7 Beyond Row Stores

Column-Stores

- Are also based on the relational data model
  - use of relational algebra (and their operators) as a foundation
  - Column stores are (just) an alternative physical data model
- Follows a logical query plan similar to row-stores model

- Column-based compression allows for data processing without decompression
  - Lossless compression techniques (e.g. Lempel-Ziv and derivates) → identical (uncompressed) values imply identical compressed representation, i.e. value within search predicate is compressed beforehand and subsequently compared with the compressed representation

  - Order-preserving compression schemes (like RLE) → search values are directly comparable or similar mechanism to lossless-compression, i.e. aggregate functions like MIN/MAX or SUM can be processed in a compressed format.

> Column-Store Operators

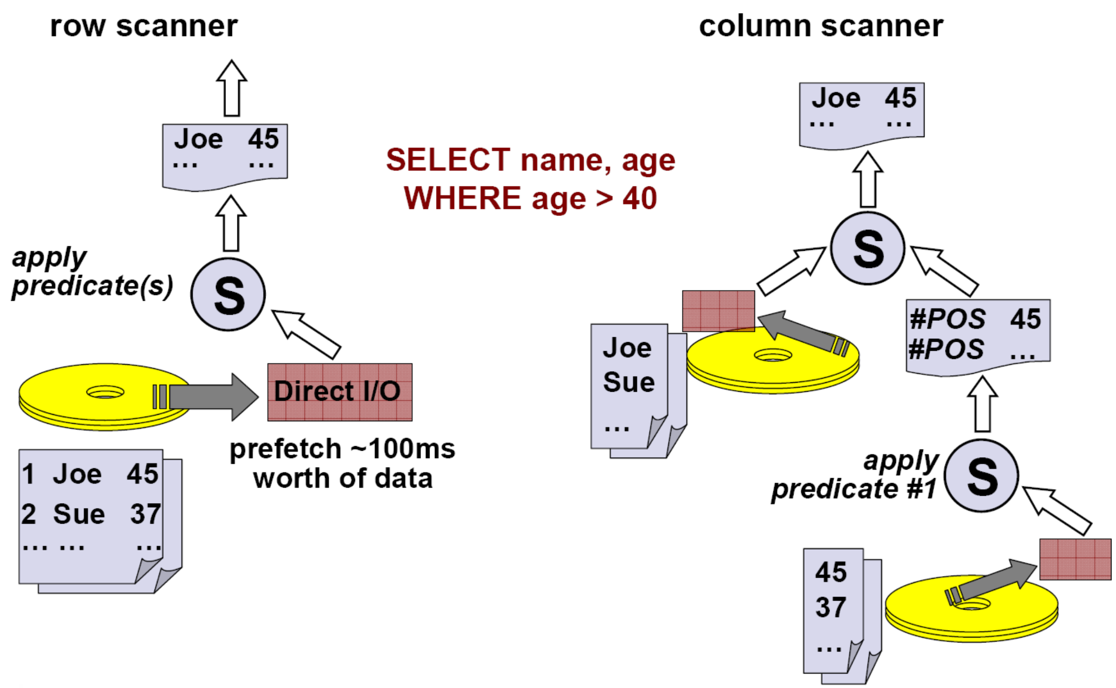
Column scan operators

- Translate value position information into disk locations
- Combine and reconstruct (when needed) partial or entire tuples out of different columns (*Materialization*)

Join operators

- Can either rely on column-scanners for receiving reconstructed tuples, or they can operate directly on columns by first computing a join index and then fetching qualifying value

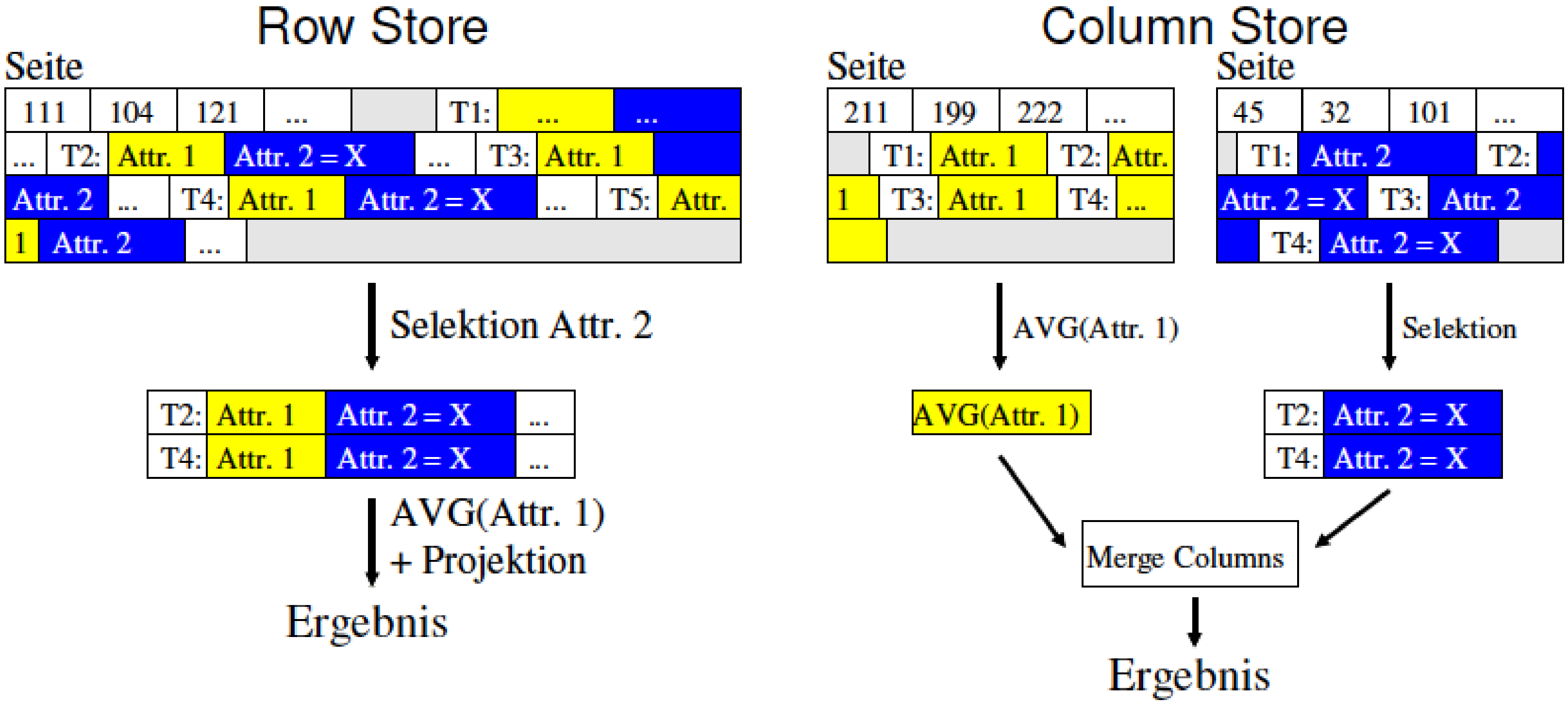
> Row-Scanner versus Column-Scanner



- Reads from a single file
- Iterates over pages, for each page iterates over tuples and applies predicates

- Must read as many files as are columns specified in the query
- Series of pipelined scan nodes
- Applies predicates by reading a column, creating {position, value} pairs for all qualified tuples
- Attaches values corresponding to input positions from other columns

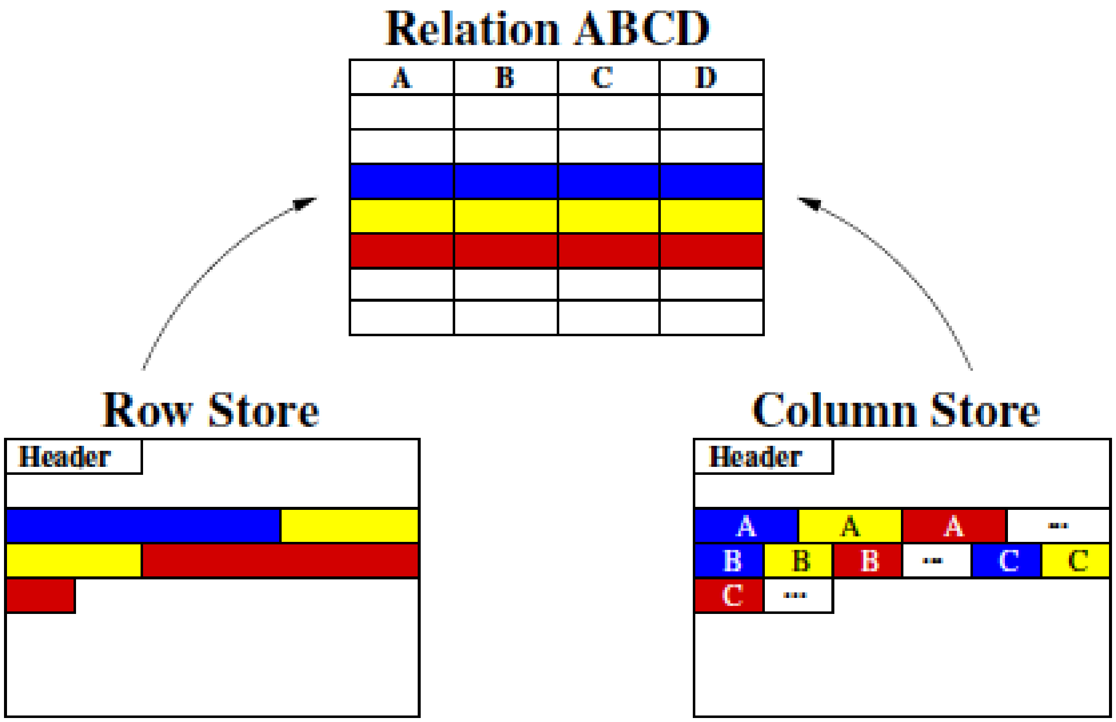
```
SELECT AVG(Attr. 1), Attr. 2 FROM Tabelle
WHERE Attr. 2=X;
```



> Tupelrekonstruktion

Operator

- Heißt SPC (Scan, Predicate, Construct) für volle Tupelrekonstruktion
- Merge ist ein k-tuplige Rekonstruktion (mit Spalten VAL1, ..., VALk)



Materialisierungszeitpunkt

Frühe Materialisierung (EM)

- Anfrageverarbeitung sehr nah an Row-Stores
- Aggregatfunktionen auf einzelnen Columns
- Tupelrekonstruktion sobald Tupel verwendet
- Zumeist Verwendung bei tupel-orientierter Anfragebearbeitung

Späte Materialisierung (LM)

- So lang wie möglich auf Columns arbeiten
- Mehrfacher Zugriff auf Basistabellen und/oder Zwischenergebnisse
- Folge: Anfrageplan kein Baum mehr
- Aber: Gleichzeitig Bearbeitung auf komprimierten und unkomprimierten Daten möglich
- Notwendig für effektive spalten-orientierte Anfragebearbeitung

Query Processing -
Tuple Materialization

> When should columns be projected?

Row-store

- Column projection involves removing unneeded columns from tuples
- Generally done as early as possible

Column-store

- Operation is almost completely opposite from a row-store
- Column projection involves reading needed columns from storage and extracting values for a listed set of tuples (Materialization)

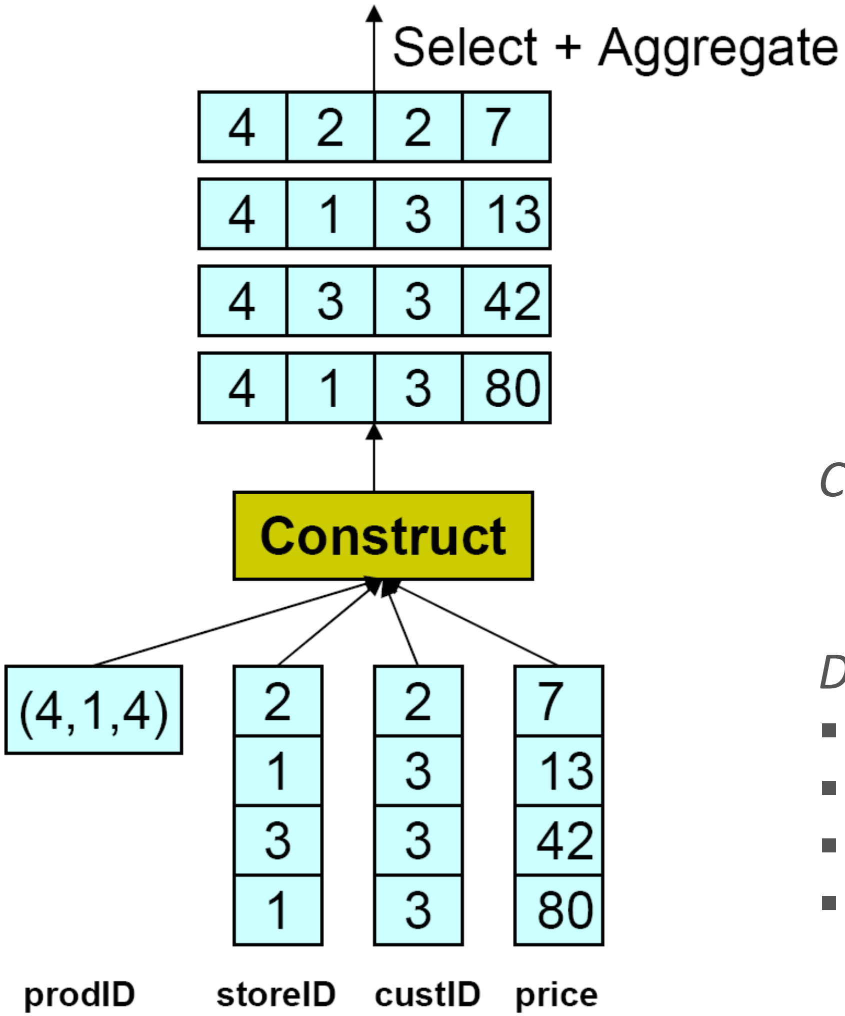
Early materialization

- Project columns at beginning of query plan
- Straightforward since there is a one-to-one mapping across columns

Late materialization

- Wait as long as possible for projecting columns
- More complicated since selection and join operators on one column obfuscates mapping to other columns from same table

> Early Materialization



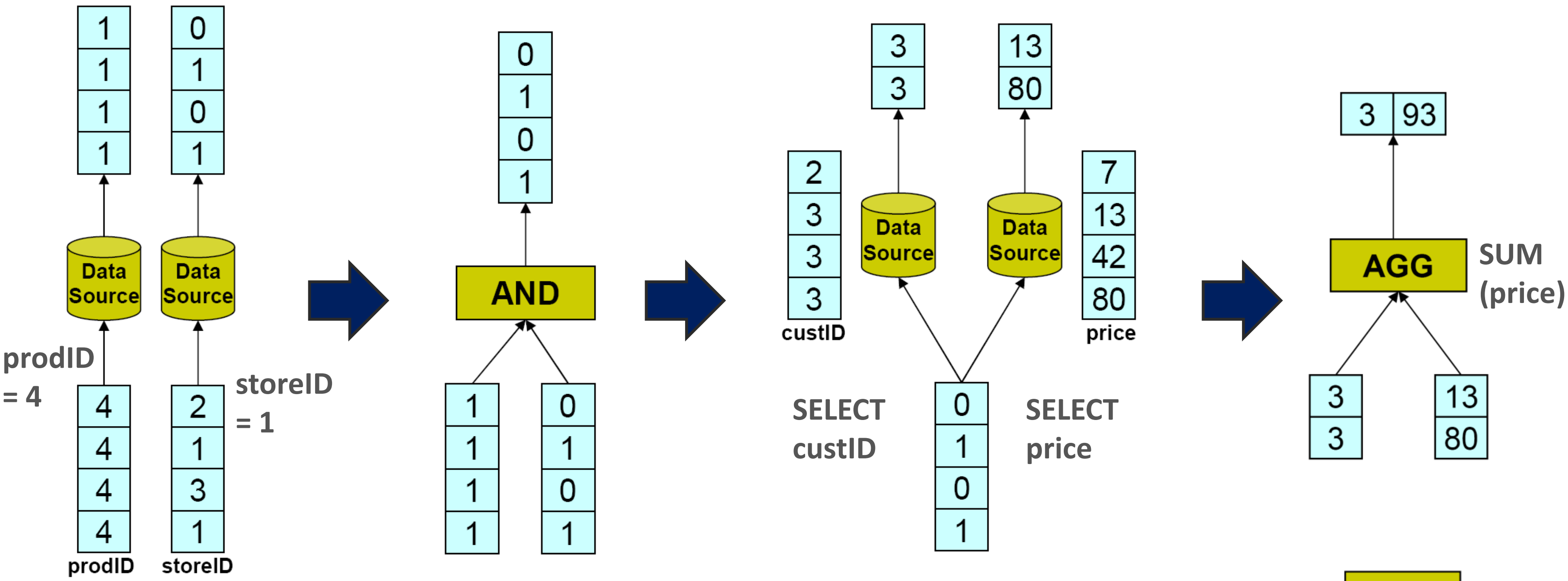

```
QUERY:

SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1) AND
GROUP BY custID
```

Create rows first

Disadvantages:

- Need to construct all tuples
- Need to decompress data
- Poor memory bandwith utilization
- Loose opportunity for vectorized operation

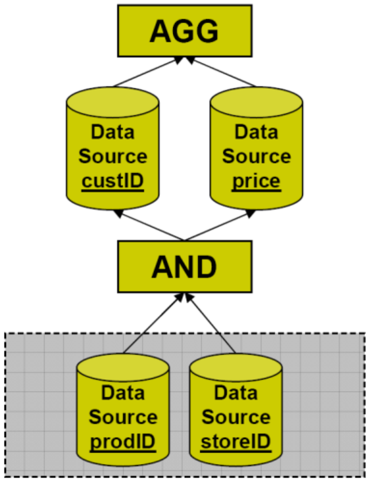
> Late Materialization Example



QUERY:

SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
(storeID = 1) AND
GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80



> Late Materialization

Operate directly on columns

- Intermediate “position” lists need to be constructed in order to match up operations that have been performed on different columns

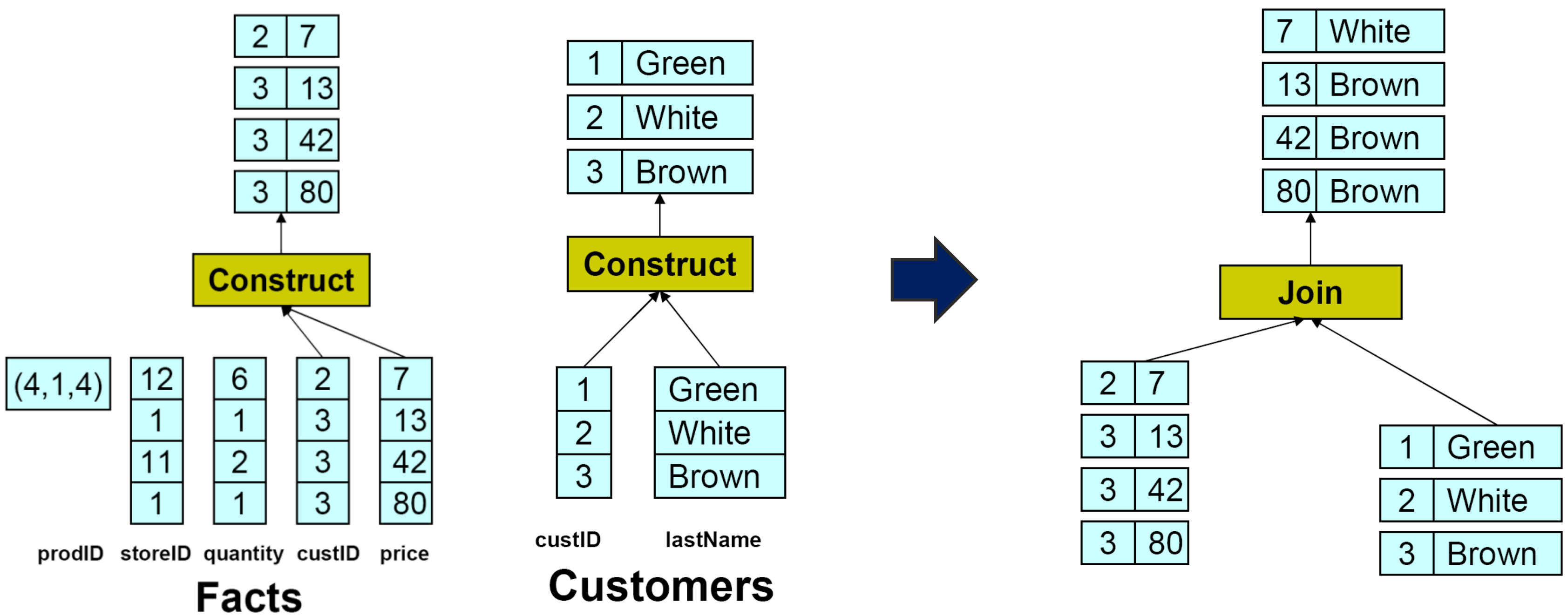
Advantages

- *Construct only relevant tuples, avoid unnecessary tuple construction*
- *Column can be kept compressed in memory*
  - Operating directly on compressed columns possible
- *Looping through column-oriented data tends to be faster than looping through tuples*
  - Values of the same column fill an entire cache line
  - Vector processing for column block accesss

Disadvantage

- *Columns may need to be accessed multiple times in a query plan*
  - Trade-off between late materialization optimizations and column reaccess costs

Early Materialization Join



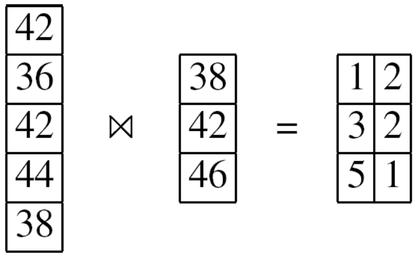
QUERY:

SELECT C.lastName,SUM(F.price)
FROM facts AS F, customers AS C
WHERE F.custID = C.custID
GROUP BY C.lastName

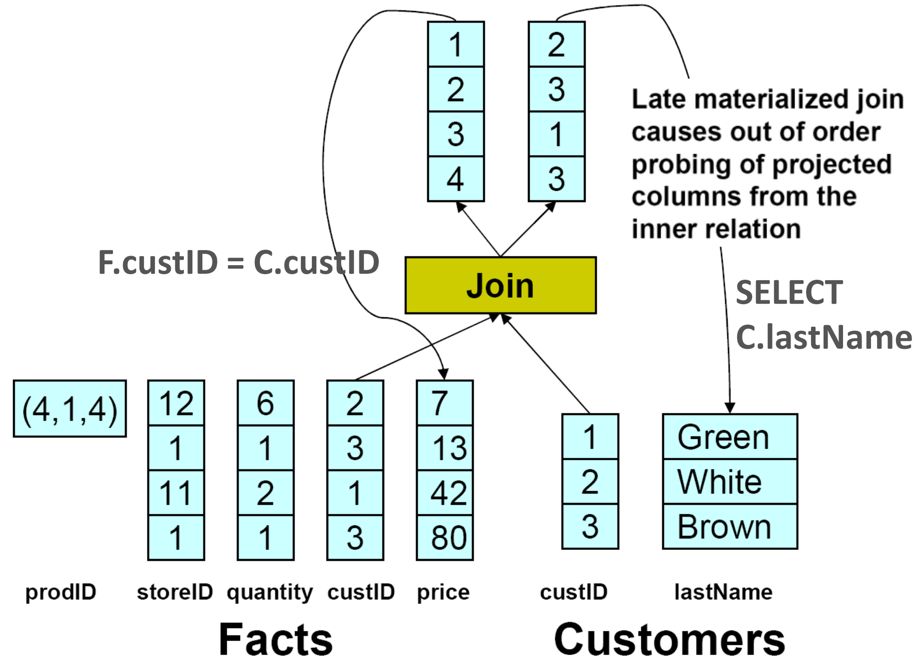
- Tuples have already been constructed before reaching the join operator
- Join functions as it would in a standard row-store system and outputs tuples

> Late Materialization Join

Results in two sets of positions, one for the fact table and one for the dimension table



- Indicates which pairs of tuples passed the join predicate
- At most one position list is produced in sorted order (fact table)
  - Merge join of positions can be used to extract other column values
- Values from dimension table need to be extracted in out-of-position order
  - Can be significantly more expensive




```
QUERY:

SELECT C.lastName,SUM(F.price)
FROM facts AS F, customers AS C
WHERE F.custID = C.custID
GROUP BY C.lastName
```

Invisible Join

Properties

- Late materialized join, minimizes the values that need to be extracted out-of-order
- Makes sure that the table that can be accessed in position order is the fact table for each join
- Rewrites joins into predicates on the foreign key columns in the fact table
- Position lists from the fact table are then intersected (in-position order)
- Reduces the amount of data that must be accessed out of order from the dimension tables

Phase 1

- For each predicate dimension table keys are extracted which satisfy the predicate
- Keys are used to build a hash table

Apply "region = 'Asia'" On Customer Table

custkey	region	nation	…
1	ASIA	CHINA	…
2	EUROPE	FRANCE	…
3	ASIA	INDIA	…

→ **Hash Table Containing Keys 1 and 3**

Apply "region = 'Asia'" On Supplier Table

suppkey	region	nation	…
1	ASIA	RUSSIA	…
2	EUROPE	SPAIN	…

→ **Hash Table Containing Key 1**

Apply "year in [1992,1997]" On Date Table

dateid	year	…
01011997	1997	…
01021997	1997	…
01031997	1997	…

→ **Hash Table Containing Keys 01011997, 01021997, and 01031997**

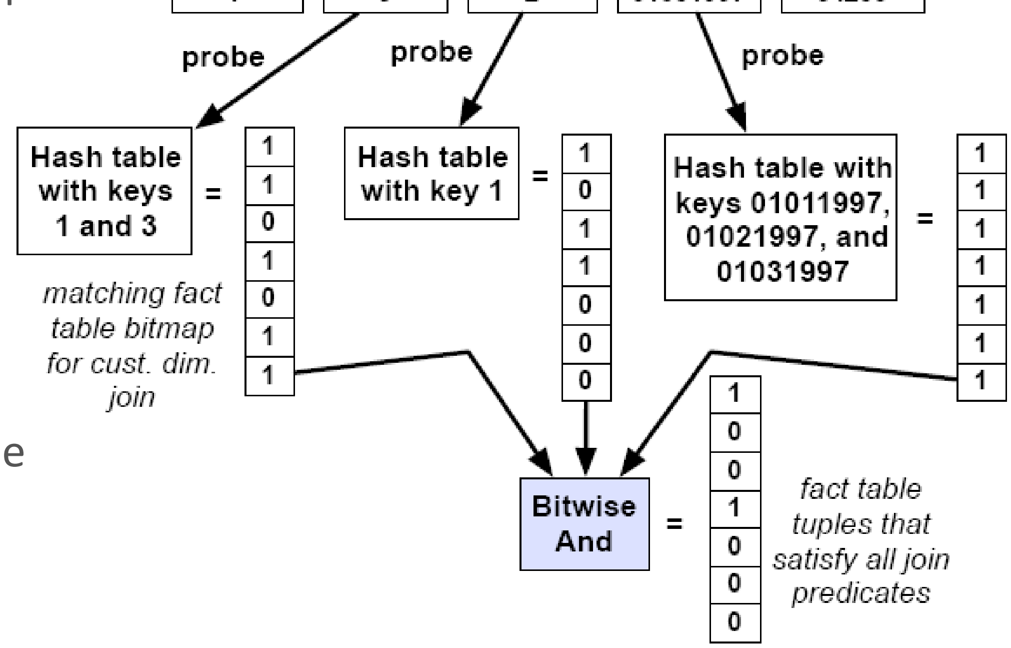
> Invisible Join

Phase 2

- Hash table is used to extract the positions of records in the fact table that satisfy the corresponding predicate
- Each value in the foreign key column of the fact table is probed into the hash table
- Results in a list of all positions in the foreign key column that satisfy the predicate
- Lists from all of the predicates are intersected to generate a list of satisfying positions P in the fact table

Fact Table

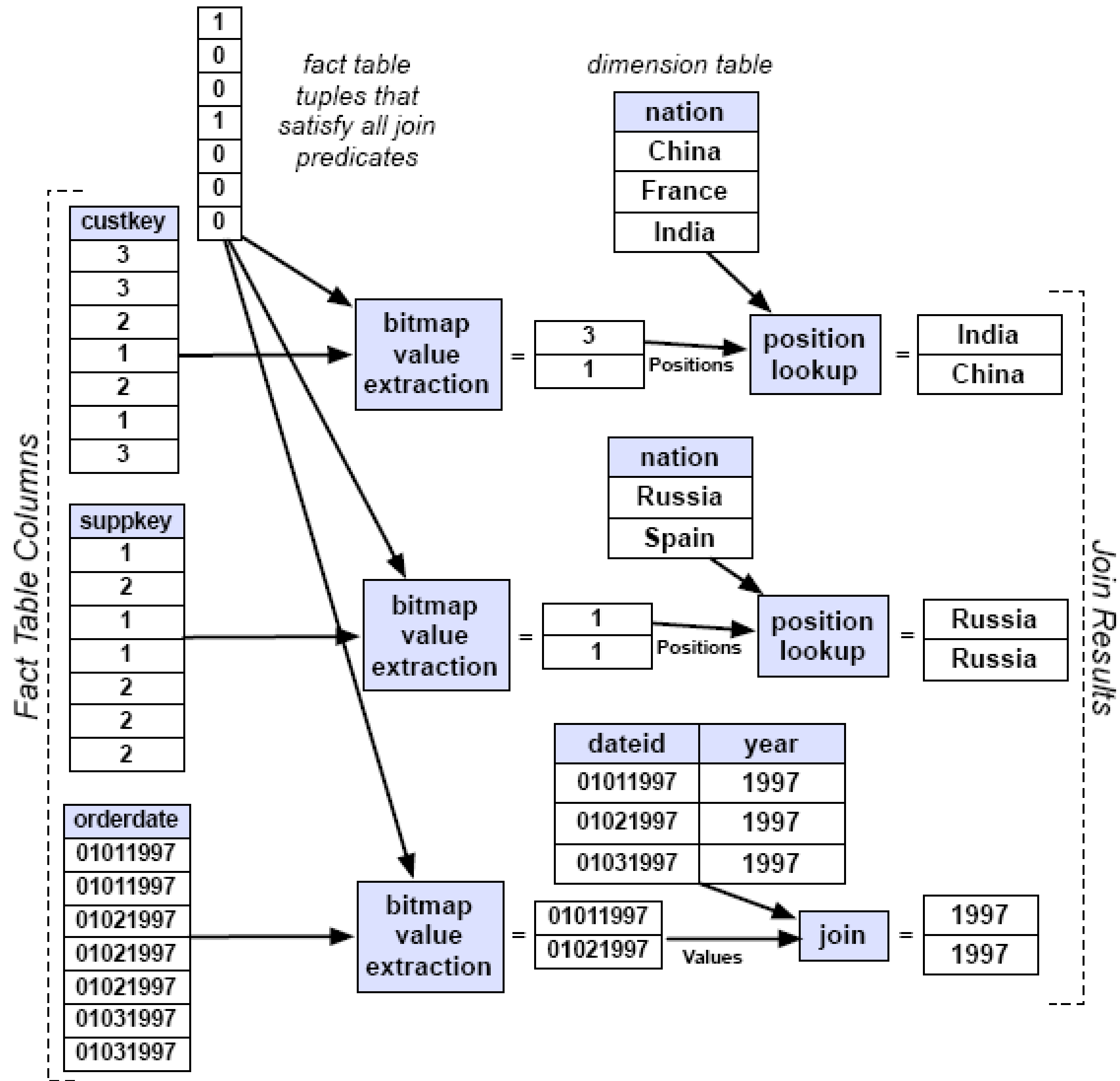
orderkey	custkey	suppkey	orderdate	revenue
1	3	1	01011997	43256
2	3	2	01011997	33333
3	2	1	01021997	12121
4	1	1	01021997	23233
5	2	2	01021997	45456
6	1	2	01031997	43251
7	3	2	01031997	34235



> Invisible Join

Phase 3:

- The third phase uses the list of satisfying positions *P* in the fact table to get foreign key values and hence needed data values from the corresponding dimension table.



Which Column-Specific Optimization is most significant?

Most significant?

Late materialization

- Improves performance by a factor of 3.

Block iteration

- Improves performance by 5% - 50%.

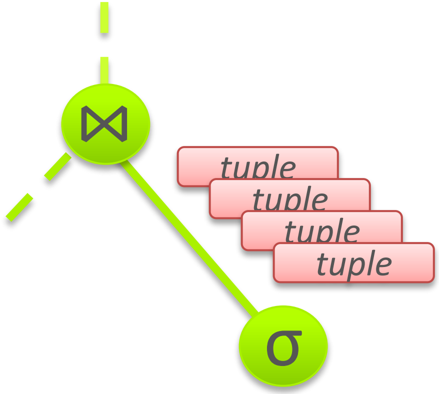
Column-specific compression

- Improves performance by a factor of 2 on average
- Improves performance by a factor of 10 on queries that access sorted data.

Invisible join on star schema

- Improves performance by 50% - 70%

> Summary

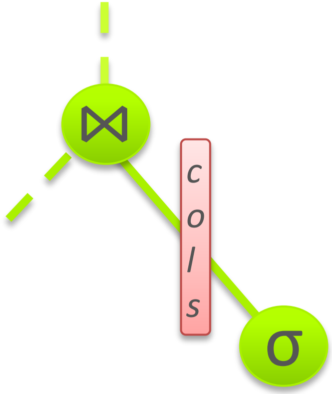


tuple(s)-at-a-time

- \+ No tuple reconstruction
- – Massive virtual function calls
- – High materialization costs

Row-Stores → *Update friendly*

Column-Stores → *Less update friendly*



column-at-a-time

- \+ Low materialization costs
- – High tuple reconstruction costs
- – Single “next” call

Vectorized Execution

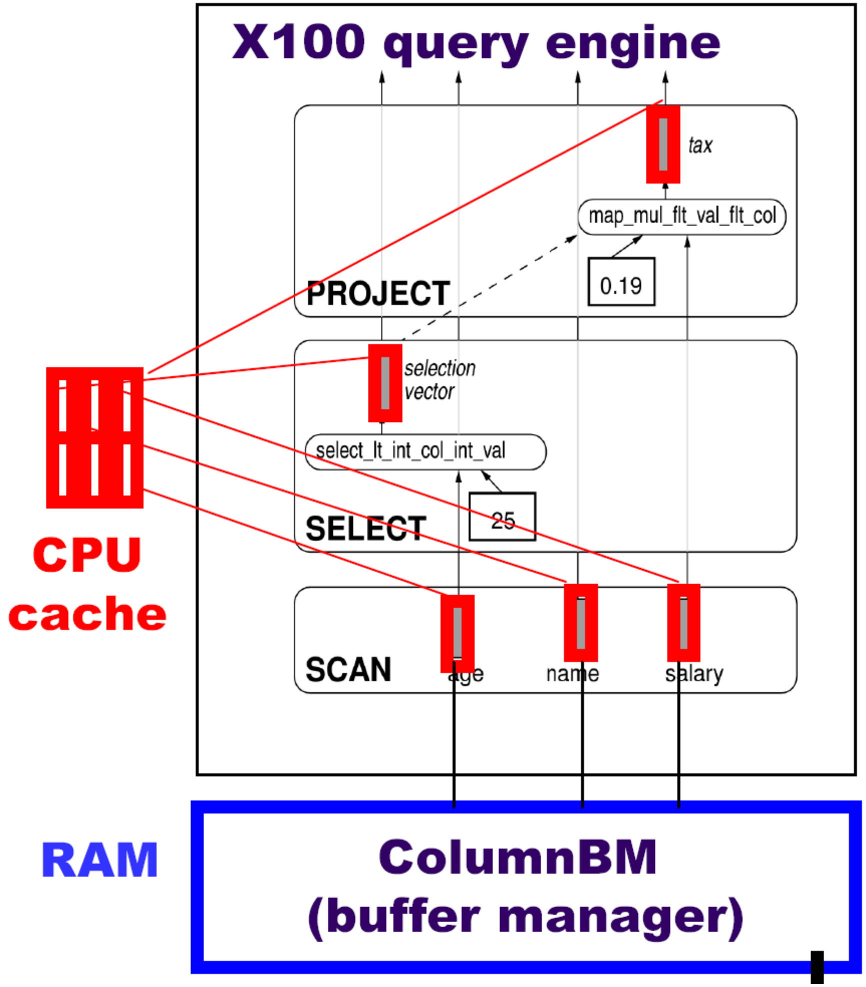
> Vectorized Processing

Block Iteration

- Blocks of values from the same column are sent to an operator in a single function call
- If the column is fixed-width, these values can be iterated through directly as an array
- Advantages
  - Minimizes per-tuple overhead
  - Exploits potential for parallelism on modern CPUs, e.g. loop-pipelining techniques

Example: MonetDB/X100

- New hyper-pipelining query execution engine for MonetDB
- In-cache vectorized processing
- MonetDB materializes all intermediate results -> limits scalability
- Volcano iterator pipeling model: used in most row-stores
  - But: tuple-at-a-time processing
- MonetDB/X100 combines benefits of low-overhead column-wise query execution with the absence of intermediate result materialization in the Volcano iterator model



Combines Volcano model with vector processing

All vectors together should fit the CPU cache

Optimizer tune vector size, given the query characteristics.

Vectors are compressed

- Decompression between RAM and CPU cache

In-cache execution

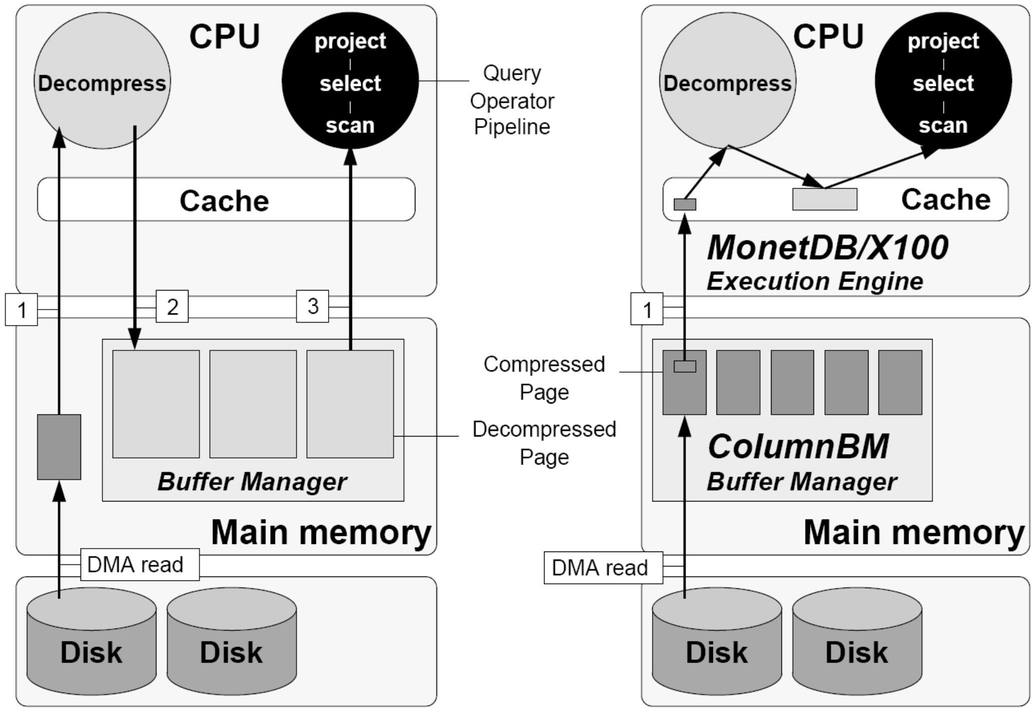
- Only „randomly“ accessible memory is the CPU cache
- Main memory only used in the buffer manager for buffering I/O operations and large intermediate results

Vectorized execution

- Vertically decomposed tables are further partitioned horizontally into small chunks called *vectors*
- A set of aligned vectors (one for each attribute), representing a set of tuples, is a single data unit in the execution pipeline
- An optional selection vector contains the positions of tuples currently taking part in processsing
- The control logic of operators is common for all data types, arithmetic functions, predicates etc.
- The actual data processing in the operators is performed by a set of *execution primitives* – simple, specialized and CPU-efficient functions

RAM-CPU Cache Compression

- Compression/decompression is used on the boundary between the CPU cache and RAM storage levels (instead of disk and RAM)
- Cache pages in the buffer manager (i.e. in RAM) in compressed form.
- Tuple values are decompressed at a small granularity (such that they fit the CPU cache) just-in-time, when the query processor needs them.



Advantages

- RAM-Cache boundary only crossed once
- More (compressed) data in RAM
- Less bandwith use