INFO-H-415 Advanced Databases
ColumnStore Indexes Vs Indexes in Relational Databases

Student: Yasmine Daoud
Ecole polytechnique ULB
Master Civil in Computer Science Engineering

December 2018

1 Introduction

A database index is an optional data structure that can be created for a column or list of columns to speed data access.

In this paper, we can understand what is an Index, what are types of indexes and what is the performance, comparing with two different methods of indexing (Row Index and Column Index) ? , independently of any tool

2 What is an Index ?

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

3 Quick Example

An application frequently looks up customers by their mobile phone number. To improve performance, developers configure an index for the field MOBILEPHONENUMBER on the CUSTOMER table. The database sets up a data structure that stores all MOBILEPHONENUMBER sorted numerically with pointers to the rows associated with each field. As the index is sorted, it can be searched quickly resulting in faster retrieval of CUSTOMER records.

4 Performance Tradeoffs

Indexes consume resources such as memory and disk space. They potentially reduce data access times and disk I/O. It is common to index primary keys. It is also common to index any fields or set of fields that are often queried. Indexes need to be maintained so indexing transactional tables that are updated more than queried can negatively impact performance.

5 Types of indexes in general

- **Clustered**:
  Clustered index sorts and stores the rows data of a table / view based on the order of clustered index key. Clustered index key is implemented in B-tree index structure.
- **Non-Clustered**:
  A non clustered index is created using clustered index. Each index row in the non clustered index has non clustered key value and a row locator. Locator positions to the data row in the clustered index that has key value.
- **Unique**:
  Unique index ensures the availability of only non-duplicate values and therefore, every row is unique.
- **Full-text**:
  It supports is efficient in searching words in string data. This type of indexes is used in certain database managers.
- **Spatial**:
  It facilitates the ability for performing operations in efficient manner on spatial objects. To perform this, the column should be of geometry type.

  Filtered:
  A non clustered index. Completely optimized for query data from a well defined subset of data. A filter is utilized to predicate a portion of rows in the table to be indexed.

6 What is a ColumnStore Index ?

A columnstore index is a type of data structure that's used to store, manage and retrieve data that is stored in a columnar-style database.

Columnstore indexes work well with read-only queries and large-scale analysis. One of their primary uses is queries for data warehousing. Columnstore indexes come with certain benefits over their row store counterparts; they can achieve higher compression rates and reduce I/O from physical media.

Columnstore indexes come in two varieties – clustered and non-clustered. Clustered columnstore indexes are updateable and are always the primary storage method for their entire table. They cannot be combined with other indexes and they do not physically store columns in a particular order. Non-clustered columnstore indexes can index subsets of columns. They require extra storage to store a copy of a column in the index and are updated by rebuilding or partitioning the index. Non-clustered columnstores can be combined with other tables and physically store columns in a particular order to optimize compression.

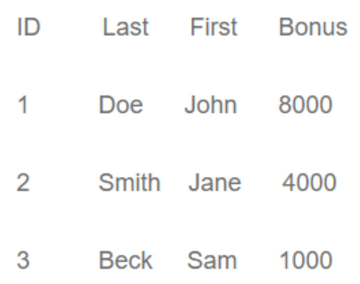
7 Columnar Database

A columnar database is a database management system (DBMS) that stores data in columns instead of rows.

The goal of a columnar database is to efficiently write and read data to and from hard disk storage in order to speed up the time it takes to return a query.

In a columnar database, all the column 1 values are physically together, followed by all the column 2 values, etc. The data is stored in record order, so the 100th entry for column 1 and the 100th entry for column 2 belong to the same input record. This allows individual data elements, such as customer name for instance, to be accessed in columns as a group, rather than individually row-by-row.

Here is an example of a simple database table with 4 columns and 3 rows.



ID	Last	First	Bonus
1	Doe	John	8000
2	Smith	Jane	4000
3	Beck	Sam	1000

In a row-oriented database management system, the data would be stored like this:
1,Doe,John,8000;2,Smith,Jane,4000;3,Beck,Sam,1000;

In a column-oriented database management system, the data would be stored like this:
1,2,3;Doe,Smith,Beck;John,Jane,Sam;8000,4000,1000;

One of the main benefits of a columnar database is that data can be highly compressed. The compression permits columnar operations — like MIN, MAX, SUM, COUNT and AVG— to be performed very rapidly. Another benefit is that because a column-based DBMSs is self-indexing, it uses less disk space than a relational database management system (RDBMS) containing the same data.

As the use of in-memory analytics increases, however, the relative benefits of row-oriented vs. column oriented databases may become less important. In-memory analytics is not concerned with efficiently reading and writing data to a hard disk. Instead, it allows data to be queried in random access memory (RAM).

8 Column Store VS Row Store - Example

EmpID	LastName	FirstName	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

Row Store	Column Store

1,Smith,Joe,40000;
2,Jones,Mary,50000;
3,Johnson,Cathy,44000;
	1,2,3;
	Smith,Jones,Johnson;
	Joe,Mary,Cathy;
	40000,50000,44000;

9 Why Column Stores ?

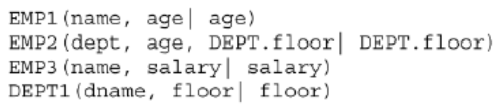
- Row oriented databases are optimized for writes
  Single disk write succes to push all fields of a record onto disk
- Systems for ad-hoc querying should be read optimized
- Common example from data warehousing
  Load new data in bulk periodically
  Long period of ad-hoc querying
- For such read intensive workloads, column store architecture is more suitable
  Fetch only required columns for query

Better cache effects
Better compression due to similar attributes within a column

10 Column Stores - Data Model

We present the C-Store data model as a representative data model for column stores.

- Standard relational logical data model
- EMP(<u>name</u>, salary, age, dept), DEPT(<u>dname</u>, floor)
- Table - Collection of projections
- Projection - set of columns, sorted

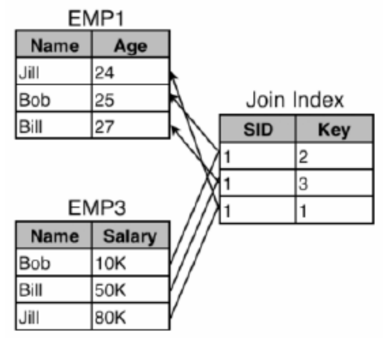


```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```

- Horizontally partitioned into segments with segment identifier(sId)
- Column-to-row correspondence identified by position in the segemnt (storage key or sKey)
- How to put together various colums into a tuple ?
  Well, we can use storage keys, what is data sorted on a different attribute ? Sorting again and merging is inefficient
- Solution: Use **Join Indexes**
- Let T1 and T2 be two projections on table T
- M segments in T1, N segments in T2
- Join index from T1 to T2 contains M tables
- Each row is of the form (s: SID in T2, k: Storage Key in Segment s)
- In other words, for each sKey in a particular partition of T1, it specifies where(which partition, which sKey) the corresponding row is located in T2

Join indexes Example - Join Index from EMP3 to EMP1

```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```



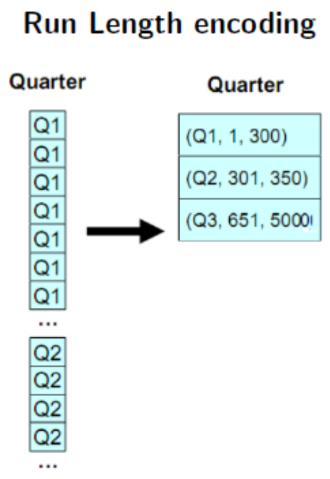
11 Compression

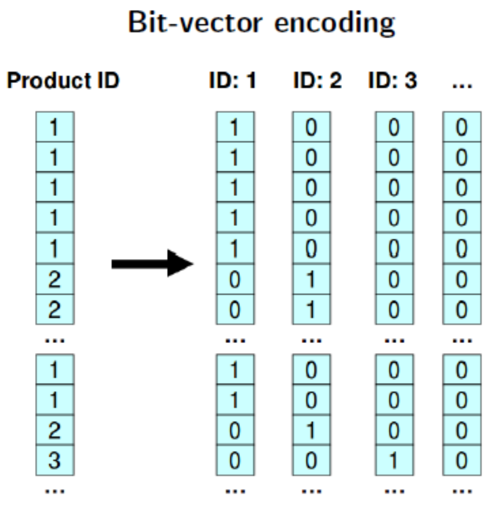
- Column-organized data can be compressed well due to good locality
- Trades IO for CPU
- Compression Schemes:Run Length encoding, Dictionary encoding, Bit-vector encoding, Null suppression, Heavy-weight schemes
- Choice of compression depends on: Characteristics of data, Decompression trade-off(more compressed necessarily not the better)

12 Compression Examples

13 Query Execution Operators

- Decompress: Converts compressed columns into uncompressed representation
- Select : Same as relational select but produces bit-string
- Mask : Takes a bit-string B and a projection Cs, emits only those values from Cs whose corresponding bit in B is 1
- Project : Equivalent to relational project
- Sort a projection on some columns
- SQL-like aggregates
- Combines two projections which are sorted in same order





- Re-order the projection according to a join index
- Join two projections on a predicate
- BAnd (bitwise-AND), BOr (bitwise-OR), BNot (bitwise-NOT)

14 Row vs Column - Summary of Differences

Row Store	Column Store
All fields in a record are stored contiguously	Each field is stored separately
Byte/word alignment of records	Dense packing
Compression/encoding discouraged	Make good use of compression
Needs to read all attributes to process any query	Only the necessary attributes can be read
Very good for OLTP queries	Not good for OLTP but performs much better for analytics workloads

15 Row Store VS Column Store

Now the simplistic view about the difference in storage layout leads to that one can obtain the performance benets of a column-store using a row-store by making some changes to the physical structure of the row store.
This changes can be

- Vertically partitioning
- Using index-only plans
- Using materialized views

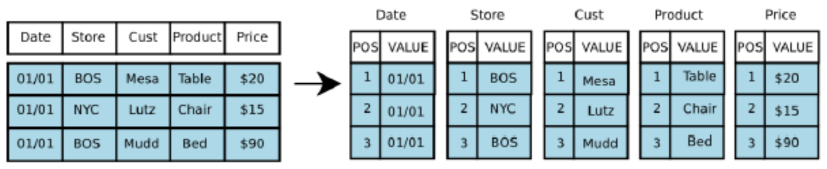
16 Vertical Partitioning

Process: Full Vertical partitioning of each relation, each column =1 Physical table. This can be achieved by adding integer position column to every table, Adding integer position is better than adding primary key and Join on Position for multi column fetch

- Partition each relation on all its columns
- How to match columns corresponding to the same row?
- Store primary key - can be expensive
- Use a 'position' attribute
- Rewrite queries to join on position attribute to get multiple columns from same table

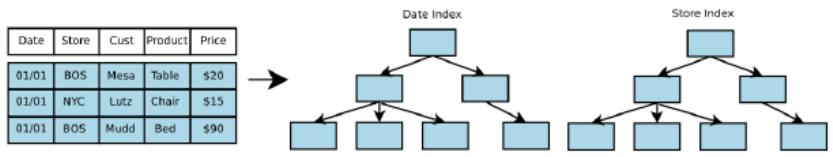
Problems:

- "Position" - Space and disk bandwidth
- Header for every tuple – further space wastage
- e.g. 24 byte overhead in PostgreSQL



17 Index-Only Plans

- Disadvantages of Vertical Partitioning
- Need to store position with every column
- large header for each tuple
- Leads to space wastage
- Alternative- Use indexes



Date	Store	Cust	Product	Price
01/01	BOS	Mesa	Table	$20
01/01	NYC	Lutz	Chair	$15
01/01	BOS	Mudd	Bed	$90

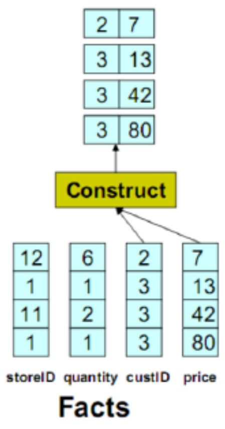
- Base relation stores using standard relational design
- B+ tree index on every column of table
- Tuple header not stored - so overhead is less
- No need to access actual tuples on disk
- Based on the query predicate, index is consulted and (recordid; column-value) pairs are returned
- These are then merged in memory
- **Disadvantages**
- If there is no predicate on the column, requires full scan of the index to read all tuples
- Example: SELECT AVG(salary) FROM EMP WHERE age ¿ 40
- With separate indexes, First Find record id's satisfying predicate on age
- Then join this with full column set for salary
- Can answer directly if there is an index on (age; salary)

18 Materialized Views

Process:

- Create 'optimal' set of MVs for given query workload
  Objective:
- Provide just the required data
- Avoid overheads
- Performs better
- Expected to perform better than other two approach

  Problems:
- Practical only in limited situation
- Require knowledge of query workloads in advance



19 Experimental Setup

System X: a traditional row store. **C-Store**: a column store
Goals:

- Compare performance of C-Store vs column store emulation on System X
- Identify which optimizations in C-store are most significant
- Infer from results if it is possible to successfully emulate a column store using a row store

- What guidelines should one follow?
- Which performance optimizations will be most fruitful?

Machine

- 2.8 GHz single processor, dual core Pentium(R) D workstation
- 3 GB RAM
- Red Hat Enterprise Linux 5
- 4-disk array, managed as a single logical volume
- Reported numbers are average of several runs
- Also, a ”warm” buffer pool - 30 per cent improvement for both systems

Data

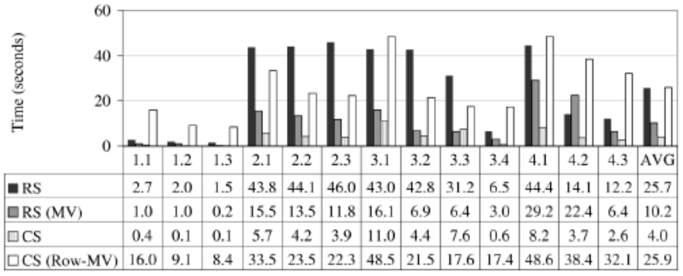
- Star Schema Benchmark (SSBM) - derived from TPCH
- Fact table: 17 columns, 60,000,000 rows
- Table : LineOrder
- 4 dimension tables: largest - 80,000 rows
- Tables: Customer, Supplier, Part, Date

Workload

- 13 queries divided into four categories or ights”
- Data warehousing queries
- Flight 1 : Restriction on 1 dimension attribute + columns on the fact table
- Flight 2: Restriction on 2 dimension attributes
- Flight 3, 4: Restriction on 3 dimensions

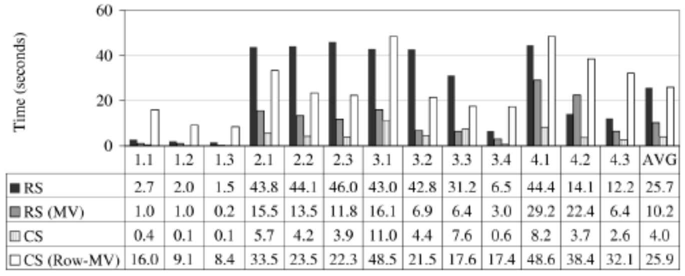
20 Baseline and Materialized View

- C-store outperforms System X
- Factor of 6 in base case (tCS vs RS)
- Factor of 3 with MV on system X (CS vs RS(MV)) Expected on warehouse workloads



	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

RS: Row store, **RS(MV)**: Row Store with optimal set of materialized views, **CS**: column store, **CS(Row-MV)**:Column store constructed from 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

RS: Row store, **RS(MV)**: Row Store with optimal set of materialized views, **CS**: column store, **CS(Row-MV)**:Column store constructed from RS(MV)

- CS(Row-MV) vs RS(MV), Expected to be comparable
- System X outperforms by a factor of 2
- System X more

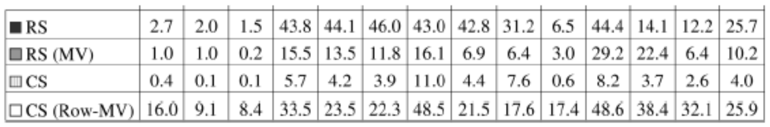
  ne tuned with advanced performance features
- Not a level ground for comparison

21 Need for a fair comparison

- Hidden factors might affect results when comparing two different systems
- Solution: Take one system at a time, and modify it Simulate column store inside System X

  Remove performance optimizations from C-Store until row store performance is achieved

.

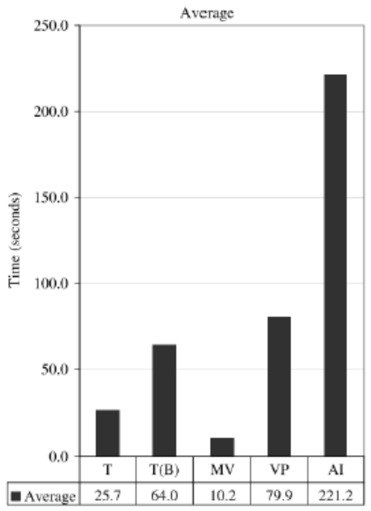


■ RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
▨ RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
▤ CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
□ CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

RS: Row store, **RS(MV)**: Row Store with optimal set of materialized views, **CS**: column store, **CS(Row-MV)**:Column store constructed from RS(MV)

- Inferences will be more reliable
- Example CS vs CS(Row-MV) - factor of 6 difference
  Although both read minimal set of columns
  Thus, less IO not the only factor

22 Column Store Simulation in System X used



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

Configurations of system X used

- Traditional (T)
- Traditional (bitmap): biased to use bitmaps; might be inferior sometimes (T(B))
- Vertical Partitioning: Each column is a relation (VP)
- Index-Only: B+Tree on each column (AI)

- Materialized Views: Optimal set of views for every query (MV)

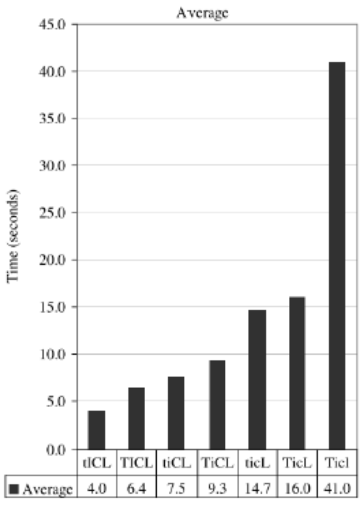
23 ColumnStore Simulation in System X - Analysis

- Materialized views performed the best -only plans the worst
  Expensive hash joins on fact table before it is

  ltered (to join columns)
  System X cannot retain fact table record id's after joining with another table

- Vertical Partitioning Comparable to MV when only a few columns were used
  Tuple overheads affected performance significantly when more than 1/4th of the columns used
  Scanning four columns in vertical partitioning approach took as long as scanning the entire table in traditional approach
  960 MB per column (vertical partitioning) vs 240 MB per column (C-store)

24 Column Store Performance



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

Approach

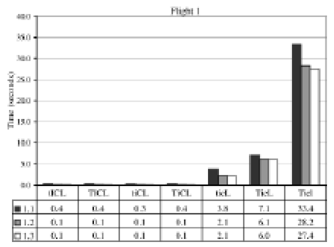
- Start with column store

- Remove each optimization

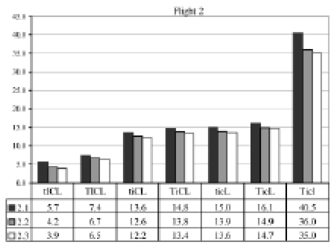
  Configuration

- 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

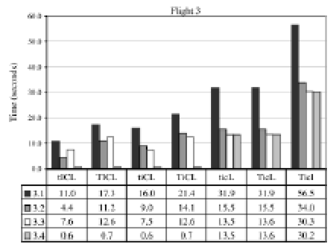
25 Column Store Performance - By Flight



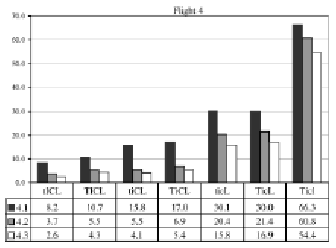
	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
■ 1.1	0.4	0.4	0.3	0.4	3.8	7.1	33.4
▨ 1.2	0.1	0.1	0.1	0.1	2.1	6.1	28.2
□ 1.3	0.1	0.1	0.1	0.1	2.1	6.0	27.4



	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

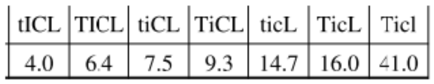


	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
■ 4.1	8.2	10.7	15.8	17.0	30.1	30.0	66.3
▨ 4.2	3.7	5.5	5.5	6.9	20.4	21.4	60.8
□ 4.3	2.6	4.3	4.1	5.4	15.8	16.9	54.4

26 Column Store Performance Analysis

Analysis

- Late materialization - factor of 3 (most important)
- Block processing - 5 per cent to 50 per cent depending on whether compression has been removed
- Invisible joins - 50 per cent to 75 per cent (largely due to between-predicate rewriting)



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

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

- Compression - factor of 2
- Tuple construction is costly - adds a factor of almost 2

27 Conclusion

- Column stores perform better than row stores for warehouse workloads
  Various optimizations in column stores contribute to improved performance
- Columns stores and row stores employ different design decisions
  No fundamental hindrances for row stores to adopt some of the techniques from column stores
  Example: Store tuple headers separately, use virtual record id's to join data etc.
- Emulating a column store inside row stores performs poorly
  Tuple reconstruction costs
  Per tuple overheads
- Some important system improvements necessary for row stores to successfully emulate column stores
- Can we build a complete row store that can transform into column store for warehouse workloads?
  SAP HANA has a solution for this