
Column-Oriented Databases

Knowledge Objectives

1. Explain the three main tunings for row-stores to simulate column-stores
2. Name three specific features provided by column-stores that justify their existence in front of tuned row-stores

Understanding Objectives

1. Elaborate on late materialization
2. Elaborate on block iteration techniques for column-stores
3. Choose between techniques to find records

Application Objectives

- ❑ Apply Run-Length Encoding compression for column-stores
- ❑ Simulate a column-store on a row-store by means of index-only query answering, materialized views and vertical partitioning
 - Discuss the limitations of this approach compared to a column-store

Column-Oriented Databases

□ Main features

- Use, to the extreme, of vertical partitioning
- Do not allow variable record sizes and apply efficient compression techniques
- Specific query processing techniques assuming the items above
 - In-memory query processing

□ Recall the main limitations of vertical partitioning. On top of that, these DBs use compression as first-class citizen

- As such, these DBs are meant for **read-only databases**
- Extremely inefficient in front of write-intensive database workloads

Column-Oriented Databases

- ❑ Column-oriented DBs are inherently aligned with decisional systems
 - A **must** for Data Warehousing! ~ Terabytes
 - A **must** for read-only Big Data systems ~ Petabytes
 - ❑ Two main types:
 - Relational Column-Oriented DBs (aka NewSQL)
 - ❑ First system: C-Store
 - ❑ Industrial examples: MonetDB, HP Vertica, SAP Hana, Oracle in-memory column store, MariaDB ColumnStore PostgreSQL Zedstore... in general, ANY relational database provides, in one way or another, a columnar engine
 - Column-Oriented NOSQL databases
 - ❑ Apache Druid (first OLAP-like NOSQL engine) - first NOSQL
 - ❑ Hadoop Ecosystem
 - ❑ Apache Parquet and Apache Arrow file-formats for HDFS
 - ❑ Apache Kudu
 - ❑ Google BigQuery (former Dremel) and Amazon Redshift (former DynamoDB). In general, most Cloud Providers provide a column-oriented PaaS
- NOTE:* Many classify Apache HBase as a column-oriented DB. **IT IS NOT**. HBase applies a hybrid partitioning strategy with horizontal partitioning as its primary partitioning strategy. Therefore, it cannot apply to its whole the optimizations for query processing

Column-Oriented DBs

- ❑ Column-Oriented Specific Features:
 - Data model
 - ❑ Pure Vertical Partitioning
 - ❑ Tuples are identified by their position (no PK needed to be replicated in each fragment)
 - Multiple sorting of data (if needed, different in each replica)
 - ❑ Remove variable size records and work with fix-sized records (dictionaries or bitmaps needed)
 - ❑ Column-specific compression techniques
 - Specific query processing
 - ❑ Late materialization: apply as many processing operators to the vertical fragments before joining them
 - ❑ Block iteration: exploit the fix-sized records to process data per blocks
 - ❑ *Vectorized query processing*: when late materialization and block iteration are combined
 - ❑ Specific join algorithms (e.g., invisible join) exploiting the previous items

Activity: Column-Oriented Specific Features

- ❑ *Objective: Understand the column-oriented specificities that tuned row-oriented databases cannot meet*
- ❑ *Tasks:*
 - 1. (15') In group of two, each of you must read one of these specific features. Namely:*
 - I. Compression*
 - II. Late materialization and block iteration*
 - 2. (5') Think tank*

Data Model: Vertical Partitioning

- Most column-oriented DBs create a fragment per column

Table T

A	B	C
Bravo	Bravo	Bravo
Bravo	Bravo	Bravo
null	null	null
null	null	null
null	null	null
Charlie	Charlie	Charlie
Charlie	Charlie	Charlie
Charlie	Charlie	Charlie

Data Model: Vertical Partitioning

- Most column-oriented DBs create a fragment PER column

<u>Table T</u> (partitioned)	A	B	C
	Bravo	Bravo	Bravo
	Bravo	Bravo	Bravo
	null	null	null
	null	null	null
	null	null	null
	Charlie	Charlie	Charlie
	Charlie	Charlie	Charlie

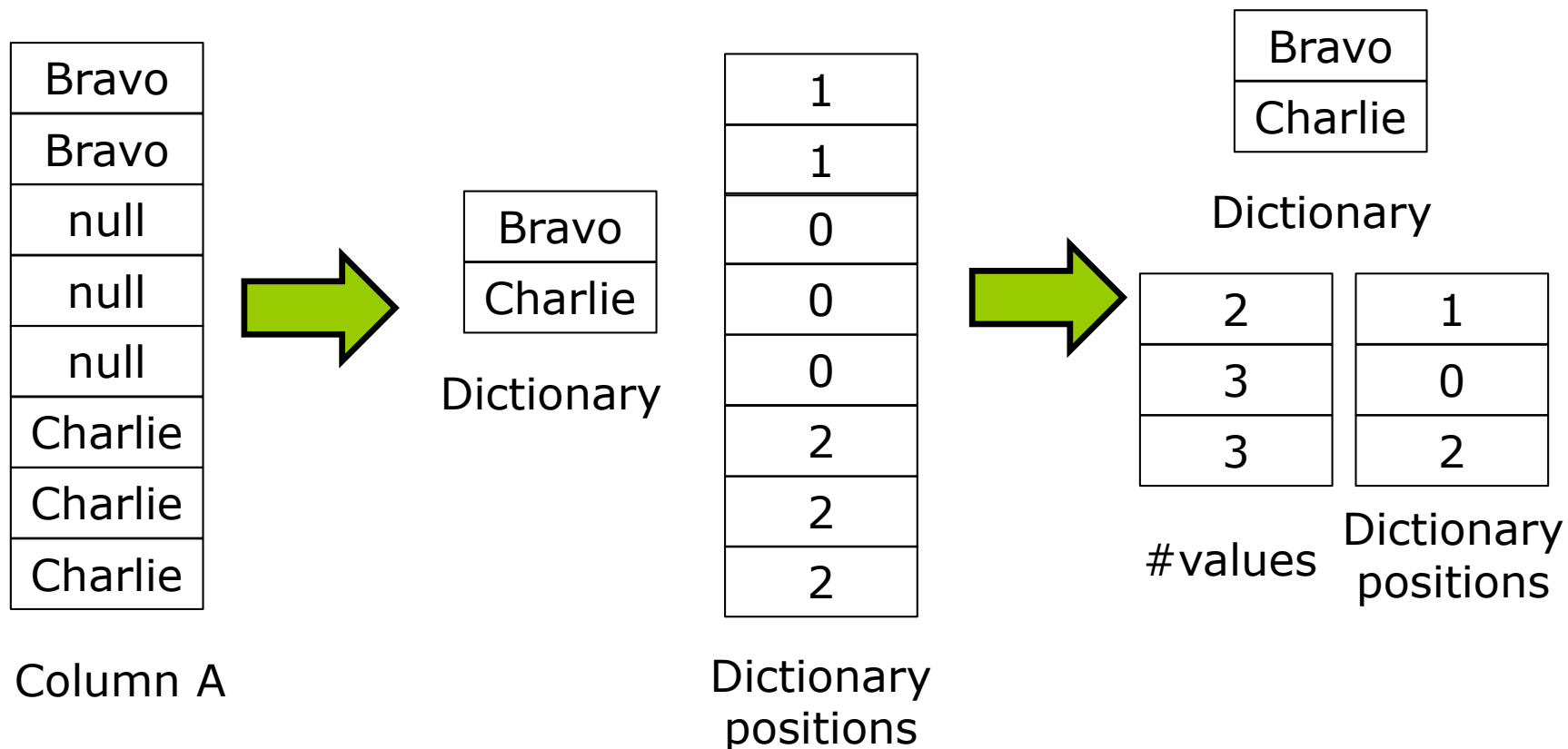
- Relevantly, some DBs allow to define groups of columns

Data Model: Compression

- ❑ Each column is not store as a regular file, but as a compressed file
- ❑ In these DBs, the compression main objective is not reducing data space but reducing I/Os
 - Yet, data in columnar format is more compressible than data stored in rows
 - ❑ High data value locality (less value entropy)
 - ❑ Benefits from sorting
- ❑ Two main trends
 - Heavy weight compression algorithms (e.g., Lempel-Ziv)
 - ❑ In general, not that useful but it might be if there is a (huge) gap between memory bandwidth and CPU performance
 - Lightweight compression (e.g., Run-Length Encoding) may allow the query optimizer work directly on compressed data
 - ❑ Improves performance by reducing CPU cost
 - ❑ Decompression is needed in front of bitwise AND / OR

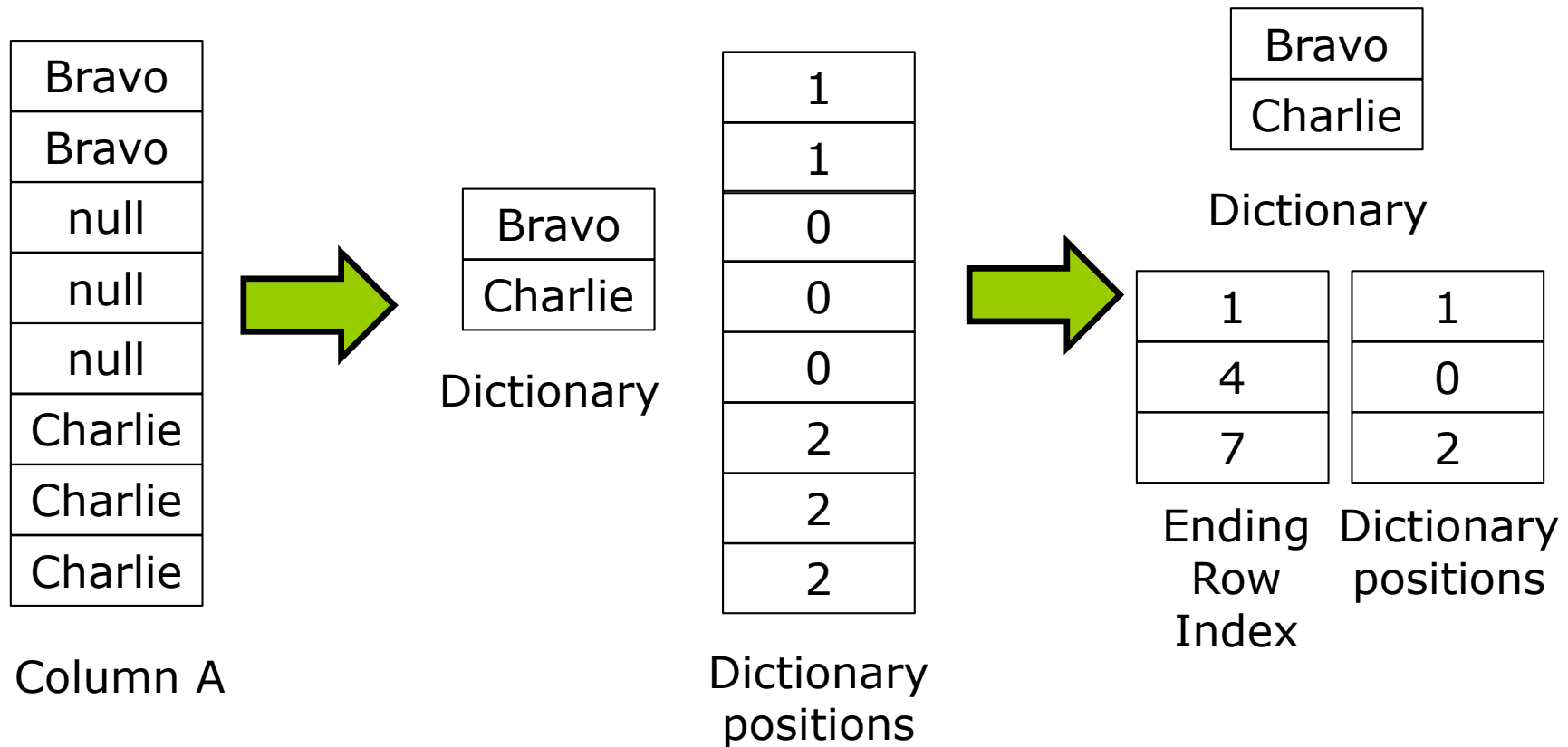
Data Model: Lightweight Compression

- Mainly based on dictionaries or bitmaps
- Example of Run-length Encoding (with dictionary)



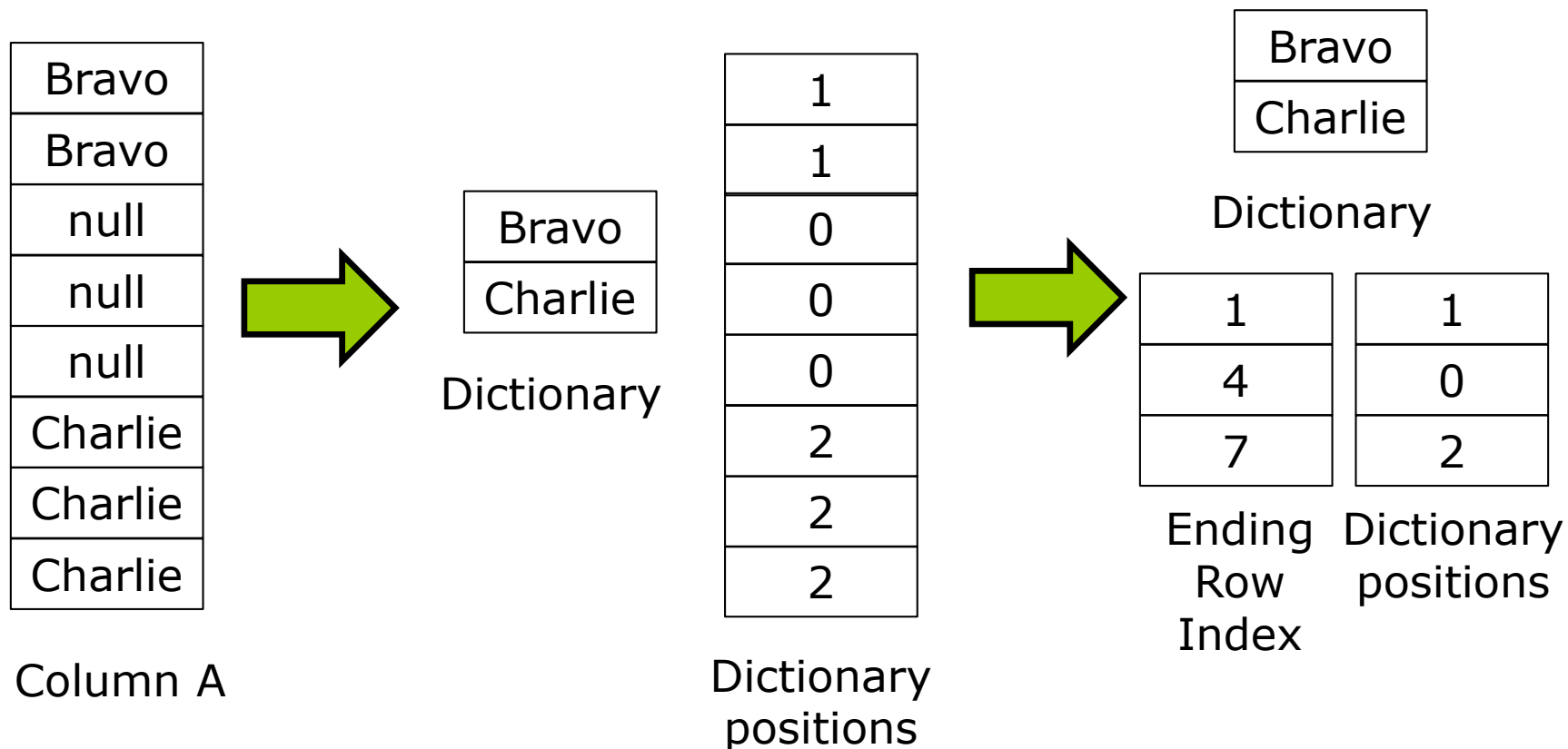
Data Model: Lightweight Compression

- Mainly based on dictionaries or bitmaps
- Example of Run-length Encoding (with dictionary)



Data Model: Lightweight Compression

- Mainly based on dictionaries or bitmaps
- Example of Run-length Encoding (with dictionary)



Realise that the ERI and Dictionary positions are always fixed-size

Data Model: Reconstructing Records

- ❑ From each table, $3 \times N$ vectors are generated ($N = \# \text{ attributes}$)
- ❑ To reconstruct the original tuple, the original vertical fragmentation used the PK
- ❑ Alternatively, in column-oriented DBs each record has the same position in all the columns (vectors) created from that table
 - Thus, column-oriented DBs do not join fragments to reconstruct the record
- ❑ Relevantly, a certain order might benefit a column and harm another
 - If necessary, in the presence of replicas, each replica might use a different order (however, the order must be the same for all fragments of a table in a given replica)

Activity: Data Model (I)

❑ *Objective: Understand Run-Length Encoding*

❑ *Tasks:*

- 1. (15') Apply the Run-Length Encoding for the table in the next slide (dictionary-based + Ending-Row Index)*

For this exercise, do not play with the order of the rows. Use the one given. At the end, identify those columns where a different order might have generated a more compact representation

- 1. (5') Think tank*

Activity: Data Model (I)

BookID	Date	Price	#ItemsBought
1	1/01/2012	19,99	1
99	1/01/2012	9,99	1
301	2/01/2012	19,99	1
44	2/01/2012	9,99	1
56	2/01/2012	9,99	1
1	2/01/2012	19,99	1
77	3/01/2012	9,99	2
8	3/01/2012	19,99	1
78	3/01/2012	9,99	1
10	3/01/2012	19,99	1

Data Model: Create Vertical Partitioning

- ❑ Creating a fragment per column, in general, is suboptimal. Thus, most advanced DBs allow the user to define partitions (each, potentially, as a set of columns)
- ❑ In these cases, finding the optimal vertical partitioning is a problem that must take into account the query workload
 - Identify correlations in the query workload and group those columns with high *affinity* (i.e., frequently queried together)
 - For m non-key attributes the search space is $B(m)$, i.e., the m th Bell number, which counts the possible partitions of a set of m elements
 - For large numbers, $B(m) \approx m^m$
- ❑ Two main approaches to compute the optimal partitioning
 - Grouping
 - ❑ Attribute affinity matrix
 - ❑ Clustering algorithms
 - ❑ ...
 - Splitting
- ❑ [RECAP] Partitioning must guarantee
 - Completeness
 - Disjointness
 - Reconstruction

Data Model: Create Vertical Partitioning

- ❑ Creating a fragment per column, in general, is suboptimal. Thus, most advanced DBs allow the user to define partitions (each, potentially, as a set of columns)
- ❑ In these cases, finding the optimal vertical partitioning is a problem that must take into account the query workload
 - Identify correlations in the query workload and group those columns with high *affinity* (i.e., frequently queried together)
 - For m non-key attributes the search space is $B(m)$, i.e., the m th Bell number, which counts the possible partitions of a set of m elements
 - For large numbers, $B(m) \approx m^m$
- ❑ Two main approaches to compute the optimal partitioning
 - Grouping
 - ❑ Attribute affinity matrix
 - ❑ Clustering algorithms
 - ❑ ...
 - Splitting
- ❑ [RECAP] Partitioning must guarantee
 - Completeness
 - **Disjointness (for the sake of performance, it might be sacrificed)**
 - Reconstruction

Example

Schema:

```
Compres(llibreIdFK, date, preu, numUnitats)  
Llibre(llibreId, autor, any, editorial, ISBN)
```

Queries:

```
SELECT llibreId, SUM(numUnitats) FROM compres c, llibre l  
WHERE c.llibreId = l.llibreId AND editorial = 'RBA'  
GROUP BY llibreId
```

```
SELECT editorial, AVG(preu) FROM compres c, llibres l  
WHERE c.llibreId = l.llibreId  
GROUP BY editorial
```

```
SELECT AVG(numUnitats) FROM compres c  
WHERE date BETWEEN '01/01/xxxx' AND '31/12/XXXX'  
GROUP BY llibreID
```

```
SELECT autor, any, COUNT(*) FROM llibre l  
GROUP BY autor, any
```

Attribute Affinity Matrix

□ Algorithm:

1. For each relation, generate the **attribute usage matrix**

	<i>llibreId</i>	<i>date</i>	<i>preu</i>	<i>numUnitats</i>
<i>Q1</i>	1	0	0	1
<i>Q2</i>	1	0	1	0
<i>Q3</i>	1	1	0	1
<i>Q4</i>	0	0	0	0

Now, create the attribute usage matrix for *Llibre*

Attribute Affinity Matrix

□ Algorithm:

2. For each relation, generate the **attribute affinity matrix**

- Consider Q1 frequency is 50%, Q2 10%, Q3 30%, Q4 10%
- For a pair of attributes A_i, A_j , compute its affinity by adding up all the frequencies in which they appear together

	<i>llibreId</i>	<i>date</i>	<i>preu</i>	<i>numUnitats</i>
Q1	1	0	0	1
Q2	1	0	1	0
Q3	1	1	0	1
Q4	0	0	0	0

Query-Attribute Usage Matrix

Query-Attribute Affinity Matrix

	<i>llibreId</i>	<i>date</i>	<i>preu</i>	<i>numUnitats</i>
<i>llibreId</i>	90	30	10	80
<i>date</i>	30	30	0	30
<i>preu</i>	10	0	10	0
<i>numUnitats</i>	80	30	0	80

Now, create the attribute affinity matrix for *Llibre*

Attribute Affinity Matrix

□ Algorithm:

3. For each matrix, reorganize the attribute orders to form clusters where the attributes in each cluster show high affinity to one another

	<i>llibreId</i>	<i>numUnitats</i>	<i>preu</i>	<i>date</i>
<i>llibreId</i>	90	80	10	30
<i>numUnitats</i>	80	80	0	30
<i>preu</i>	10	0	10	0
<i>date</i>	30	30	0	30

Now, do the same for *Llibres*

Attribute Affinity Matrix

□ Algorithm:

3. For each matrix, reorganize the attribute orders to form clusters where the attributes in each cluster show high affinity to one another

	<i>llibreId</i>	<i>numUnitats</i>	<i>preu</i>	<i>date</i>
<i>llibreId</i>	90	80	10	30
<i>numUnitats</i>	80	80	0	30
<i>preu</i>	10	0	10	0
<i>date</i>	30	30	0	30

Now, do the same for *Llibres*

Attribute Affinity Matrix

□ Output:

- P1: LlibreID, numUnitats
- P2: Preu
- P3: Date

□ Assess the result. Compute the **effective read ratio** for each query and partition

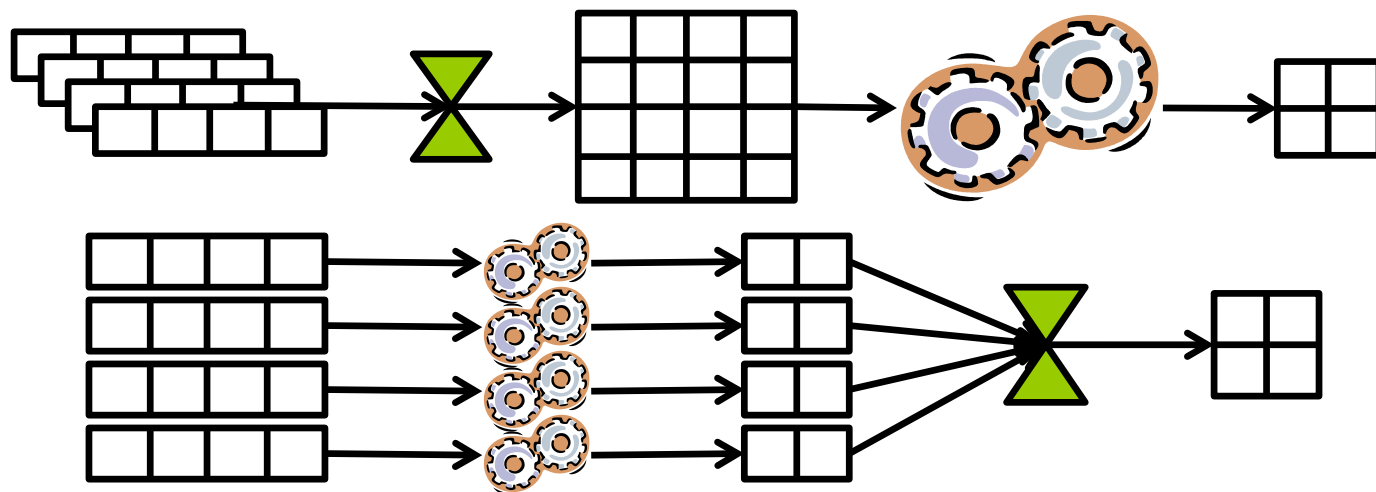
- For each query, for each partition it must read, compute the following ratio:

$$\#Atr(Q_j, P_i) / \#Atr(P_i)$$

- Where $\#Atr(Q_j, P_i)$ means the number of attributes Q_j must read from P_i and $\#Atr(P_i)$ is the total number of attributes in P_i
- Example: $RR(Q1, P1): 2/2 = 1$

Data Processing: Late Materialization

- *Tuple reconstruction* can be done at the beginning or at the end of the query



- **Advantages**

- Some tuples do not need to be constructed (because of selections and projections)
- Some columns remain compressed more time
- Cache performance is improved (kept at column level)
- Helps block iteration for values of fixed length columns

Data Processing: Block Iteration

- ❑ Blocks of values of the same column are passed to the next operation in a single function call
- ❑ Values inside the block can be:
 - Iterated as in an array (fixed-width)
 - Remain compressed together
 - ❑ Not necessarily using multiples of 8 bits
 - ❑ I can count or even identify the tuples for which the predicate is true
 - Exploits parallelism / pipelining

1
4
7

Ending
Row
Index

1
0
2

Dictionary
positions

Summary

- ❑ Advantages of column-oriented DBs
 - Bring into memory only relevant data
 - Provide fewer and simpler internal functions
 - Easier to recognize all execution strategies
 - Simpler tuning required by users
 - More reliable
- ❑ Daniel Abadi & Michael Stonebraker (M.I.T.)
 - C-Store (blueprint)
 - MonetDB (academic edition, open-source)
 - ❑ <http://www.monetdb.org/Home>
 - Vertica (commercial version, bought by HP)
 - ❑ <http://www.vertica.com/>

Bibliography

- Daniel J. Abadi, Samuel R. Madden and Nabil Hachem. Column-Stores Vs. Row-Stores: How Different Are They Really?
Proceedings of SIGMOD 2008, pp. 967-980
- Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widow. *Database Systems*. Pearson Prentice Hall, 2009
- Hasso Plattner and Alexander Zeier. *In-Memory Data Management*. Springer, 2011
- George P. Copeland , Setrag N. Khoshafian. A Decomposition Storage Model.
Proceedings of SIGMOD 1985