Column-stores vs. row-stores: how different are they really?

Daniel J. Abadi, Samuel R. Madden, Nabil Hachem
Sigmod/Pods 2008

Presented by: Eric Lim, Noah Pantoja, Harshit Gupta, Xuan-Huai Weng, Xing Liu

Background

- **Current Landscape**: Traditional databases are row-oriented, ideal for transactional workloads (e.g., banking, online retail).
- **Challenge**: Analytical tasks with large data volumes often suffer in row-stores due to inefficient data access.
- **Key Question**: Do column-stores outperform row-stores in read-heavy, analytical workloads?

Contribution

- **Main Idea**: Systematic comparison between column-stores and row-stores.
- **Primary Contribution**: Benchmarking of both storage types under standardized conditions.
- **Key Insights**: Detailed analysis of query speed, compression efficiency, and join performance for each model.

Prior Work

- **Row-Stores**: Optimizations focused on transaction processing efficiency (e.g., IBM DB2, Oracle).
- **Column-Stores**: Initial studies showed column-stores excel in read-heavy tasks and data compression (e.g., C-Store, Vertica).
- **Research Gap**: Lack of direct comparison between row-stores and column-stores under the same workloads.

Motivation for this Study

- **Why Column-Stores?**
  - **Big Data Growth**: Increase in data volume creates demand for efficient analytics.
  - **Need for Speed**: Analytical queries require faster data retrieval than row-stores can offer.
- **Focus**: Understand if column-stores can deliver better performance and scalability in data-intensive environments.

Related Work and How This Paper Differs

- **Benchmarking Approach**: First study to conduct a direct comparison of row-store and column-store architectures.
- **Evaluation Criteria**: Focuses on retrieval speed, compression ratios, and join operations.
- **Positioning**: Provides practical insights for choosing storage based on workload type, going beyond isolated optimizations.

STAR SCHEMA BENCHMARK

- Data warehousing benchmark that the paper uses for comparing the performance of C-store with the commercial row store.
- Chosen because-
  - Easier to implement than TPC-H
  - Did not require C-store modifications
- Has a scale factor that can be used for scaling the database up/down, since each table’s size is defined relative to this factor.
- The paper uses a scale factor of 10
- Consists of a fixed set of 13 queries divided into 4 categories called “Flights”.

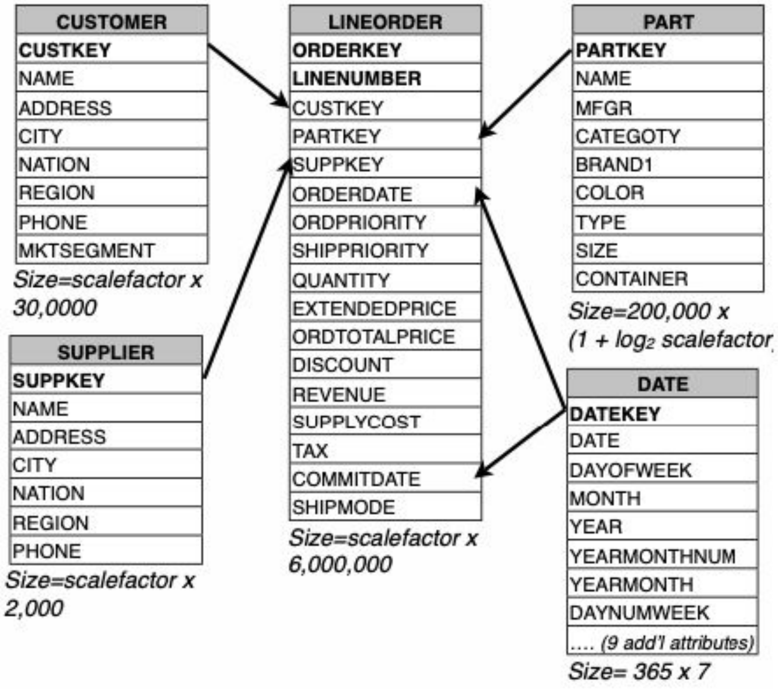


Figure 1: Schema of the SSBM Benchmark

Figure taken from: [Rex: Representative Extrapolating Relational Databases]()

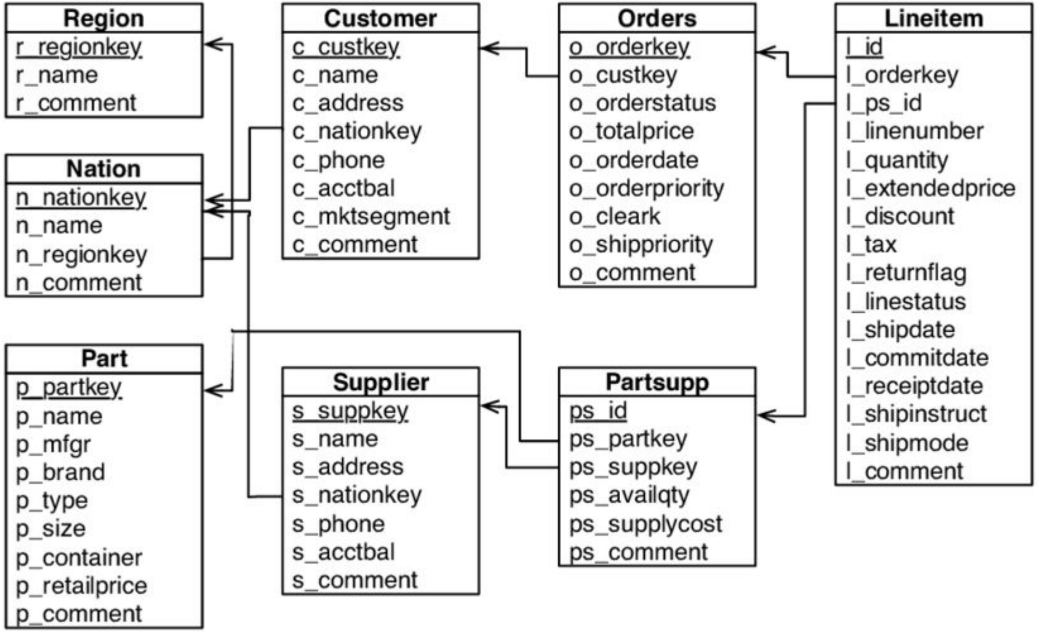


Figure 2: TPC-H Benchmark Schema

Figure taken from: [Rex: Representative Extrapolating Relational Databases]()

Row-Oriented Execution

- The objective is to implement column-database design in a commercial row-oriented DBMS, in an attempt to emulate the former’s performance.
- 3 physical design approaches-

1. **Vertical Partitioning:**
    - Creates one table corresponding to each column, with an additional “Position” column in each new table.
    - When fetching multiple columns, the rewritten queries perform join on the position attributes.
    - Authors experimented with Index Joins for joining without observing performance improvements over the default Hash joins, due to additional I/Os in Index Joins.

2. Index Only Plans:

- Unclustered B+ Tree index is added on each column of each table
- Based on the predicates, builds lists of (recordId, value) pairs. For multiple predicates on the same table, merges the lists in-memory.
- Further optimized by creating indices with composite keys, with secondary keys being from predicate-less columns - this prevents scanning of the entire predicate-less columns.

3. Materialized Views:

- For each flight, materialized views are created consisting of only the needed columns for that flight
- Allows the DB to access only relevant data from the disk, and avoid overheads such as position/ rid
- Requires advanced knowledge of the query workload.

Column-Oriented Execution

- **Objective**:

  Enhance performance for analytical queries by optimizing how data is accessed and processed.
- **Problem Being Solved**:

  Traditional row-store databases access all columns even if only a few are needed, leading to inefficiency.
- **Solution**:

  Column-oriented execution focuses on selective data retrieval and efficient join operations.

Compression

- Reduce disk I/O and enhance in-memory efficiency.
- Column stores use methods like Run-Length Encoding (RLE) and lightweight schemes.
- Similar values in columns (e.g., repeated entries in a sorted column) are highly compressible.
- Example: In a column with sorted dates, repeated values can be stored as [Date, Count] instead of individual entries.

Late Materialization

- Defer combining columns into full rows until absolutely necessary.
- Apply filtering and operations on individual columns first, delaying row construction.
- Benefits:
  - Avoids unnecessary data processing.
  - Enhances performance by only fetching relevant data.
- Example: For an “Employees” query filtering by “Department” and “Salary,” only those columns are accessed initially; other details (like “Name”) are fetched later if required.

Block Iteration

- Process data blocks rather than individual rows to reduce overhead.
- A block of values (e.g., arrays of column values) is passed between operations instead of row-by-row.
- Minimizes function call overhead and exploits CPU parallelism.
- Example: Instead of iterating row-by-row, a query processes a block of date values as a single unit, improving cache efficiency and reducing call overhead.

Invisible Join

- Efficiently handle joins, particularly in star schemas, by reimagining joins as column-based predicates.
- Steps:

    1. Apply predicates to dimension tables (e.g., filter by “region” or “year”).
    2. Use the results to filter positions in the fact table based on foreign keys.
    3. Retrieve values only for the selected positions, minimizing data processing.

Example

- For a query to find total revenue from “Asia” customers between 1992 and 1997:

```
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;
```

Example - Invisible Join Approach

- Phase 1: Apply filters to dimension tables and build hash tables of keys that match:

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

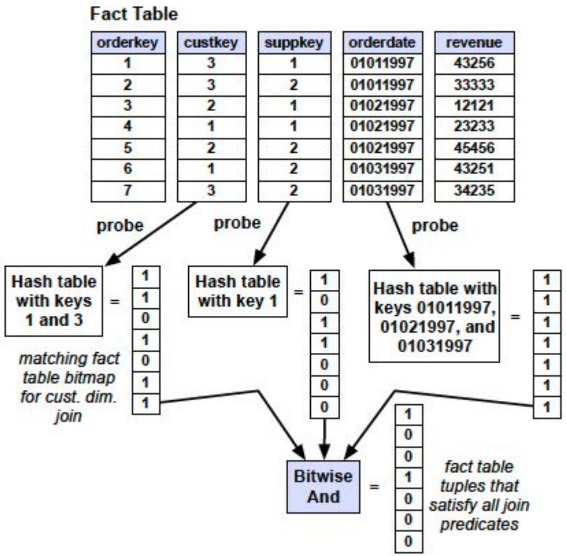
dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

Hash table with keys 01011997, 01021997, and 01031997

```
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;
```

Example - Invisible Join Approach

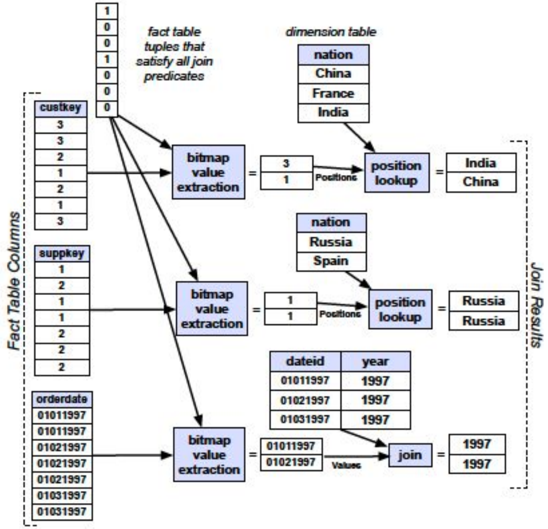
- Phase 2: Apply these hash tables as predicates directly on lineorder:




```sql
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;
```

Example - Invisible Join Approach

- Phase 3: Fetch and Aggregate Results:




```sql
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;
```

Benefits of Invisible Join

- Efficient Filtering: Avoids early construction of tuples, reducing unnecessary data processing.
- Optimized Joins: Transforms joins into predicates on the fact table, minimizing out-of-order extractions.
- Improved Query Performance: By minimizing data retrieval steps and handling only relevant rows, the invisible join improves performance on complex queries like this.

Experiments

- Four key questions
  - How do the different attempts to emulate a column store in a row-store compare to the baseline performance of C-Store?
  - Is it possible for an unmodified row-store to obtain the benefits of column-oriented design?
  - Of the specific optimizations proposed for column-stores (compression, late materialization, and block processing), which are the most significant?
  - How does the cost of performing star schema joins in column-stores using the invisible join technique compare with executing queries on a denormalized fact table where the join has been pre-executed?

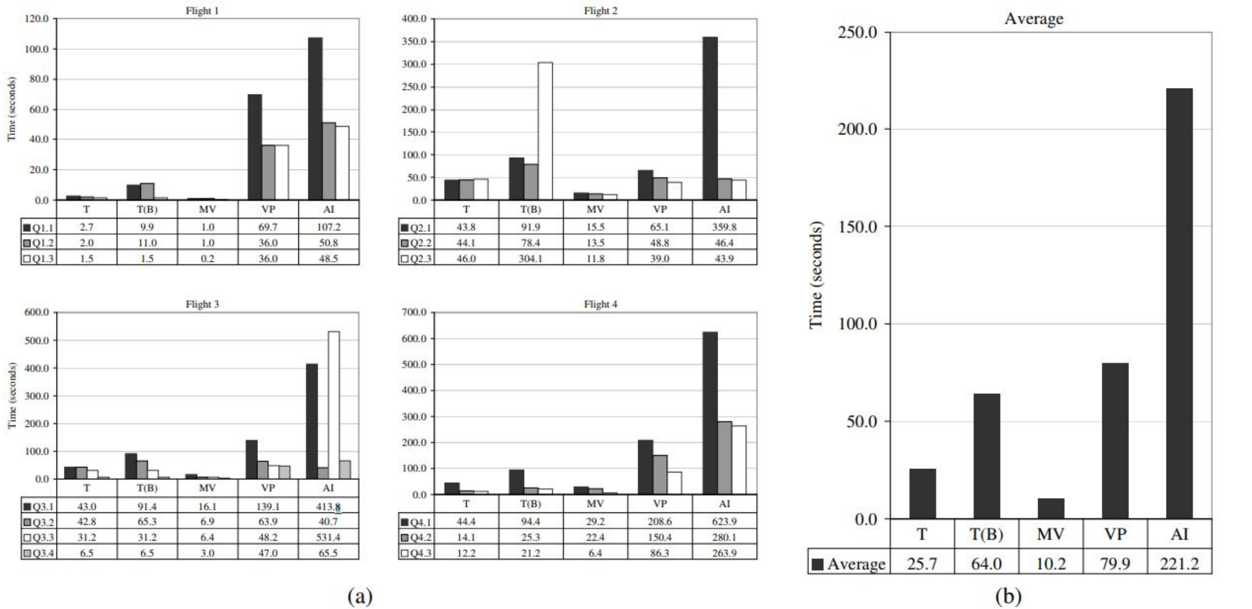
Experiment logistics

- Processor: 2.8 GHz single processor, dual-core Pentium(R) D workstation
- RAM: 3 GB
- Operating System: RedHat Enterprise Linux 5
- Storage Configuration: 4-disk array, managed as a single logical volume with file striping
- I/O Throughput:
  - Per disk: 40 - 50 MB/sec
  - Aggregate (striped files): 160 - 200 MB/sec
- Performance Measurements:
  - Based on averages of several runs
  - Conducted using a "warm" buffer pool

Experimental Setup

- RS: Row Row
- CS: Column Storage
- MV: Optimal collection of materialized views containing minimal projections of tables needed to answer each query
- CS (Row-MV) and the RS (MV) are executing the same queries on identical data stored in the same way

Row-store Performance



	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

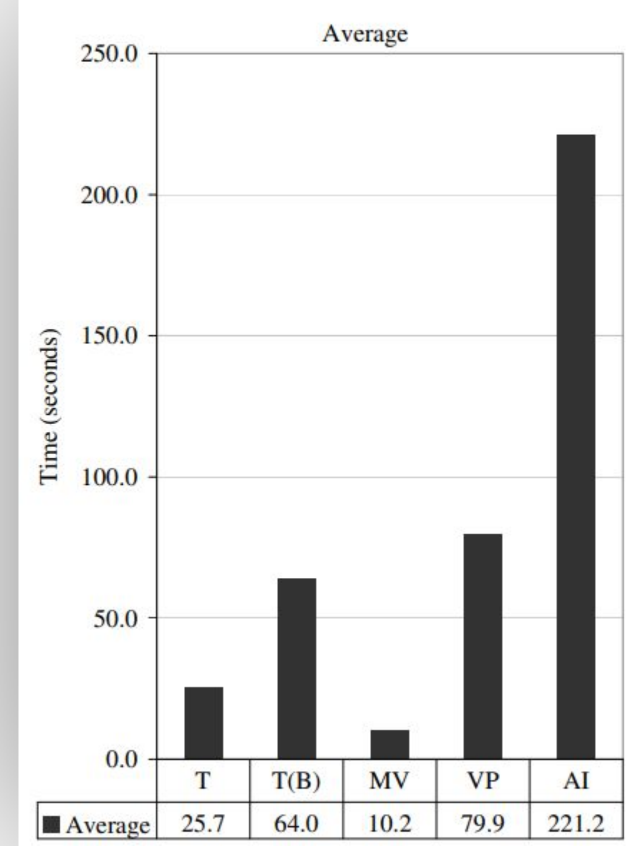
	T	T(B)	MV	VP	AI
Q4.1	44.4	94.4	29.2	208.6	623.9
Q4.2	14.1	25.3	22.4	150.4	280.1
Q4.3	12.2	21.2	6.4	86.3	263.9

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

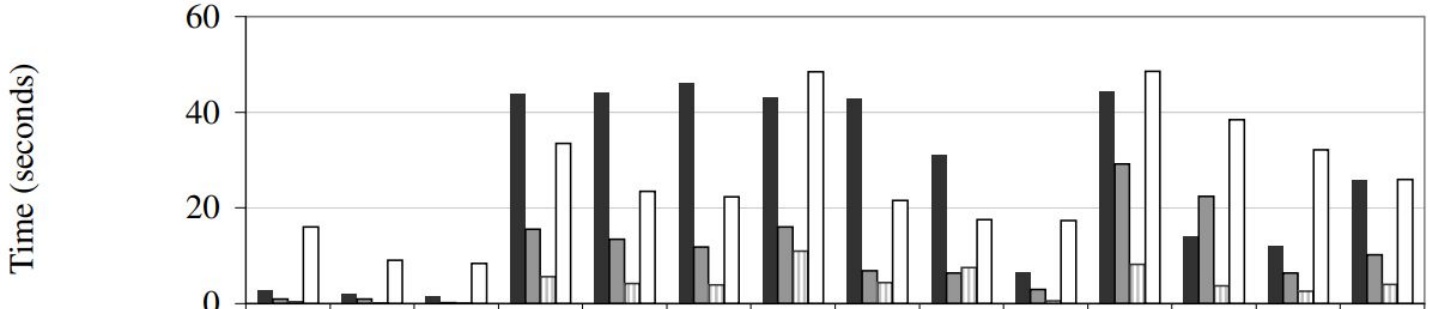
Figure 6: (a) Performance numbers for different variants of the row-store by query ight. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all queries.

Breakdown (2.1)

- Traditional
  - Performs well due to direct access and relatively low join costs
- Traditional Bitmap
- Materialized View
  - Significant reduction in data to read
- Vertical partitioning
  - limited by tuple overheads and costly joins
- Index-only plans
  - Suffers from performance issues due to frequent large hash joins and high data volume handling.



Column-Store Simulation in a Row-Store



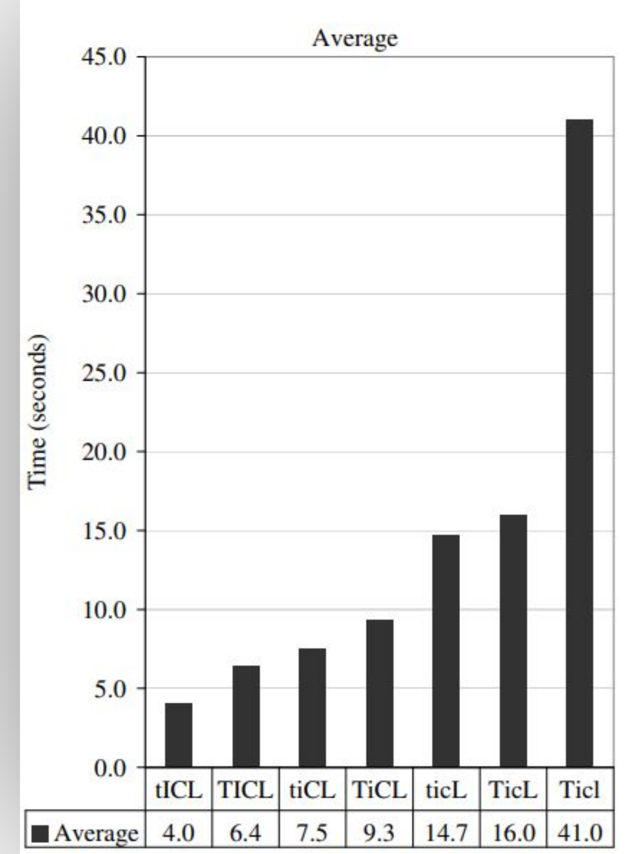
	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	AVG
■ RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
▒ RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
▥ CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
□ CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3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

Row-Store Simulations of Column-Stores

- High tuple overheads and expensive joins needed to reconstruct columns
  - Approx. 4Gb compressed
  - Approx. 2.3Gb for C-Store
- Partitioning and multi-threading not available for C-Store

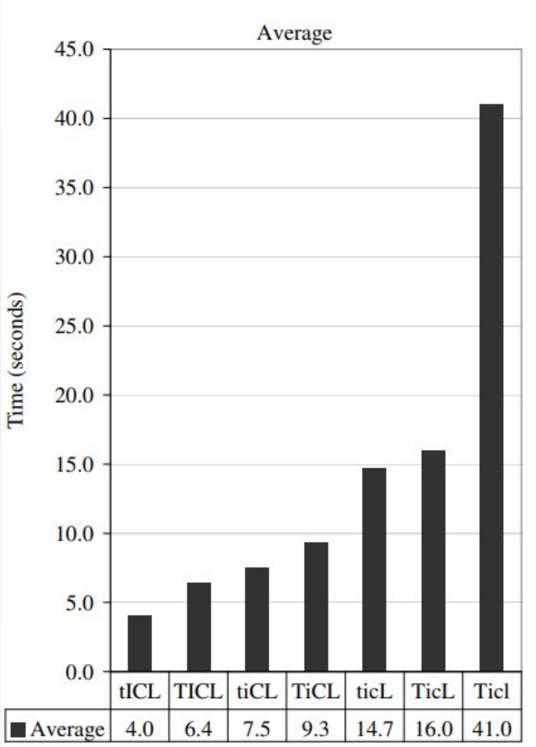
Column-Store Performance

- Block iteration
  - T=tuple-at-a-time processing,
  - t=block processing
- Invisible join
  - I=invisible join enabled,
  - i=disabled
- Compression
  - C= enabled
  - c=disabled
- Late materialization
  - L=enabled
  - l=disabled



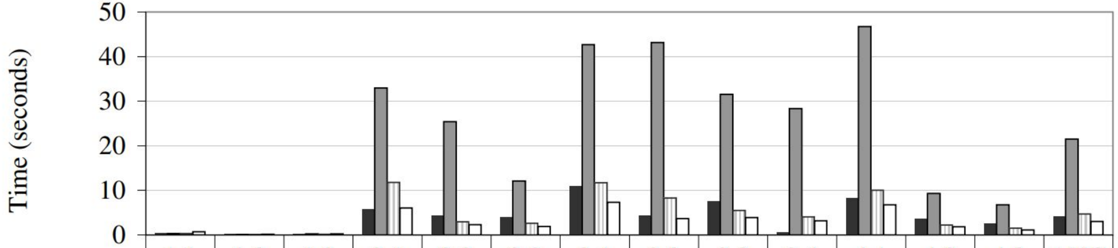
Column-Store Performance Breakdown

- Late materialization improves performance by a factor of approx. three
  - more selective the predicate, the more wasteful it is to construct tuples early on
- Compression improves performance by a factor of approx. two on average
- Invisible join improves performance by 50-75%
  - Performance difference is mostly due to the between-predicate rewriting optimization
- Block-processing can improve performance anywhere from a factor of 5% to 50%
  - If compression is removed Block Processing is less significant due to I/O bottlenecks



Join Performance

- Base: Invisible join
- PJ: Pre-executed join
    - C: No Compression
    - Int C: Compressed integer Dictionary
    - Max C: Maximum Compression



	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

Experiment Answers

- Four key questions
  - How do the different attempts to emulate a column store in a row-store compare to the baseline performance of C-Store?
    - See prior slides (Worse)
  - Is it possible for an unmodified row-store to obtain the benefits of column-oriented design?
    - No
  - Of the specific optimizations proposed for column-stores (compression, late materialization, and block processing), which are the most significant?
    - Compression and late materialization
  - How does the cost of invisible join technique compare with executing queries on a denormalized fact table?
    - Invisible joins perform just as well because they are cheap.

Conclusion - main contributions

- Row-store systems emulating column-store systems cannot achieve the same performance
  - Row-stores require a lot of overhead
  - Column-store benefits more from compression
  - Only column-store can utilize invisible joins
- Invisible join greatly improves the performance in column stores
  - On average gave a 50% - 75% performance increase
- Full analysis on column store optimizations
  - Late materialization
  - Compression
  - Invisible join
  - Block processing

Conclusion - Limitations and Future works

Limitations:

- **System comparison restraints -**
  - Row-stores can use multi-threading and row-store specific I/O optimizations making the performance comparisons skewed towards row-stores
- **Only one workflow and system tested -**
  - Only used the star schema benchmark so other workloads could have greatly differing results
  - Different systems with different memory/optimizations could have different numbers

Future work:

- Figuring out how to efficiently transform a row-store system into a column-store for analytical workflows

Study Questions

- Why do column-stores generally perform better than row-stores for analytical queries in data warehouses, and what specific optimizations in column-stores contribute to these performance gains?
- When simulating a column-store using a row-store, what are the primary limitations faced by row-oriented databases, and why do these methods (like vertical partitioning or index-only plans) fail to match the efficiency of true column-stores?