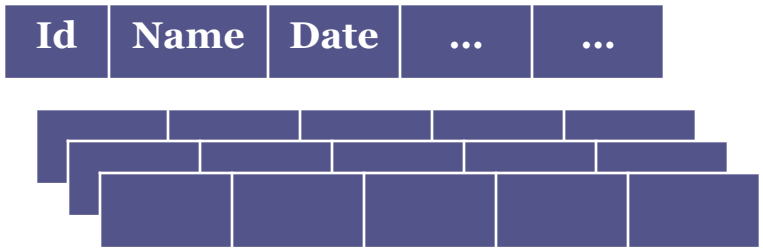
Column-Oriented Database Systems

Liliya Rudko
University of Helsinki

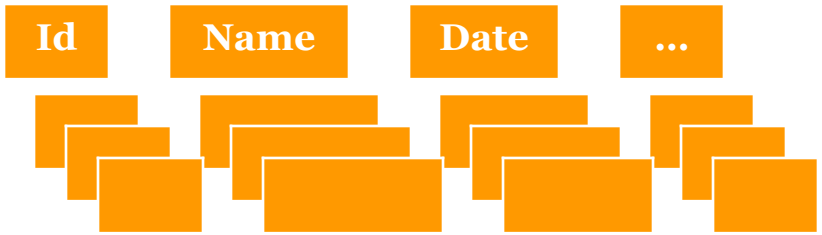
Contents

1. Introduction (1/4)

Row-oriented storage



Column-oriented storage



Used for

- transactional queries
- from time to time perform analytics (e.g., monthly reports)
- this analytics consume a lot of time

- analytical queries
- in research and analytics (e.g., medicine and astronomy)

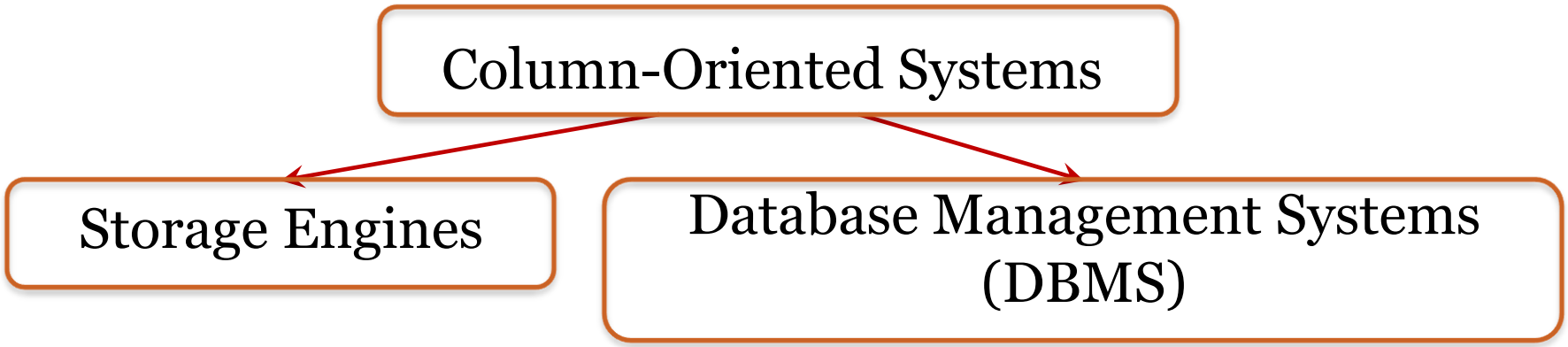
1. Introduction (2/4)

According to Abadi et. al. [1]:

• pure *storage simulation* (e.g., indexing each column and vertical partitioning) is not enough

• *query execution* process should also be reconsidered

1. Introduction (3/4)



- focus on storage layout
- query execution is out of scope
- difficult to test (own set up trial data flow)

- include storage engines
- focus on storage layout + query execution
- important bit – query execution techniques

1. Introduction (4/4)

Most beneficial query execution techniques [1]:

1. Vectorized query processing
    - 1.1. Block iteration (avoids storing large intermediate results into the main memory)
    - 1.2. Late materialization
2. Column data compression

2. Storage engines. ECOS (1/8)

ECOS:

Pure column-oriented storage manager

Main goal:

Customize storage structure (according to changeable data size and access patterns) with minimal human intervention

Currently:

ECOS' prototype has been completed

Storage layout:

- table-level customization
- column-level customization

2. ECOS. Storage layout (2/8)

Table-level customization

- identification of the table storage model
- there are five *Decomposed Storage Models (DSM)*

Conventional 2-copy DSM

- each column value – as <*key*, *value*>
- each relation – two copies

Column k0		Column k1	
Key	Value	Key	Value
k1	731	k1	20090327
k2	137	k2	20071201
k3	173	k3	20010925
k4	371	k4	20090327

(a) Columns clustered on key

Column v0		Column v1	
Key	Value	Key	Value
k2	137	k3	20010925
k3	173	k2	20071201
k4	371	k1	20090327
k1	731	k4	20090327

(b) Columns clustered on value

Fig. 1. Conventional 2-copy DSM [2]

2. ECOS. Storage layout (3/8)

Variations of the conventional 2-copy DSM

1. Key-copy DSM
2. Minimal DSM
3. Dictionary based minimal DSM
4. Vectorized dictionary based minimal DSM

Meant for specific cases (e.g., Key-copy DSM – for those tables that are queried for their key attributes only)

However

1. Algorithm for DSM selecting is not specified

2. Rahman et. al. [2] claim that Conventional 2-copy DSM is the most suitable (easy to use and implement, does not require human intervention, storage requirements are at most 50 Mbyte greater)

2. ECOS. Storage layout (4/8)

Column-level customization. Reasons:

- different workload
- different access patterns
- different number of distinct data

Customize column	=	Fit it in the appropriate place in the defined hierarchy of column structures

Benefits of having hierarchy for column structures

- can be mapped to the hardware hierarchy => storage optimization
- easy access
- can gather statistics => improve

2. ECOS. Storage layout (5/8)

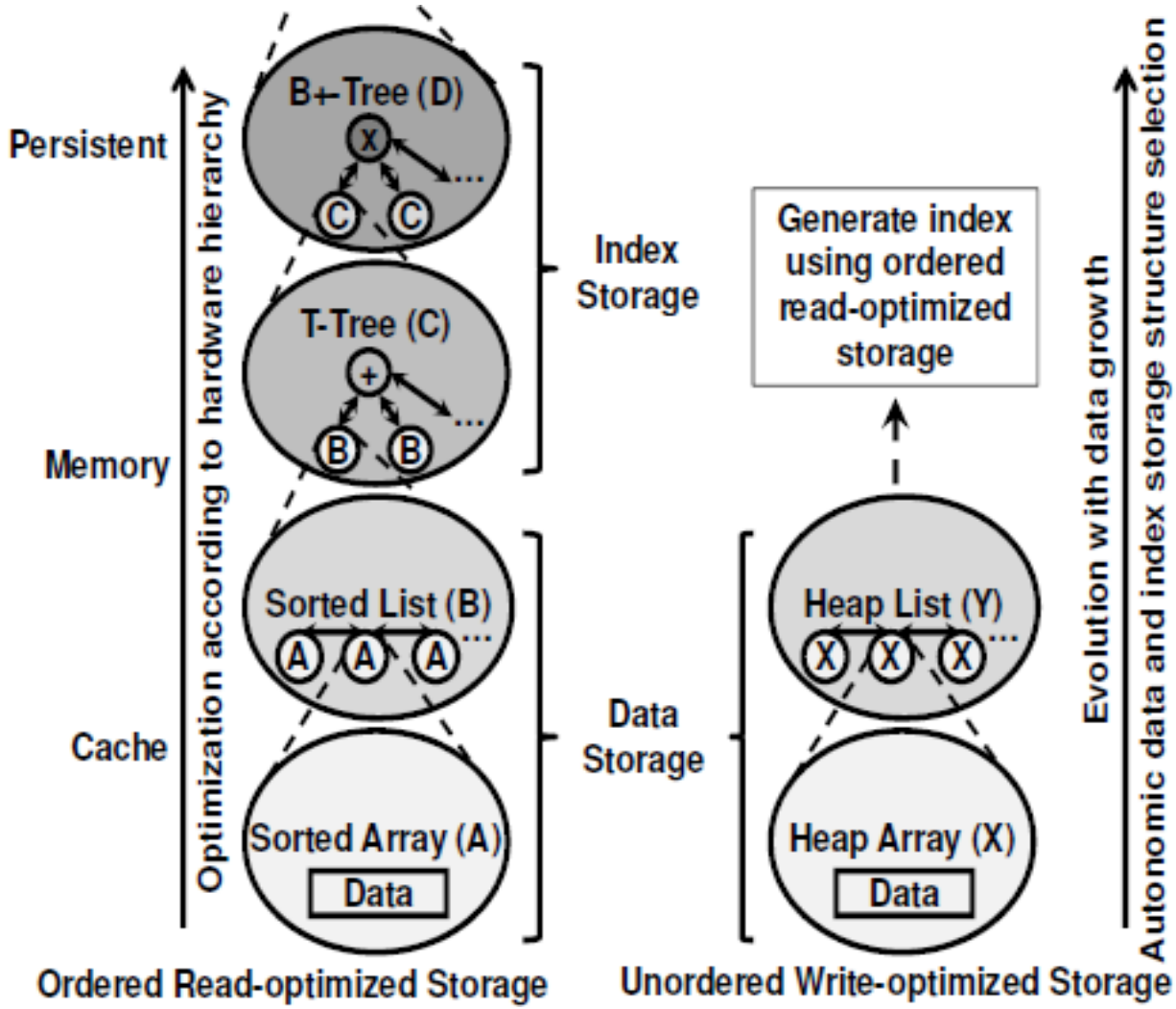
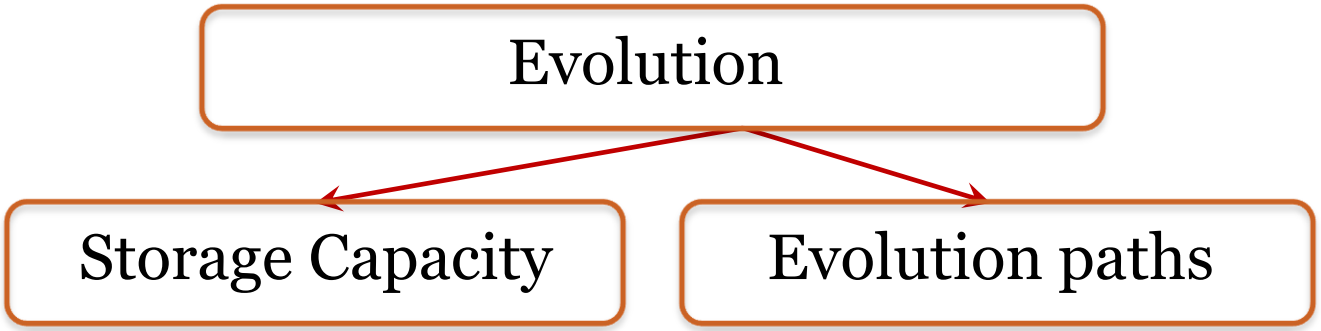


Fig. 2. Evolving storage structures[2]

2. ECOS. Storage layout (6/8)



Eliminates performance degradation due to unlimited data growth

Mutation rules

Event:
Sorted array=Full
Heredity based selection:
Workload=Read intensive
Data access=Unordered
Mutation:
=> Evolve (Sorted array
– >Sorted list)

Fig. 3. Mutation rule example [2]

2. ECOS. System level (7/8)

API allows direct access to any of the column storage structures (for testing)

2. ECOS. Discussion [2] (8/8)

1. Only one DSM is used

2. Human intervention for identifying column as ordered read-optimized of unordered write-optimized

3. Evolution is mainly based on data sizes (little attention is drawn to access patterns)

4. Evaluation results are doubtful (blurred queries and access patterns, just some extractions of the system are evaluated, “performance improvement” occurs to be negligible)

2. Storage engines. HYRISE (1/8)

HYPISE:

Hybrid row/column-oriented storage engine

Main goal:

Maximize cache performance for both OLTP and OLAP-style queries

Currently:

ECOS' prototype has been completed

Storage layout:

- based on cache and main memory only
- cost model that predict cache performance has been developed

2. HYRISE. Storage layout (2/8)

- tables are vertically partitioned
- adjusting number of columns in the partitions (according to access patterns)
- for analytical queries – narrow partitioning, for transactional queries – wider partitioning

Partitioning algorithms

- *"Layout selection" , "Divide and conquer partitioning"* [3]
- find the best possible (in terms of cache performance) physical design for a table with up to hundreds attributes
- known query workload
- the set of queries and their weights are used in the cache performance evaluation [3]

2. HYRISE. Storage layout (3/8)

"Layout selection" partitioning algorithm

Step 1. Candidate generation – identifies primary partitions that are always accessed together.

Example

- relation: N tuples, a_1 - a_4 attributes
- workload: projection $\pi_1 = \{a_1, a_2, a_4\}$, weight ω_1
  projection $\pi_2 = \{a_2, a_3, a_4\}$, weight ω_2
  selection σ of all the attributes, ω_3
- $\pi_1 => P_1 = \{a_3\}, P_2 = \{a_1, a_2, a_4\}$
  $\pi_2 => P_1 = \{a_3\}, P_2 = \{a_2, a_4\}, P_3 = \{a_1\}$
  σ does not change anything

2. HYRISE. Storage layout (4/8)

Step 2. Candidate merging – analyze performance gain by actually merging back some partitions.

Example

as we have two projections that access three attributes and one selection that accesses all four attributes, it may be more beneficial to have two partitions: $P_1 = \{a_3\}$, $P_2 = \{a_1, a_2, a_4\}$

- merge is advantageous for wide scans, disadvantages for narrow scans (+ extra overhead, depending on the width of the attributes, cache line size and the frequency of the operations)
- algorithm analyzes workload cost for every merge – if it is less than the sum of individual partitions – adds new partitions to the current set

2. HYRISE. Storage layout (5/8)

Step 3. Layout generation – analyze all the possible combinations from the Step 2, calculate their workload cost and choose the one with the lowest.

- algorithm running time is exponential to the number of partitions
- for wide tables there is a risk of poor performance
- *"Divide and conquer"* partitioning algorithm can scale to large sizes of relations with complex, non-regular workloads

2. HYRISE. Storage layout (6/8)

"Divide and conquer" partitioning algorithm

Step 1. Candidate generation – with maximum *K* partitions in one cluster (*K* is a constant)

Cost-optimized clusters (clustering problem in the research community)

Step 2. Candidate merging – applied to every cluster

New partitions for every cluster

Step 3. Layout generation – combines pairs of partitions from different clusters, whose combination is the most cost saving

2. HYRISE. Execution engine (7/8)

1. Implements projection, selection, join, sort and group by, supports late and early materialization

2. Single-threaded, however thread-safe data structures are used for later query parallelization

2. HYRISE. Performance and discussion [3] (8/8)

1. Compared to pure row-oriented systems – uses 4 times less CPU cycles

2. Compared to pure column-oriented systems – about 1.6 times faster for OLTP queries and virtually the same for OLAP queries

3. Good ground for further development

4. Suggested algorithms and physical designs can be used in other systems (cache performance gain)

3. DBMS. MonetDB (1/6)

MonetDB:

Pioneer among pure column-oriented DBMS

Main goal:

Performance improvement for analytics over large data collections

Currently:

Open-source, 10.000 downloads monthly

3. MonetDB. Storage layout (2/6)

- 1-copy DSM (*<key, value>* for each table – *Binary Association Table, BAT)*
- clustered on *key*
- *value* addresses to the BLOB location with the actual value

3. MonetDB. Execution engine (3/6)

Front end

- *strategic optimization* of user-space query language
- models of the user-space data are mapped to *BATs*
- user-space query language is translated to *MonetDB Assembly Language (MAL)*

Back end

- *tactical optimization* of the each given *MAL* program

Kernel

- provides final *BAT* structures
- *operational optimization* at run-time

3. MonetDB. Execution engine (4/6)

- implements data compression, vectorized execution
- constantly researching new ways of utilizing these and other techniques for performance improvement

3. MonetDB. System level (5/6)

- supports SQL:2003 standard, provides ODBC and JDBC client interfaces and application programming interfaces (e.g., C, Java, Ruby and Python)

- mainly focuses on read queries and updates of a large data chunks at a time

3. MonetDB. Some research areas (6/6)

1. Hardware-conscious database technology => new breed of query processing algorithms

2. Algorithms for reusing intermediate results in query processing

3. Adaptive indexing and database cracking

4. Stream processing in a column-store

3. DBMS. SQL Server 2012 (1/5)

SQL Server:

General-purpose DBMS that successfully implements row-wise indexes

SQL Server 2012:

Implements a new index type - *column-store index* and a new processing mode that handles batches of rows at a time

Currently:

Was successfully tested on real customers' workloads

3. SQL Server 2012. Storage layout (1/5)

- column-store index can function similarly to the row-store
  - any index can be stored as a column-store index (e.g., primary, secondary and filtered)
  - support all the known index operations (e.g., scans and updates)
- with a certain workload performance of column-store index is much higher

3. SQL Server 2012. Storage layout (2/5)

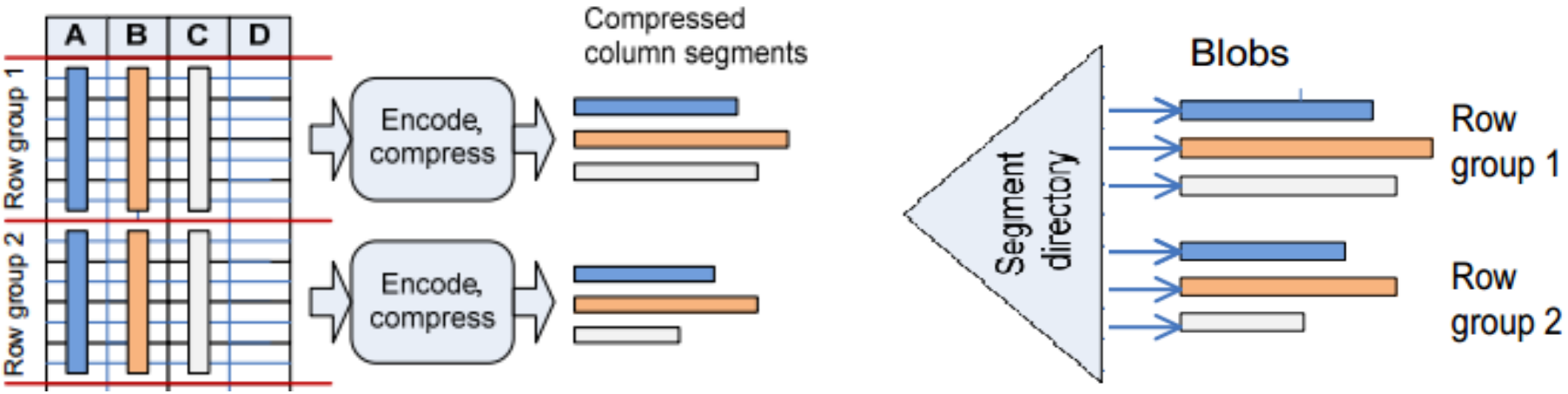


Fig. 4. Creation of a column-store index[4]

3. SQL Server 2012. Execution engine (3/5)

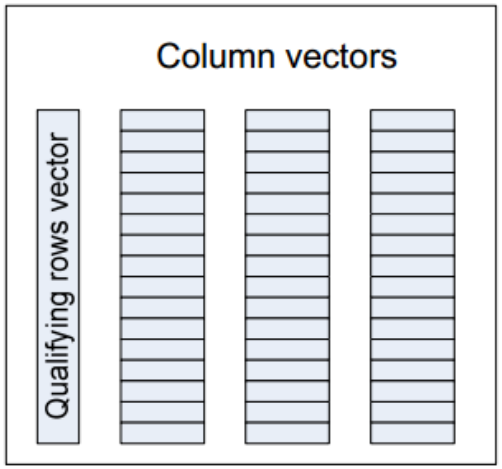


Fig. 5. A row-batch object [4]

Query optimizer identifies whether to use

- batch-mode processing
- row-mode processing

3. SQL Server 2012. System level (4/5)

- column-store indexes support up to 15.000 partitions per table. User can load parts of a table, index it with a column-store index and switch as a newest partition

- column-store is built on a fact table, table can not be updated or loaded with new data after indexing

3. SQL Server 2012. Discussion (5/5)

- batch-mode processing supports only some operations (e.g., scan and filter)
- there are limitations for using these operations (e.g., for hash inner join hash table must entirely fit in memory)

However

- uses column-wise data compression and batch-mode => first step
- customer experiences even with this functionality show benefits of using column-store indexes

3. DBMS. OpenLink Virtuoso (1/3)

OpenLink Virtuoso:

Row/column-oriented DBMS, storing data in relational and graph forms

Main goal:

Serving both OLTP and OLAP-style queries, achieving memory usage efficiency, locality and bulk read throughput, keeping random reads and updates low-latent

Currently:

Has also commercial version, both of them are successfully used

3. OpenLink Virtuoso. Storage layout (2/3)

- any index of the table can be represented row or column-wise
- B-tree of a row-wise index has indexes at the top and values at the leaves
- B-tree of a column-wise index at the leaves has an array page numbers with correspondent column-wise compressed values of thousand of rows
- *segment* – rows that are stored under the same leaf
- the same page size of 8K for both column and row stores

3. OpenLink Virtuoso. Execution engine and system level (3/3)

- supports data compression and vectored execution
- the system provides complete support for transactions

Discussion

The system is not a column-store specific, however reasonable utilizes column-store performance and efficiency benefits

4. Conclusion

1. Pure column-oriented systems are meant for analytical workloads (e.g., ECOS, MonetDB, Vertica, Vectorwise and C-store)

2. Hybrid row/column-oriented systems are meant for performance gain for both transactional and analytical workloads.

   Some systems impose column-store on a row-store (e.g., SQL Server and OpenLink Virtuoso), while other implement both from scratch (e.g., HYRISE)

3. We believe that implementation from scratch may be more beneficial, however not always possible

References

1. M. Grund et. al., “HYRISE – a main memory hybrid storage engine,” *VLDB, vol. 4, Nov. 2010, pp. 105-116.*

2. S.S. Rahman, E. Schallehn, G. Saake, “ECOS: evolutionary column oriented storage,” *BNCOD 2011, pp. 18-32.*

3. M. Grund et. al., “HYRISE – a main memory hybrid storage engine,” *VLDB, vol. 4, Nov. 2010, pp. 105-116.*

4. P.-A. Larson, E.N. Hanson, S.L. Price, “Columnar storage in SQL Server 2012,” *IEEE Data Eng. Bull (DEBU) 35(1):15-20 (2012).*

Thank you!