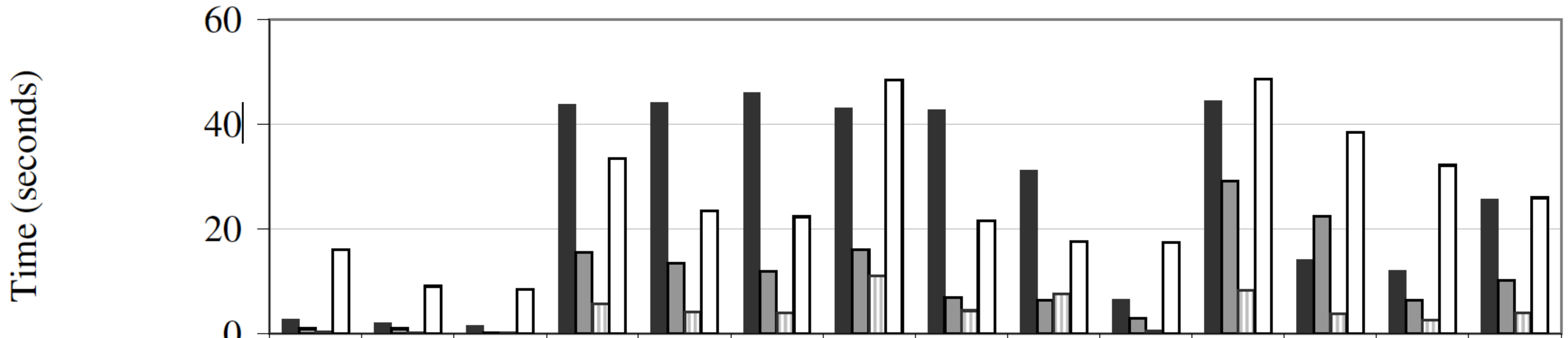
- Apply list of satisfying positions to fact tables
  - Identify foreign key references in the appropriate dimension table
  - Extract corresponding values

Note: “If dimension table key is sorted, contiguous list of identifiers starting from 1 [..], then the foreign key actually represents the position of desired tuple in dimension table”



Experiments

Motivation: C-Store vs System X - SSBM

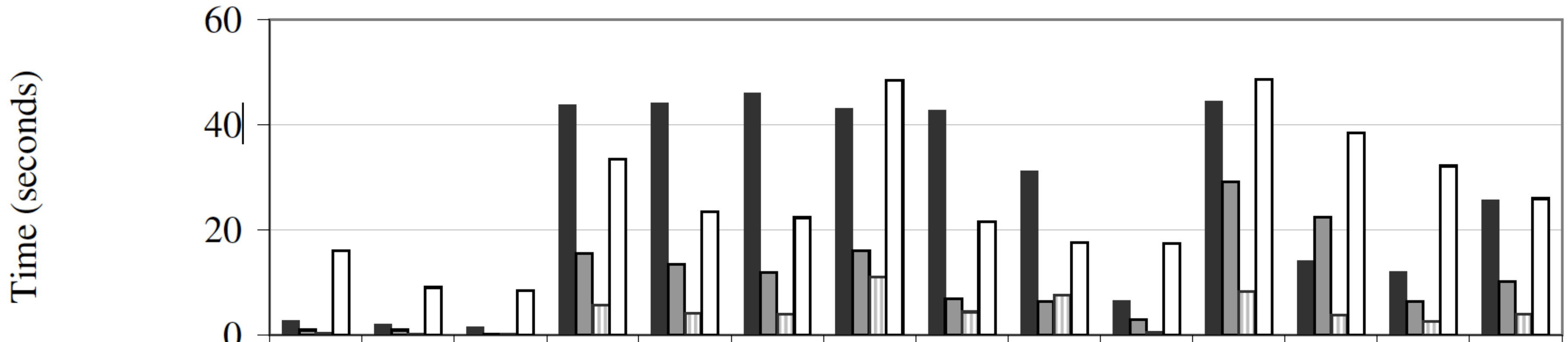


	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	AVG
■ RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
■ RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
▥ CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
□ CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

x 6.425

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

Motivation: C-Store vs System X - SSBM



	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

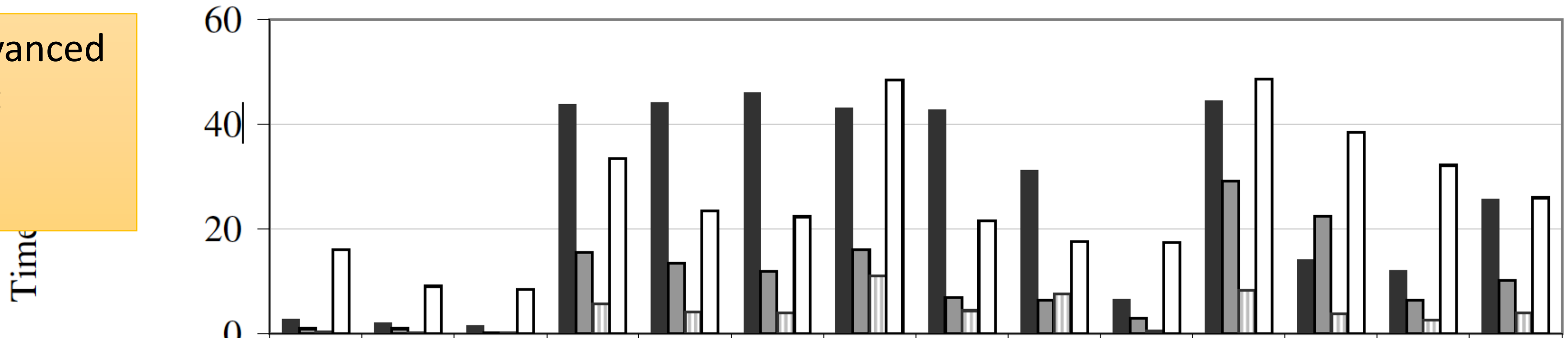
x 2.55

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

Motivation: C-Store vs System X - SSBM

System X supports advanced performance features:

- Partitioning
- Multi-threading

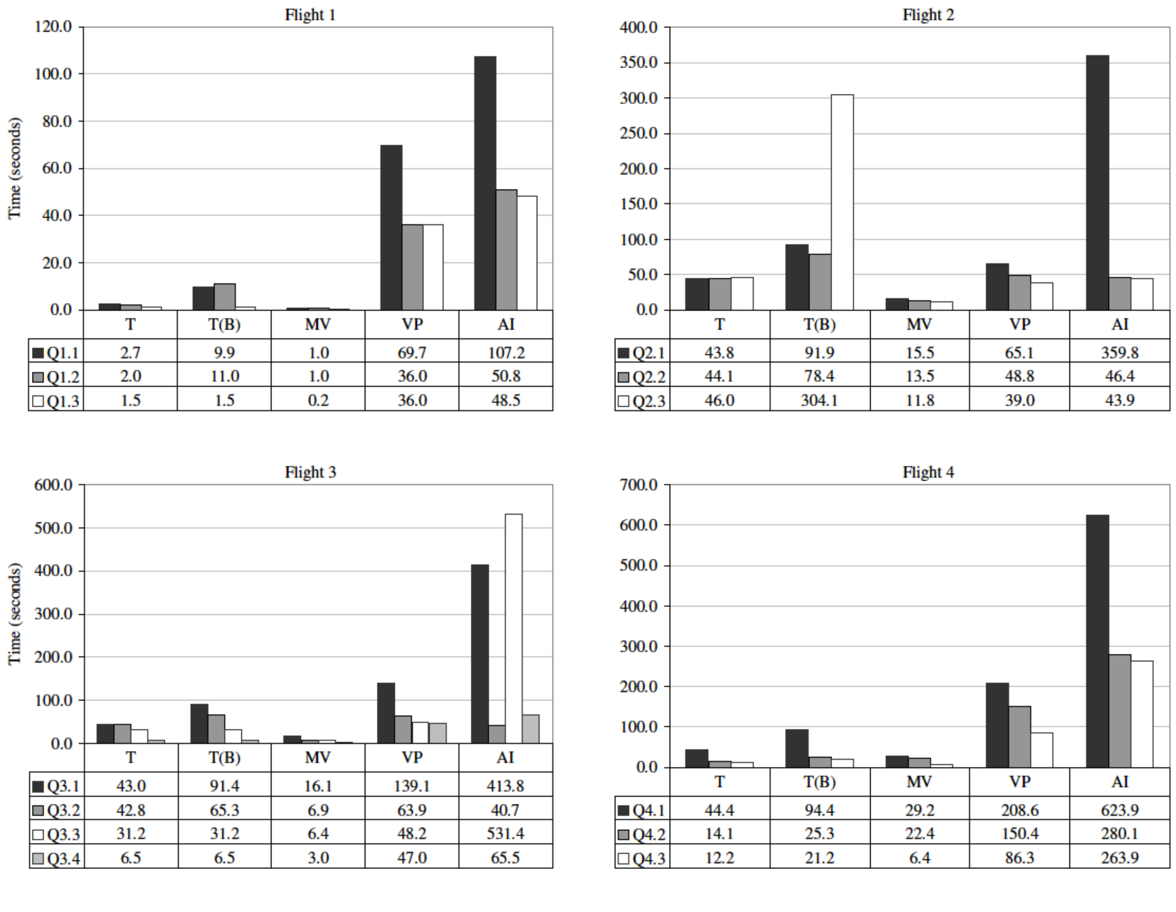


	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	AVG
RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

x 2.54

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

Column-Store Simulation in a Row-Store



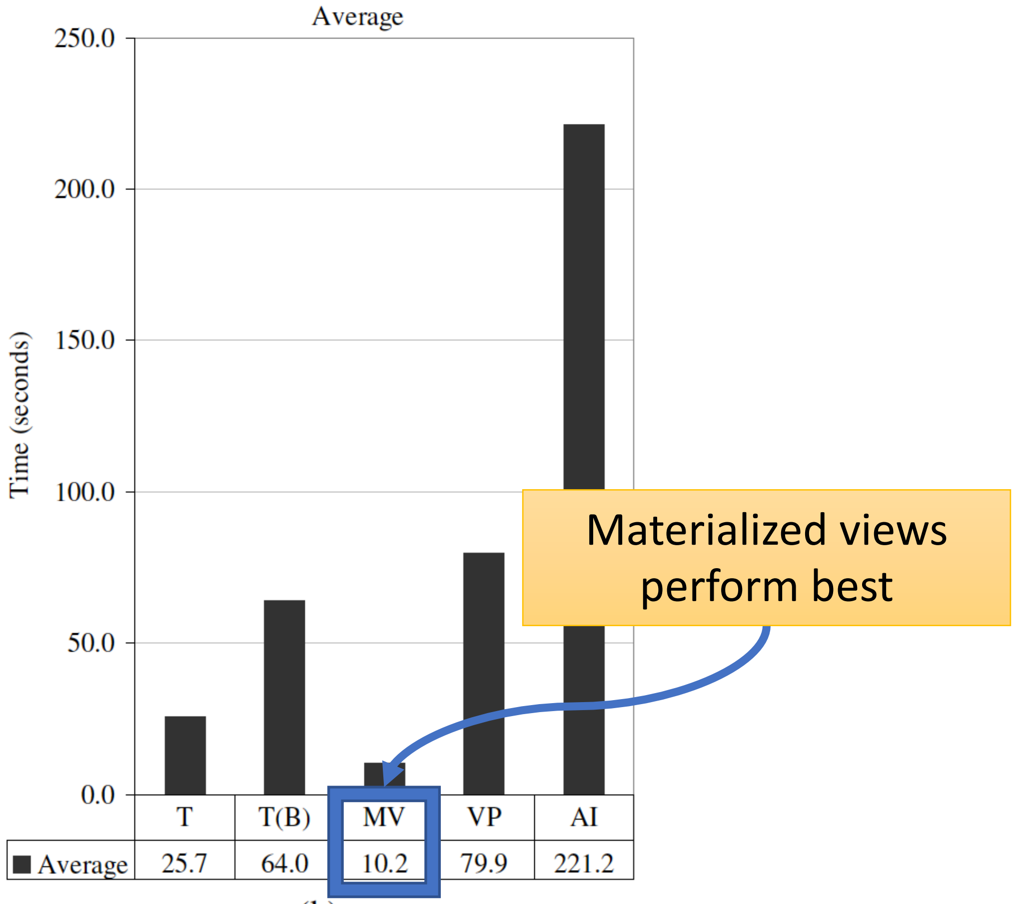
Flight 1	T	T(B)	MV	VP	AI
Q1.1	2.7	9.9	1.0	69.7	107.2
Q1.2	2.0	11.0	1.0	36.0	50.8
Q1.3	1.5	1.5	0.2	36.0	48.5

Flight 2	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

Flight 3	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

Flight 4	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)

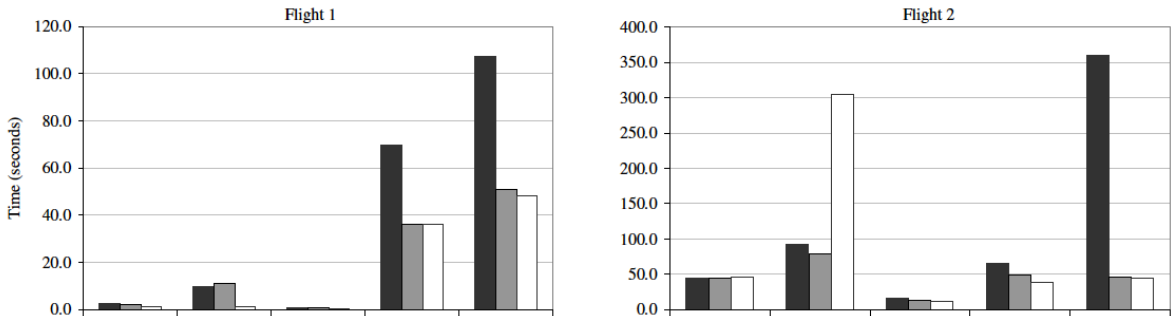


	T	T(B)	MV	VP	AI
Average	25.7	64.0	10.2	79.9	221.2

(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.

Column-Store Simulation in a Row-Store



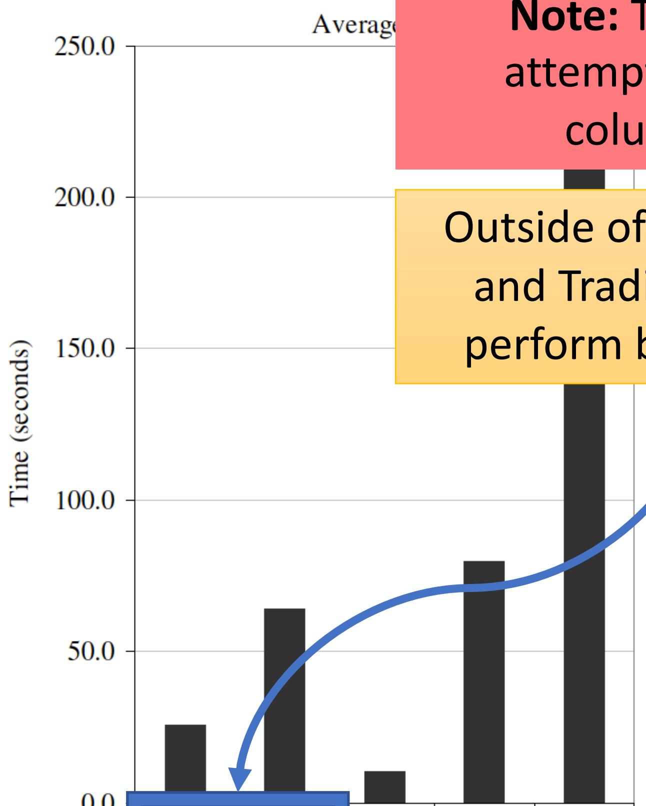
Flight 1	T	T(B)	MV	VP	AI
Q1.1	2.7	9.9	1.0	69.7	107.2
Q1.2	2.0	11.0	1.0	36.0	50.8
Q1.3	1.5	1.5	0.2	36.0	48.5

Flight 2	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

Flight 3	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

Flight 4	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)

Note: These are not attempts to emulate column-stores

Outside of `MV`s, Traditional and Traditional(bitmap) perform best on average

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.

Column-Store Simulation in a Row-Store

- Why can't we outperform traditional methods (`T` and `T(B)`)?
  - Tuple Overheads
    - Tuple overhead is quite large in fully vertical portioned approach
    - Must maintain `rids` or primary keys with each column → **tuple construction**
      - Adds significant overhead to read operations
- **Vertical partitioning** (`VP`) approach is competitive with row store when few columns are selected
  - However, as the number of columns selected grows
    - Tuple headers waste space and redundant `rids` yield inferior performance

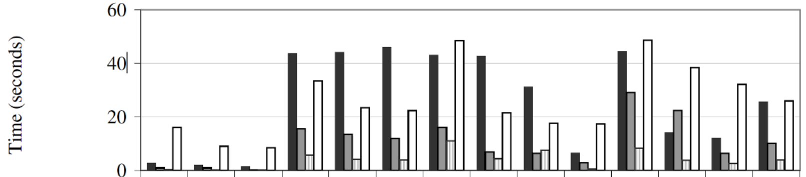
Column-Store Simulation in a Row-Store

- **Indexing Only** (`IA`) approach has low per-record overhead, but hash joins with fact table are expensive
  - System X is unable to defer joins until later in the query plan
    - Cannot retain `rids` from fact table after joining with a dimension table

Column-Store Performance

- Column stores → No tuple overhead + low join costs
  - Tuple headers are stored separately from data
  - Column stores rely on position order not keys or `rids`
- How does it beat `RS (MV)` as they have similar I/O and no joins are required from same table.
  - With all else being the same `CS`' advantage may result from its optimizations
    - Compression
    - Late materialization
    - Block Iteration
    - Invisible Join

Recall: `AVG CS` is faster than `RS (MV)` !

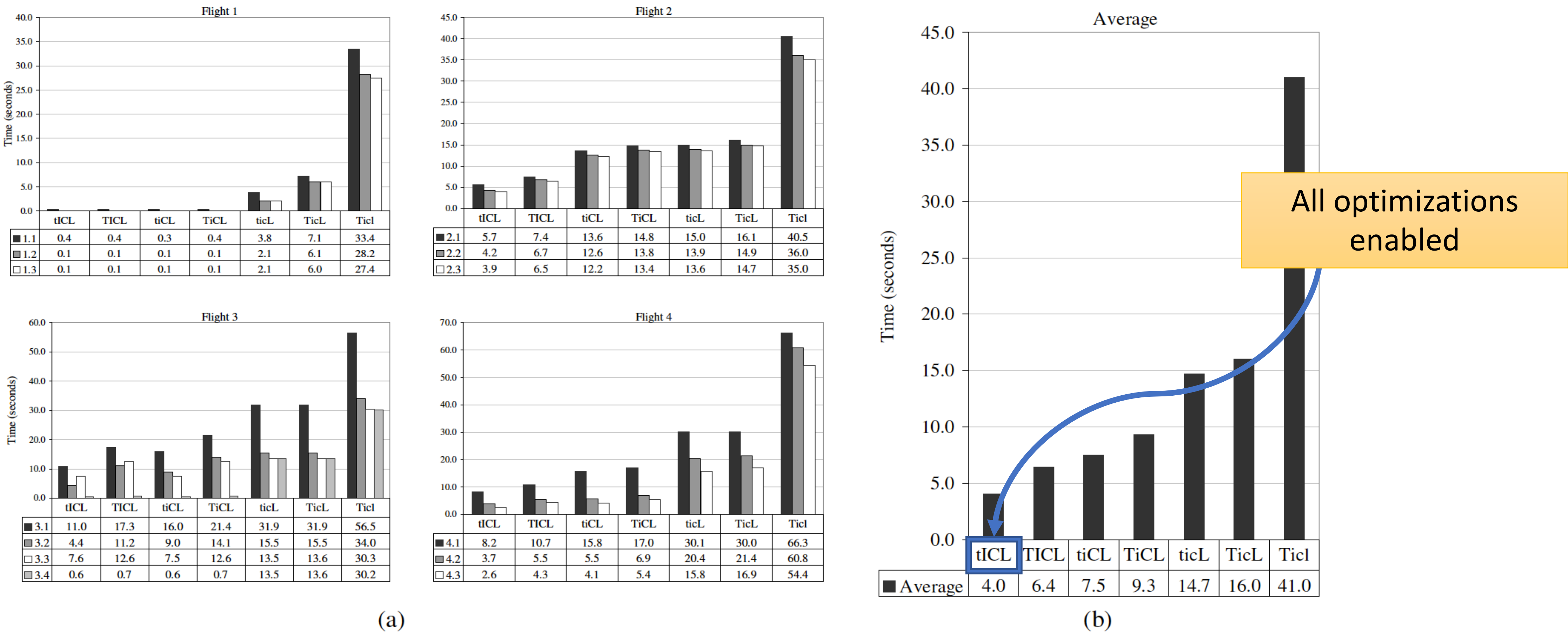


	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	AVG
■ RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
▩ RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
▥ CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
□ CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

x 2.55

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

Column Store Performance



Flight 1	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

Flight 2	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

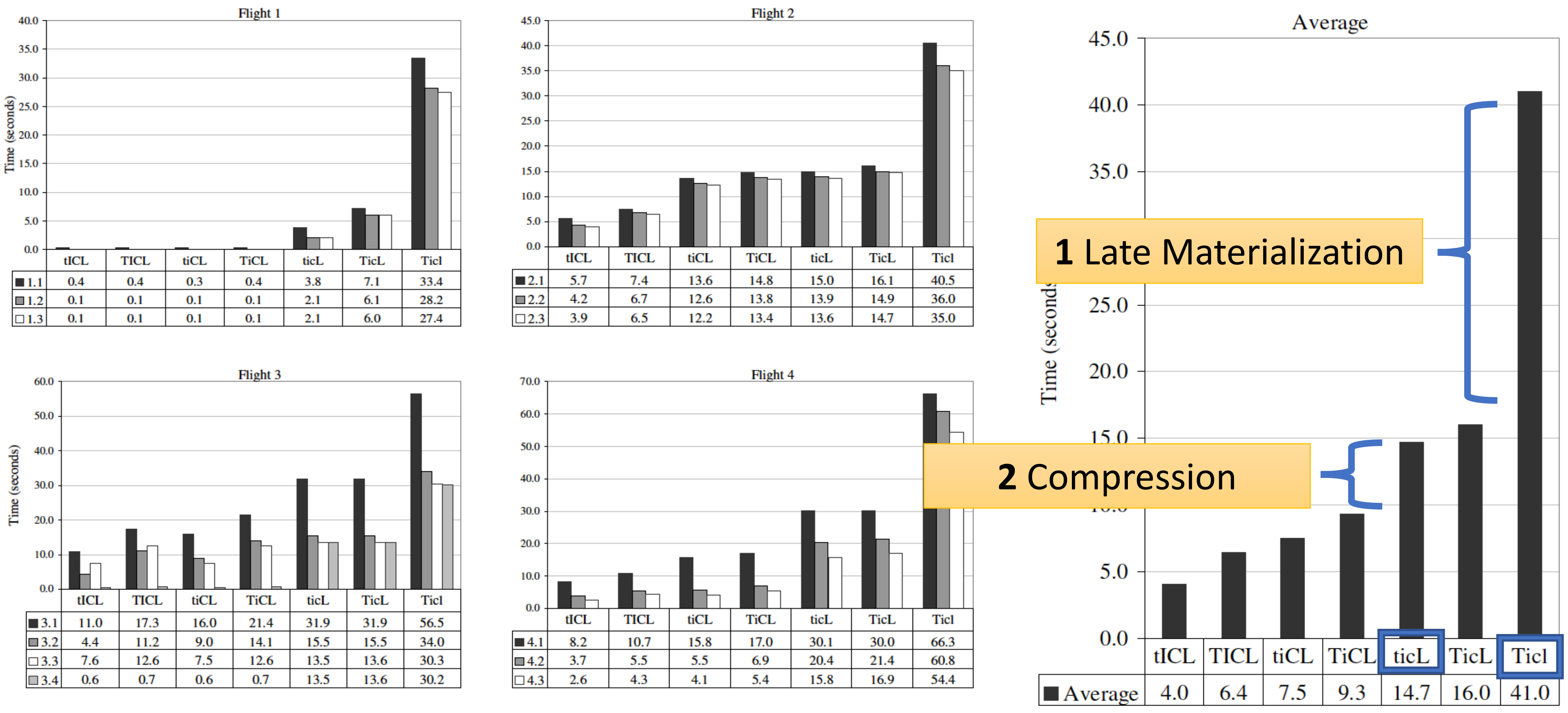
Flight 3	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
3.1	11.0	17.3	16.0	21.4	31.9	31.9	56.5
3.2	4.4	11.2	9.0	14.1	15.5	15.5	34.0
3.3	7.6	12.6	7.5	12.6	13.5	13.6	30.3
3.4	0.6	0.7	0.6	0.7	13.5	13.6	30.2

Flight 4	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

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

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.

Column Store Performance



Flight 1	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

Flight 2	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

Flight 3	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

Flight 4	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

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

(a) (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.

Conclusion

- Authors successfully illustrate attempts to reproduce I/O performance of column-stores in row-stores were rather fruitless
  - High tuple reconstruction costs
  - High per tuple overheads
    - Tuple headers
    - `rids` or primary keys
- Optimizations of column-stores were thoroughly explored
  - Identifying the key advantages over row-stores in **late materialization** and **compression** optimizations
- Proposed a new join technique **invisible join**
  - Extending late materialization via **between-predicate rewriting**

References

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (*SIGMOD '08*). Association for Computing Machinery, New York, NY, USA, 967–980. DOI:https://doi.org/10.1145/1376616.1376712