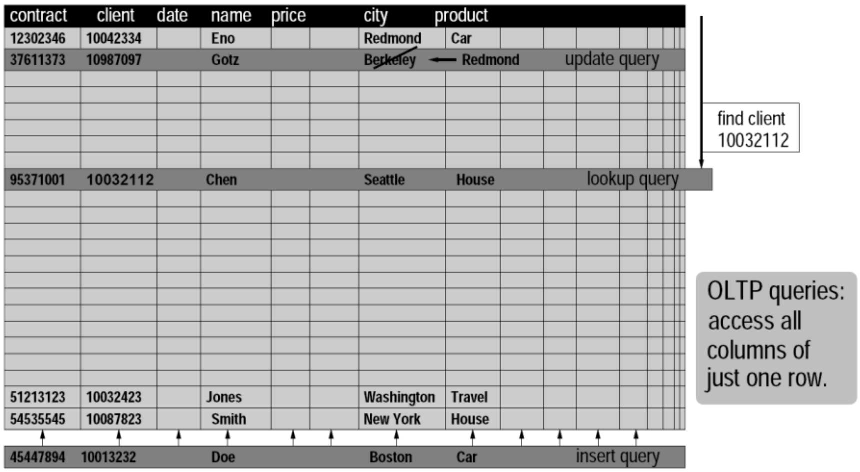
6 Column-based Record Management

> Different Access Characteristics

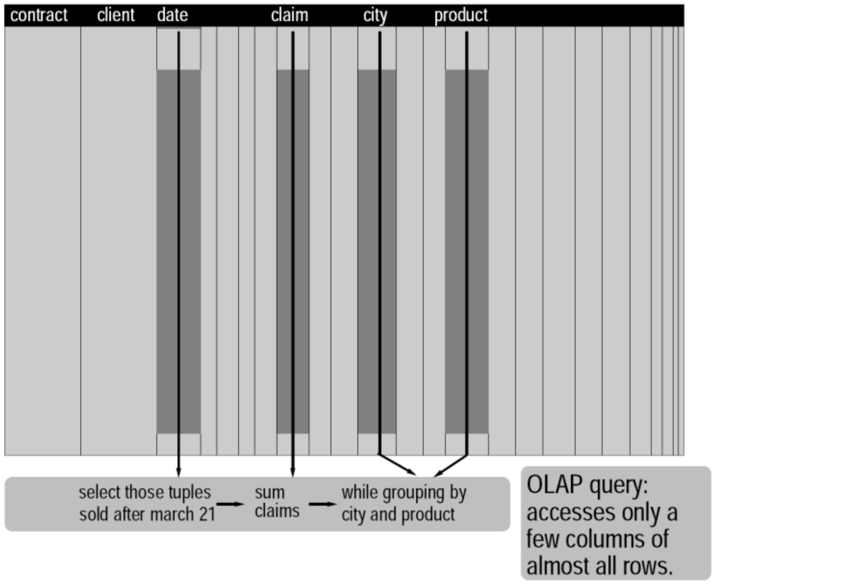
OLTP (On-line Transaction Processing)

- Mix between read-only and update queries
- Minor analysis tasks
- Used for data preservation and lookup
- Read typically only a few records at a time
- High performance by storing contiguous records in disk pages



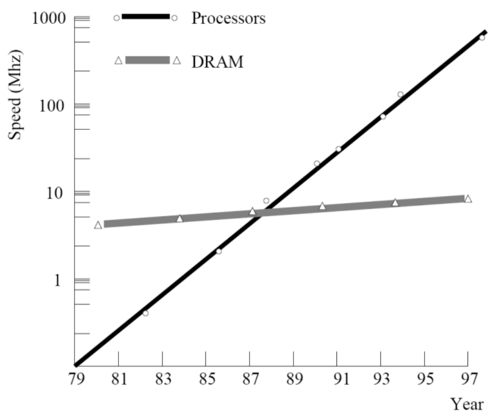
OLAP (On-line Analytical Processing)

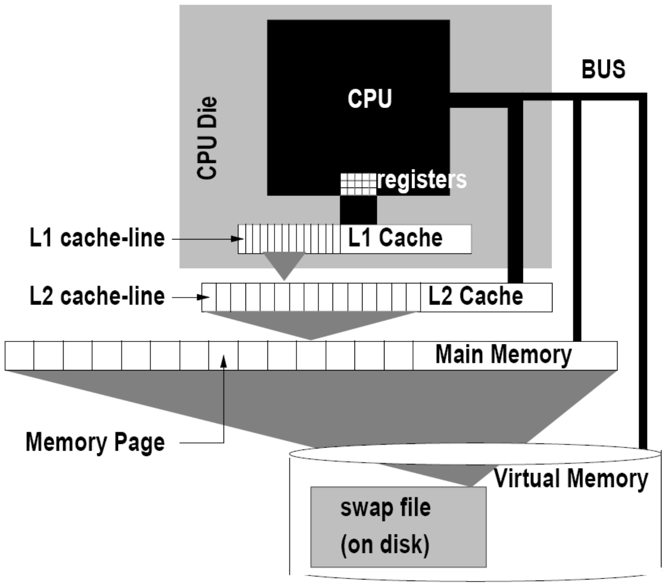
- Query-intensive DBMS applications
- Infrequent batch-oriented updates
- Complex analysis on large data volumes
- Read typically only a few attributes of large amounts of historical data in order to partition them and compute aggregates
- High performance by storing contiguous values of a single attribute



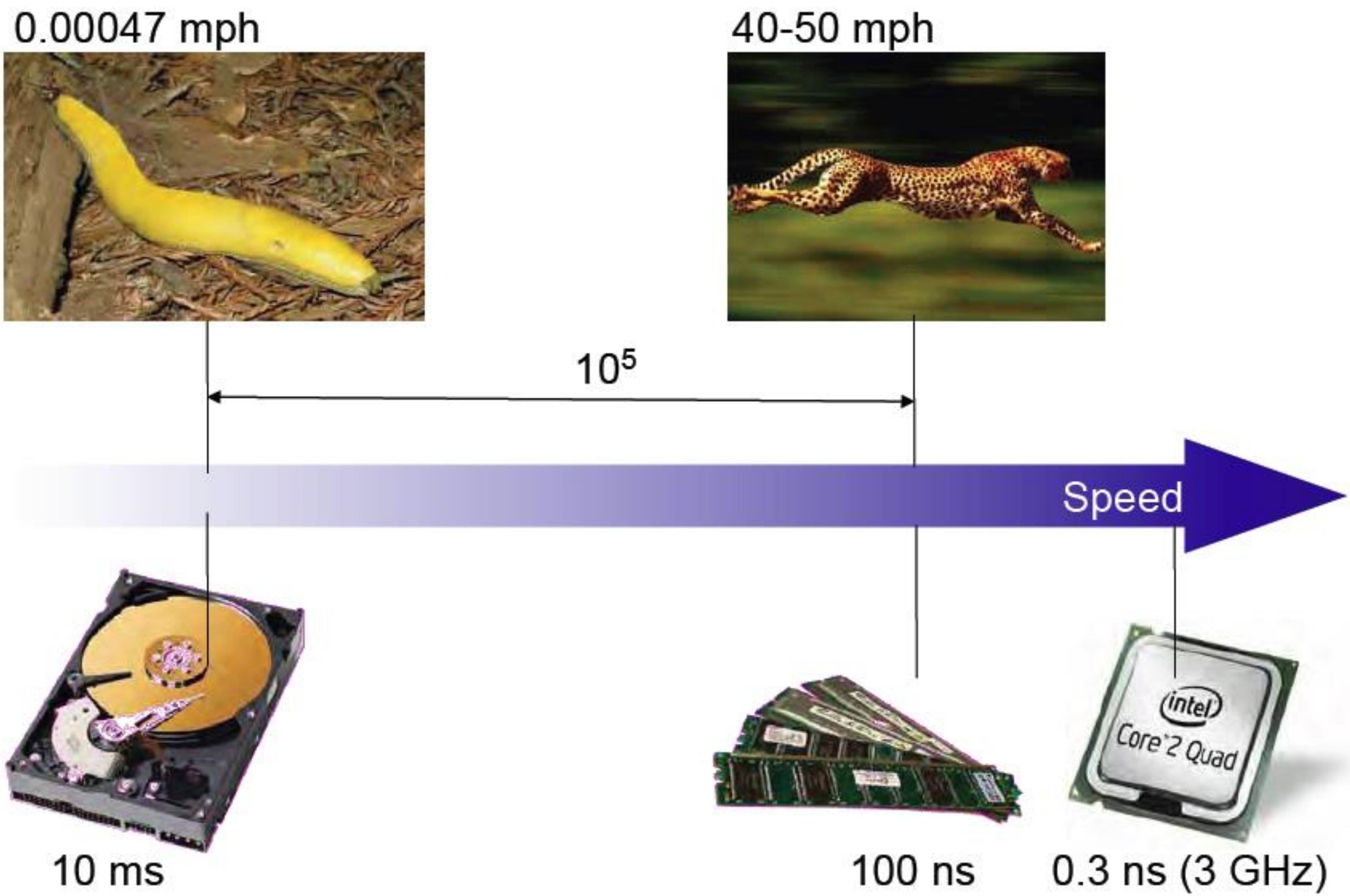
- Hardware improvements not equally distributed
- Advances in CPU speed have outpaced advances in RAM latency
- Main-memory access has become a performance bottleneck for many computer applications
  - Bandwidth
  - Latency
  - Adress translation (TLB)

  → Memory Wall
- Cache memories can reduce the memory latency when the requested data is found in the cache.
- Vertically fragmented data structures optimize memory cache usage

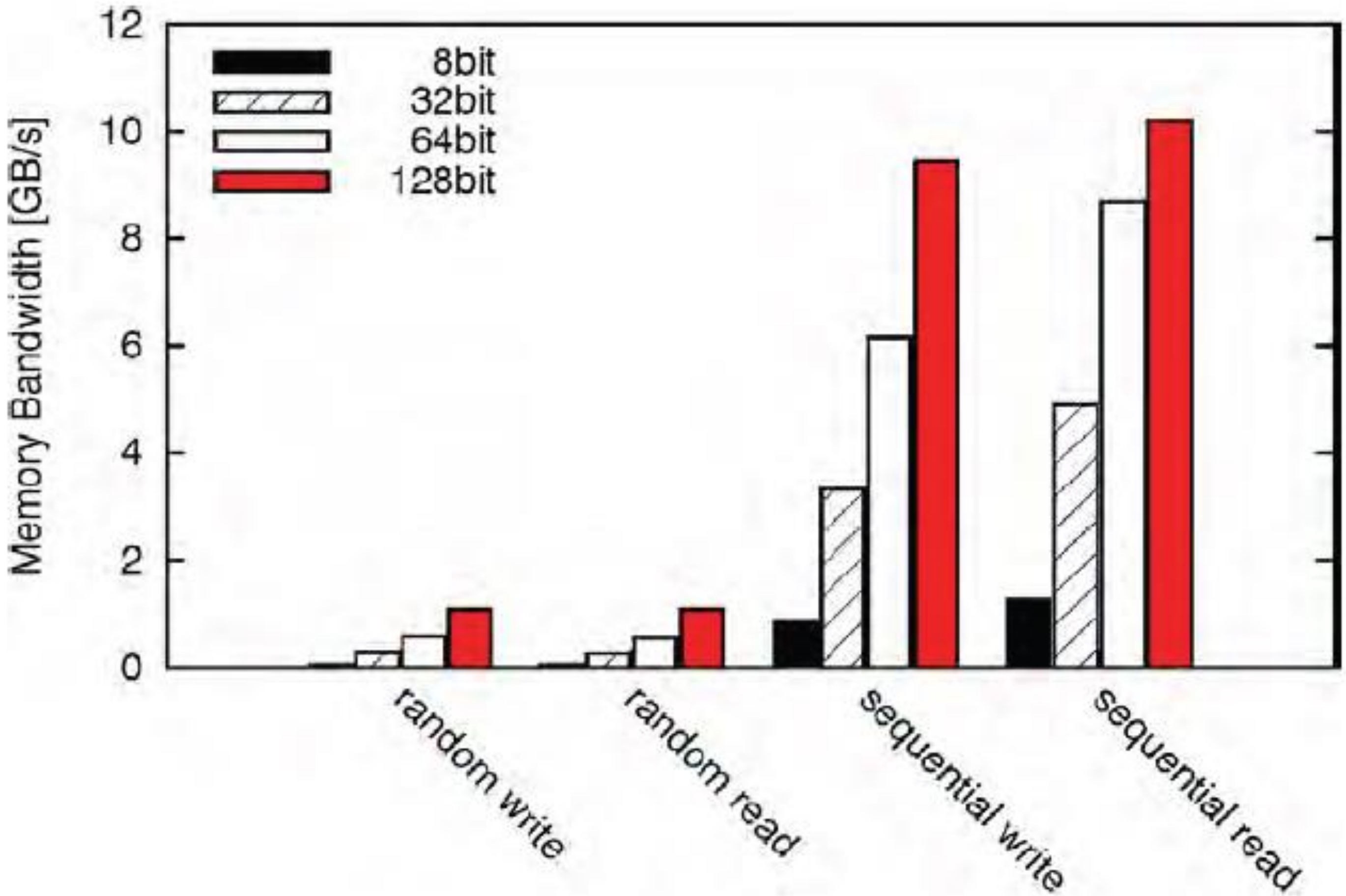




> Speed in Relation...



> Memory Performance Comparison

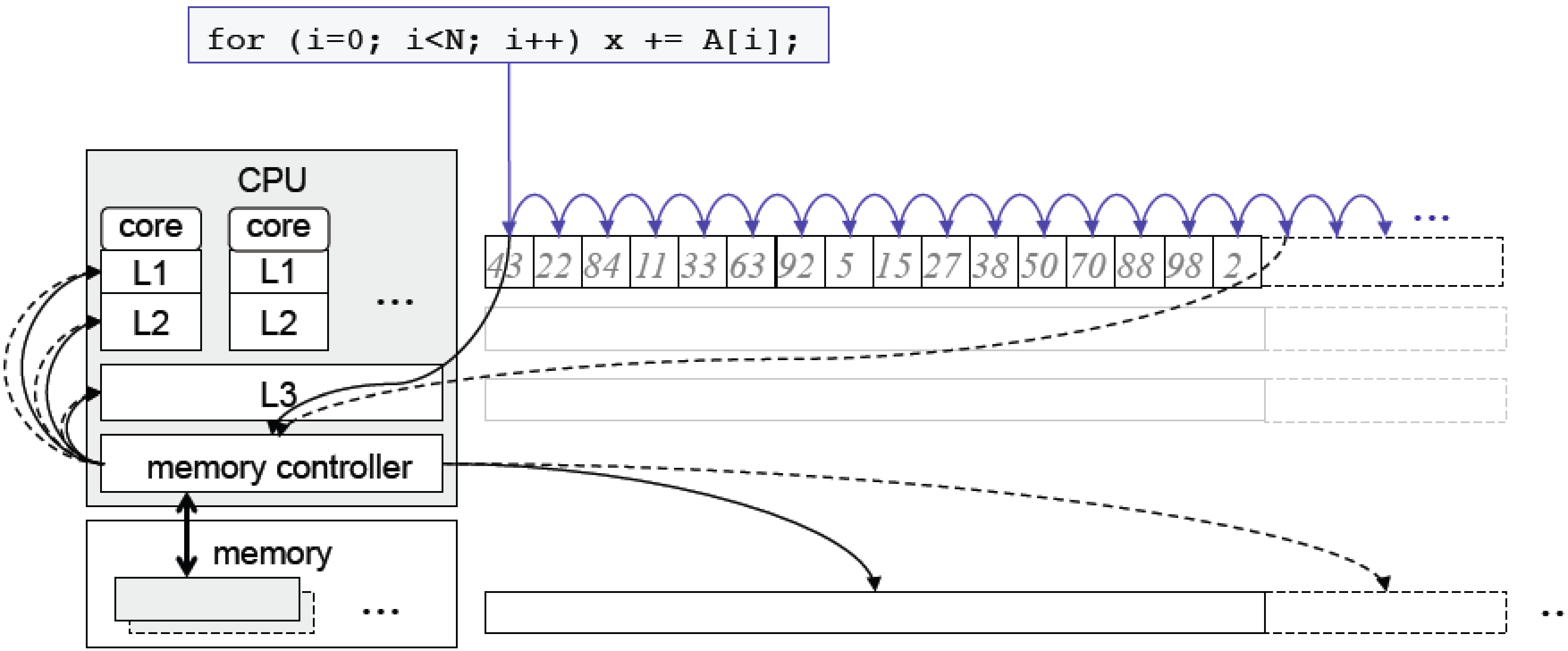


Results for a quad-core i7 2.66GHz, DDR3 1666. 32GB data accessed total.

> The Role of Caches

Caches – the sunny side

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:

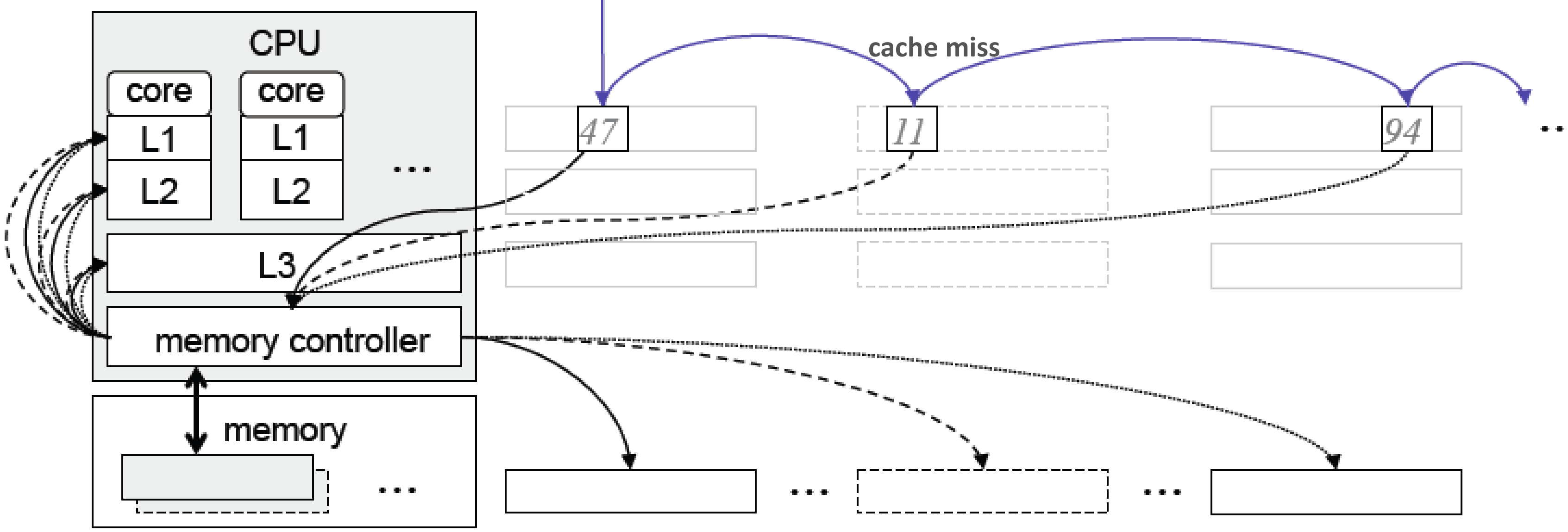


➔ linear memory access maximizes cache & bandwidth utilization

Caches – the bad side

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:

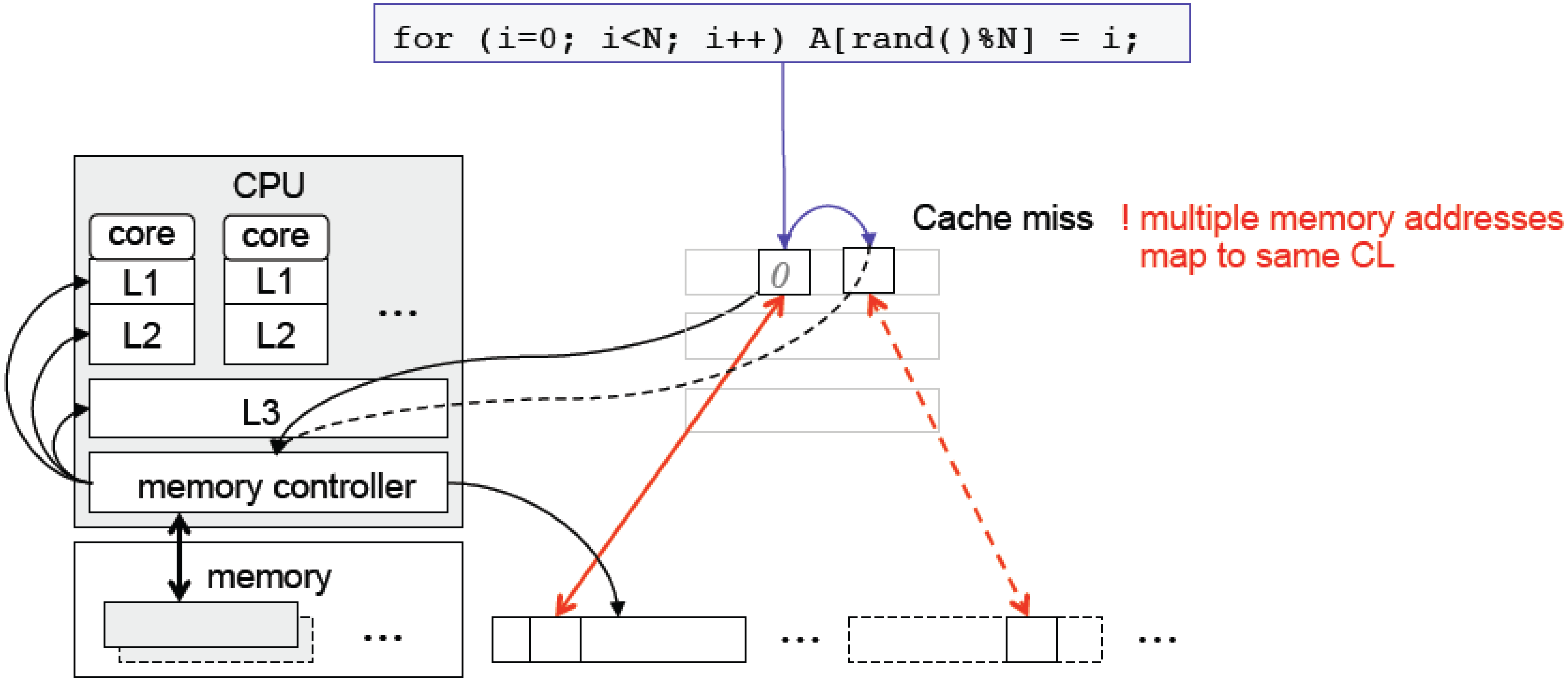
```
for (i=0; i<N; i++) x += A[rand()%N];
```



➔ Random memory access wastes up to 98.5%* of bandwidth

Caches – the Ugly

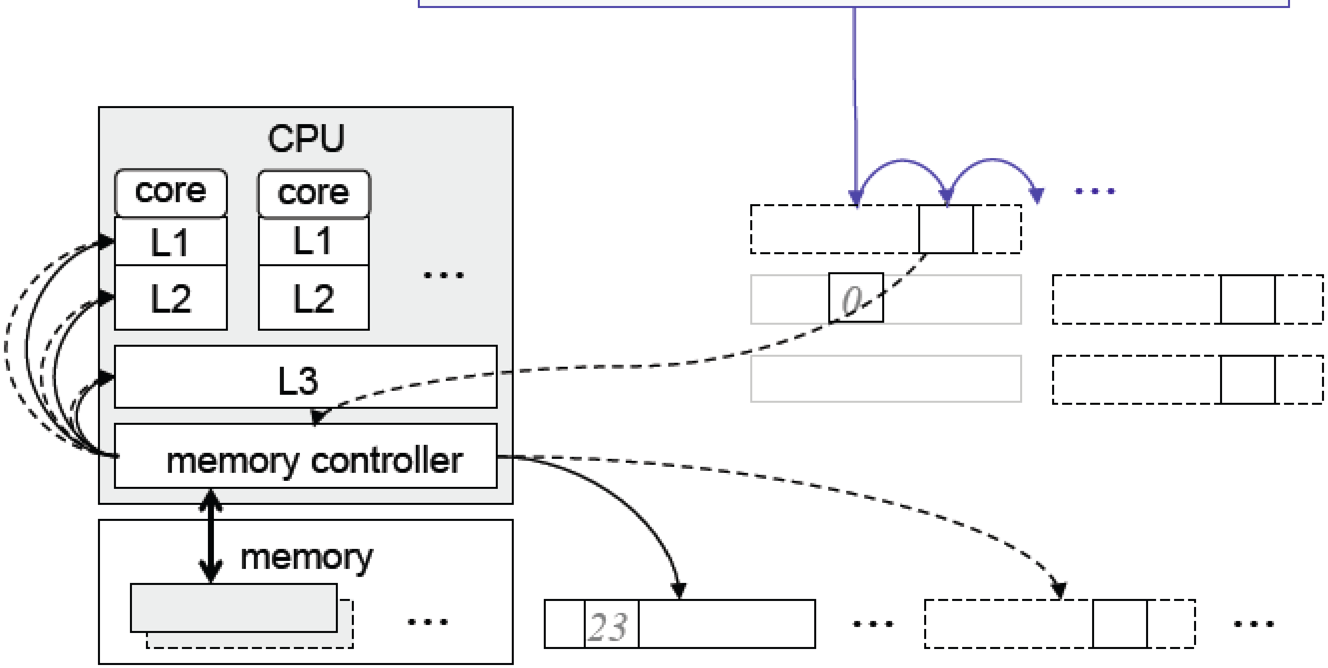
- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes effectively turn into read-modify-write
  - Many memory addresses map into the same cache line(s)
  - “Dirty” cache line needs to be evicted before new one loads



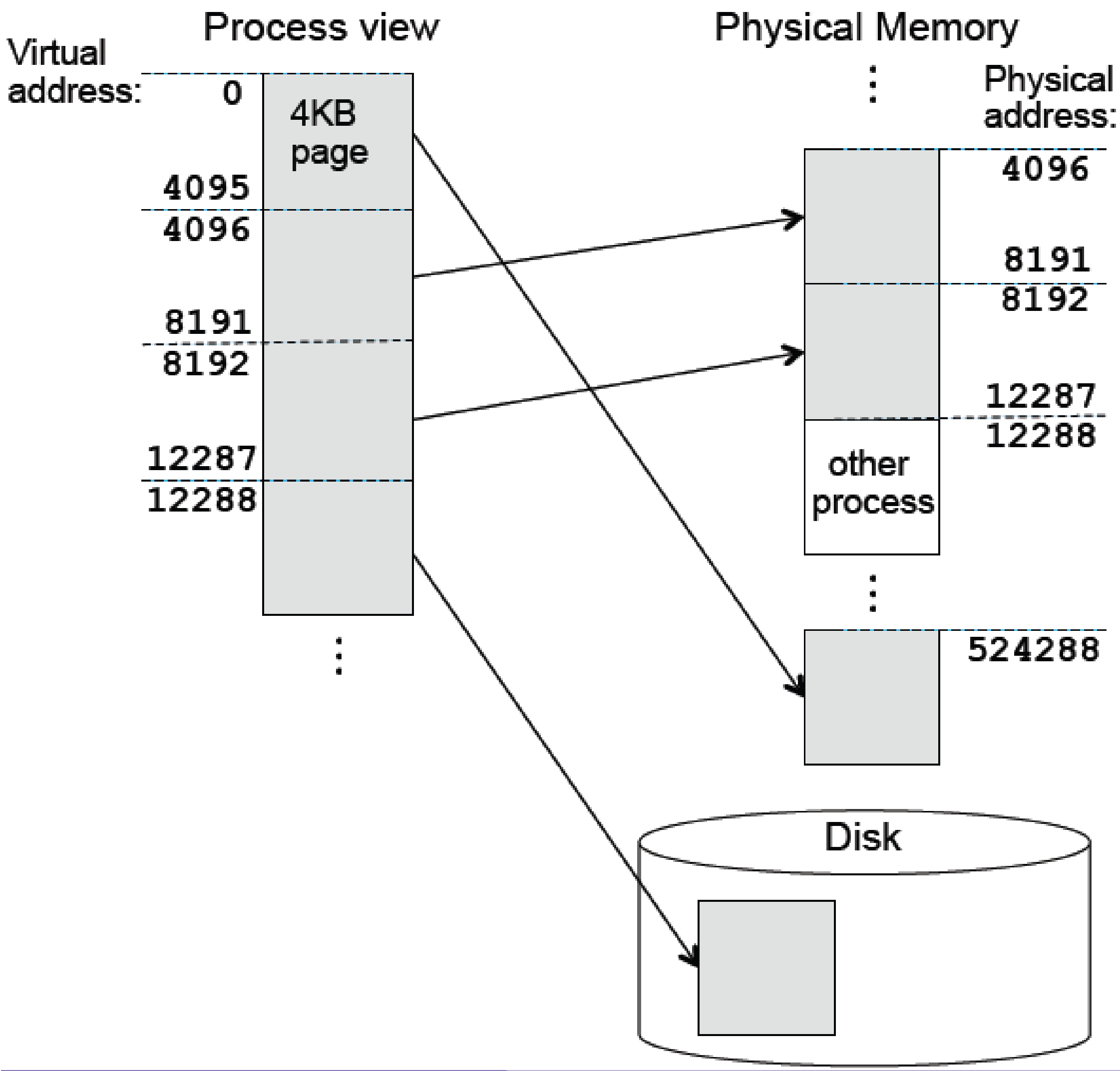
Caches – the Ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes effectively turn into read-modify-write
  - Many memory addresses map into the same cache line(s)
  - “Dirty” cache line needs to be evicted before new one loads

```
for (i=0; i<N; i++) A[rand()%N] = i;
```



> Virtual Memory Management



- Virtual to physical address translation
- Mapping is stored in memory itself
- Memory access now requires 2 round trips to memory
- Caching: Translation Look-aside Buffer (TLB), e.g. Core i7 has 64 entries in L1

➔ TLB misses are costly!

> Is Memory the new Disk?

Is memory the new disk → in terms of behavior?

→ Not quite

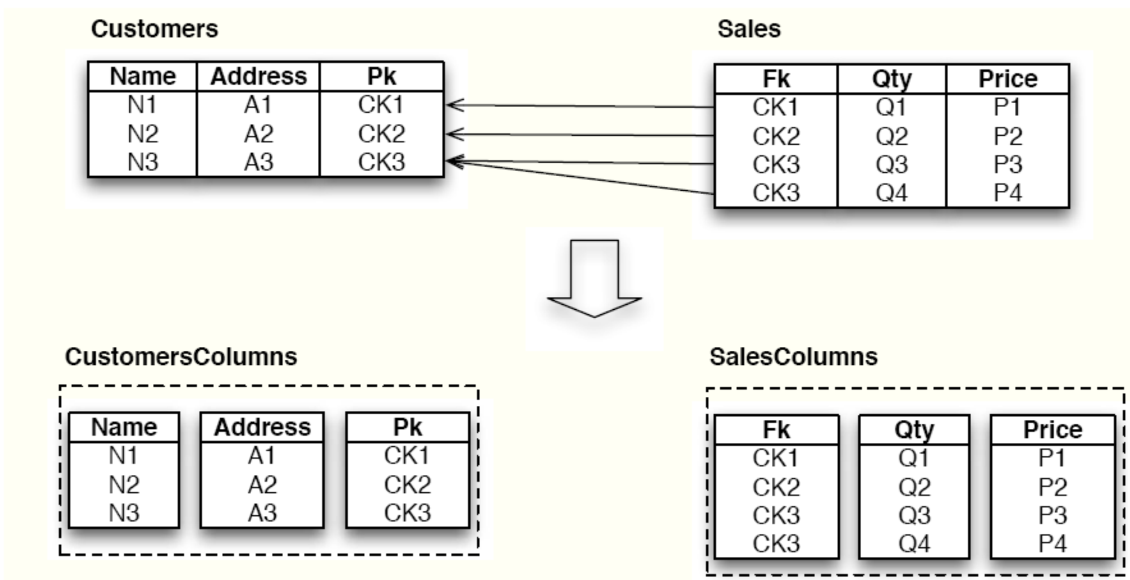
- Some characteristics are very similar, e.g. random vs. sequential
- Memory architecture complicates things !

Aspect		
Rand vs. seq	1-2 orders of magnitude	3 orders of magnitude
Access granuarity	Byte addressable in theory, Caches get in the way	1 disk block, usually 4KB
Writes	Read-modify write (CL)	Read-modify-write (block)
Concurrency	Parallel memory access for peak performance	Multiple seq. streams ➔ random access

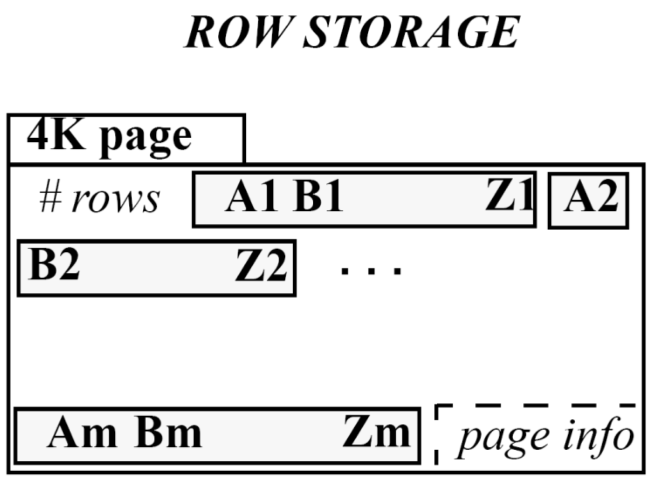
Tables are stored by columns rather than by rows

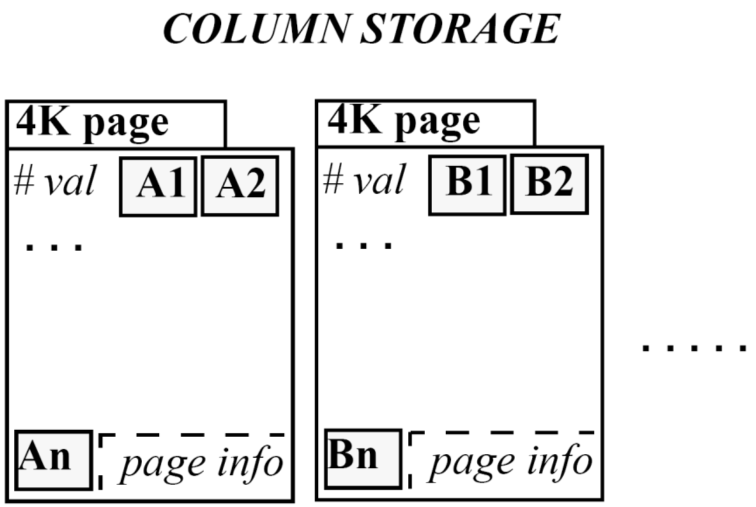
Columnar techniques provide benefits for many application areas (OLAP)

- Data warehousing, BI
- Information retrieval, graphs, e-science



> Row-Storage vs Column-Storage





+ easy to add/modify a record

- might read unnecessary data

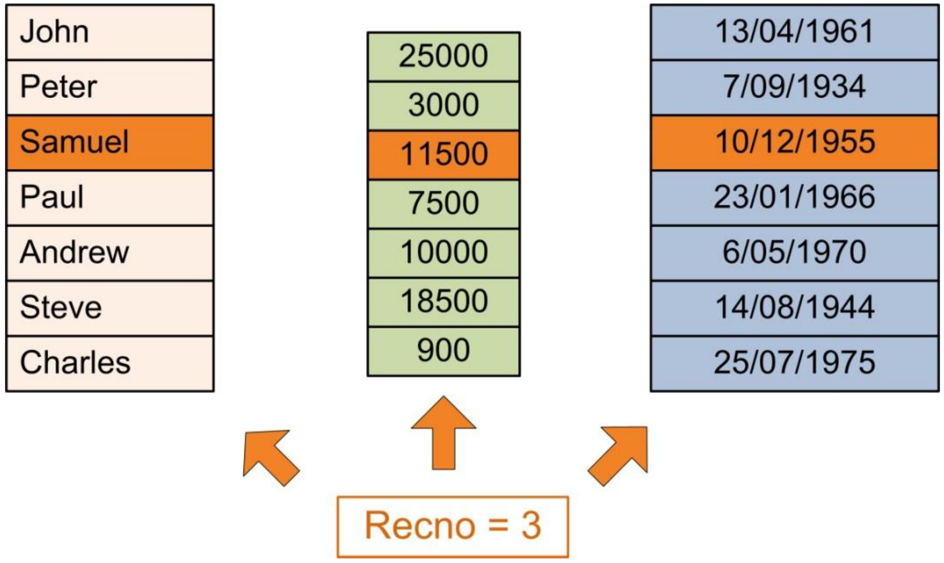
+ only need to read in relevant data

- tuple writes require multiple accesses

-> suitable for read-mostly, read-intensive, large data repositories

> Positional Addressing

- Addressing traditionally based on the physical position of the element within the database (e.g. TID concept)
- Positional addressing based on the record position within the table
  - Columnar organization leads to simplified physical structures
  - Fields are stored fixed length
- Implicit relationship across column files - Header/ID elimination of columns
- All addresses are simply represented by integer numbers (efficient to store and process)



> Column-Store Operators

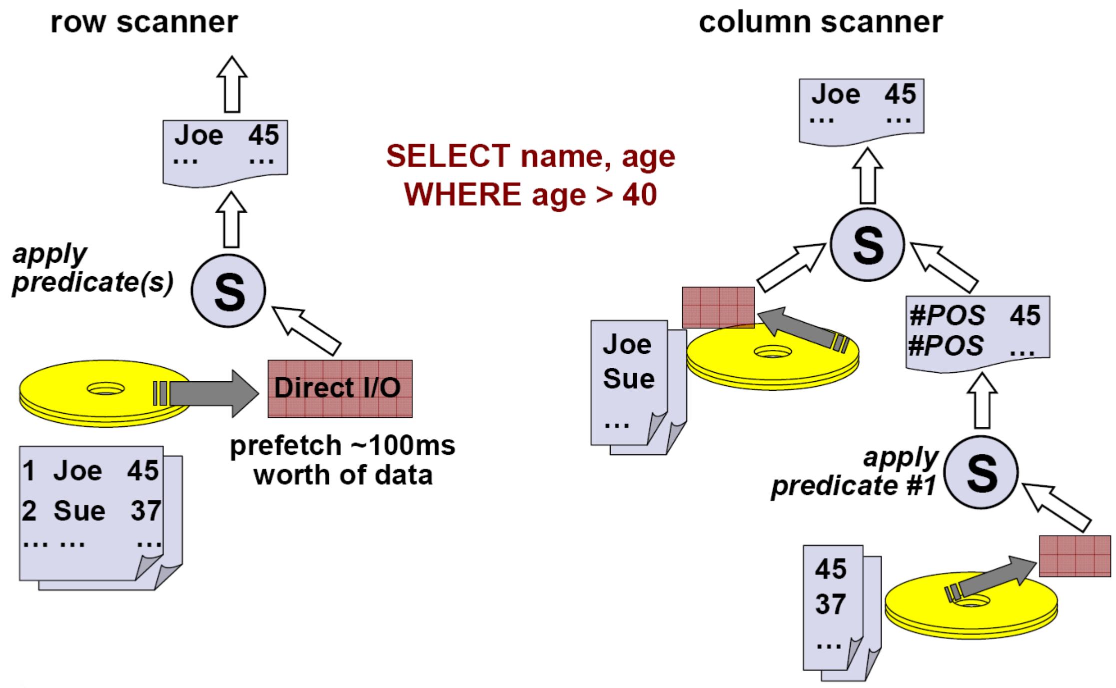
Column scan operators

- 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



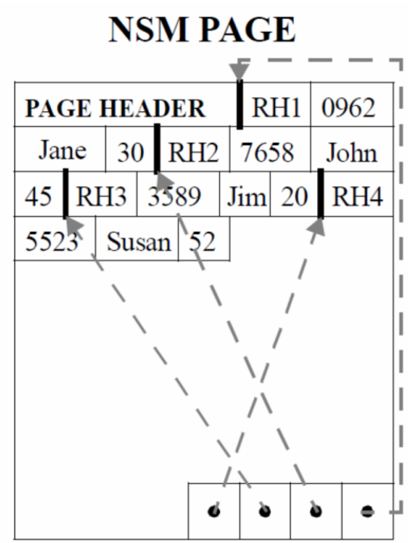
- Reads from a single file
- Iterates over pages, for each page iterates over tuples and applies predicates

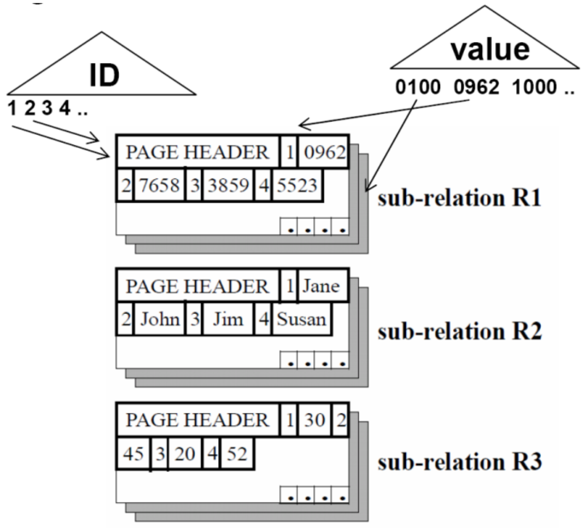
- Must read as many files as are columns specified in the query
- Series of pipelined scan nodes
- Applies predicates by reading a column, creating {position, value} pairs for all qualified tuples
- Attaches values corresponding to input positions from other columns

Column-Store Architecture – History and Example

1985: DSM (Decomposition Storage Model)

- Proposed as an alternative to NSM (Normalized Storage Model)
- Decomposition storage mode, decomposes relations vertically
- 2 indexes: clustered on ID, non-clustered on value
- Speeds up queries projecting few columns
- Disadvantages: storage overhead for storing tuple IDs, expensive tuple reconstruction costs

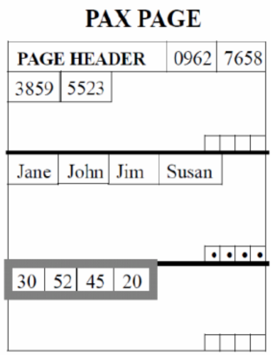




> From DSM to Column-Stores

Late 90s – 2000s: Focus on main-memory performance

- MonetDB
- PAX: Partition Attributes Across
  - Retains NSM I/O pattern
  - Optimizes cache-to-RAM communication



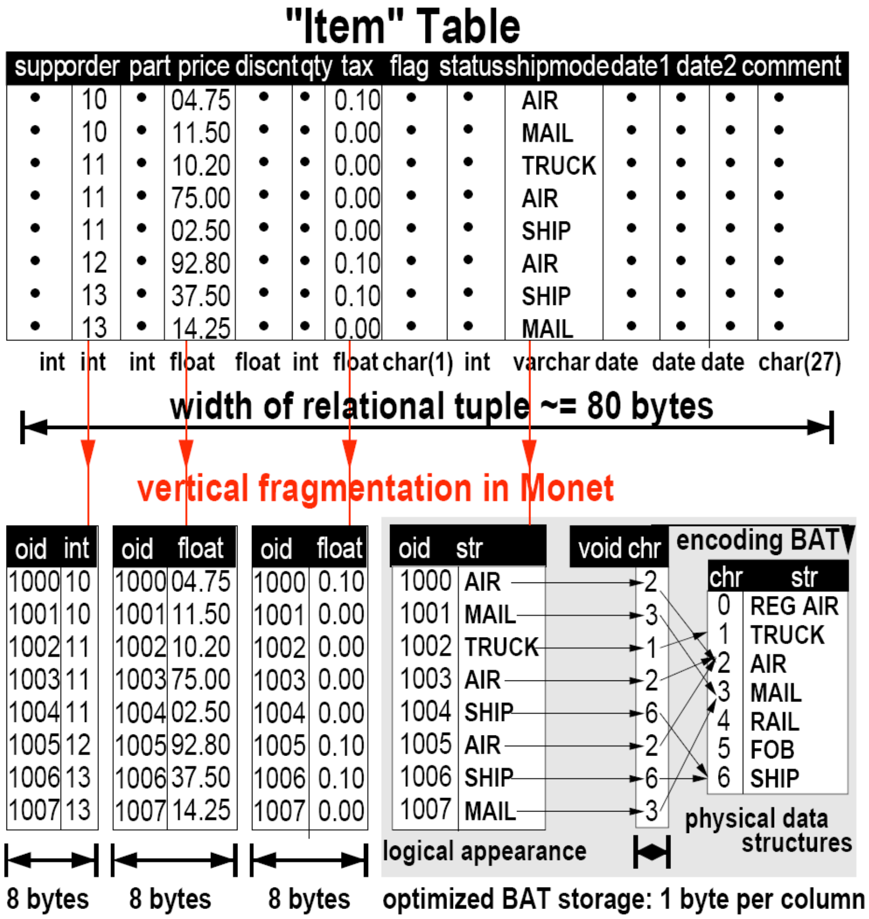
2005: the (re)birth of column-stores

- New hardware and application realities
  - Faster CPUs, larger memories, disk bandwidth
  - Multi-terabyte Data Warehouses
- New approach: combine several techniques
  - Read-optimized, fast multi-column access, disk/CPU efficiency, light-weight compression
- Used in read oriented environments - OLAP

Some column store systems

- MonetDB, C-Store, Sybase IQ, SAP Business Warehouse Accelerator, Infobright, Exasol, X100/VectorWise

- Each column is stored in a separate binary table (BAT – Binary Association Table)
  - Array of fixed-size two-field records, e.g. [OID, value])
- Two space optimizations that further reduce the memory requirements in BAT
  - Virtual-OIDs: use identical system-generated column of OIDs and compute the OID values on-the-fly
  - Byte-encoding: use fixed-size encoding in 1- or 2-byte integer value
- All (intermediate) results of a query are stored in BATs

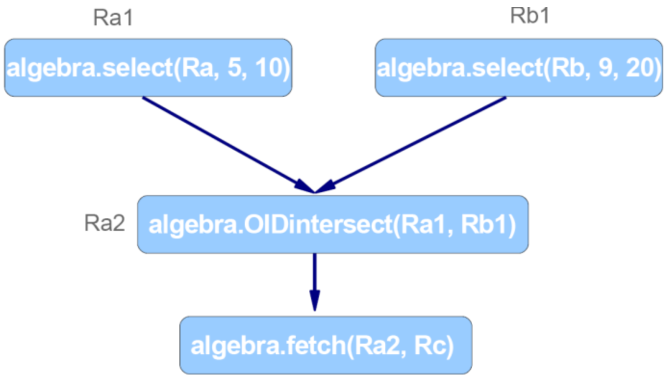


MonetDB

- SQL queries are translated into a query template
  - MonetDB Interpreter Language (MIL)
- Bind operations are used to localize persistent BATSs for tables
- Template is processed by a chain of optimizers
  - Simple constant expression evaluation
  - Preparation for multi-core parallel processing
  - Garbage collection
- Every relational operator takes one or more columns and produces a new set of columns

select R.c from R where $5 \leq$ R.a ≤ 10 and $9 \leq$ R.b ≤ 20

```
Ra1 := algebra.select(Ra, 5, 10);
Rb1 := algebra.select(Rb, 9, 20);
Ra2 := algebra.OIDintersect(Ra1, Rb1);
Rc1 := algebra.fetch(Rc, Ra2);
```



Heavy use of code expansion to reduce cost
55 selection routines
149 unary operations
335 join/group operations
134 multi-join operations
72 aggregate operations

C-Store

- Data in C-Store is not physically stored using the logical data model
- C-Store implements only **projections**
  - Some number of columns form a fact table
  - Plus columns in a dimension table – with a 1-n join between fact and dimension table
- Tuples in a projection are sorted on the same sort key which can be any column in the projection
- Every projection is horizontally partitioned into 1 or more segments
  - value-based partitioning on the sort key
- To reconstruct complete rows C-Store needs to join segments from different projections
  - Each segment associates every data value of every column with a storage key
  - Join index: contains the segment ID and storage key of the corresponding (joining) tuple

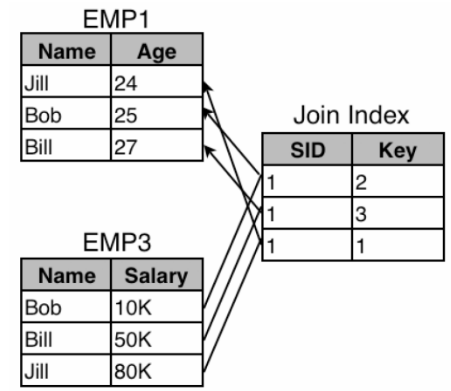
User view

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K

Possible set of MVs

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

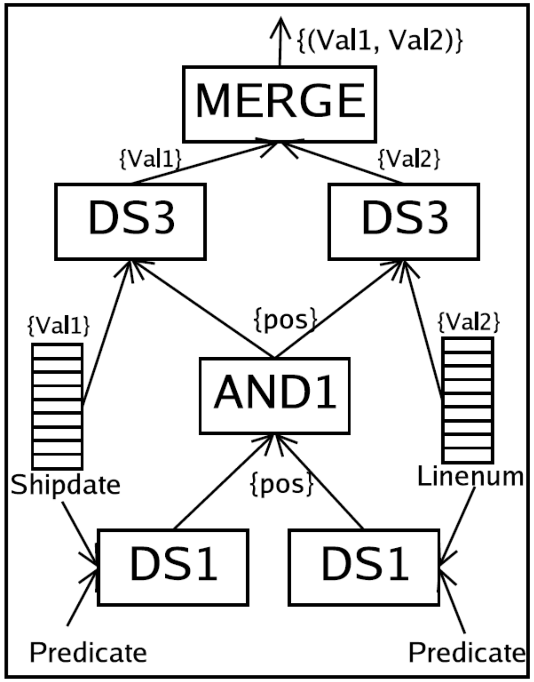
Join Index



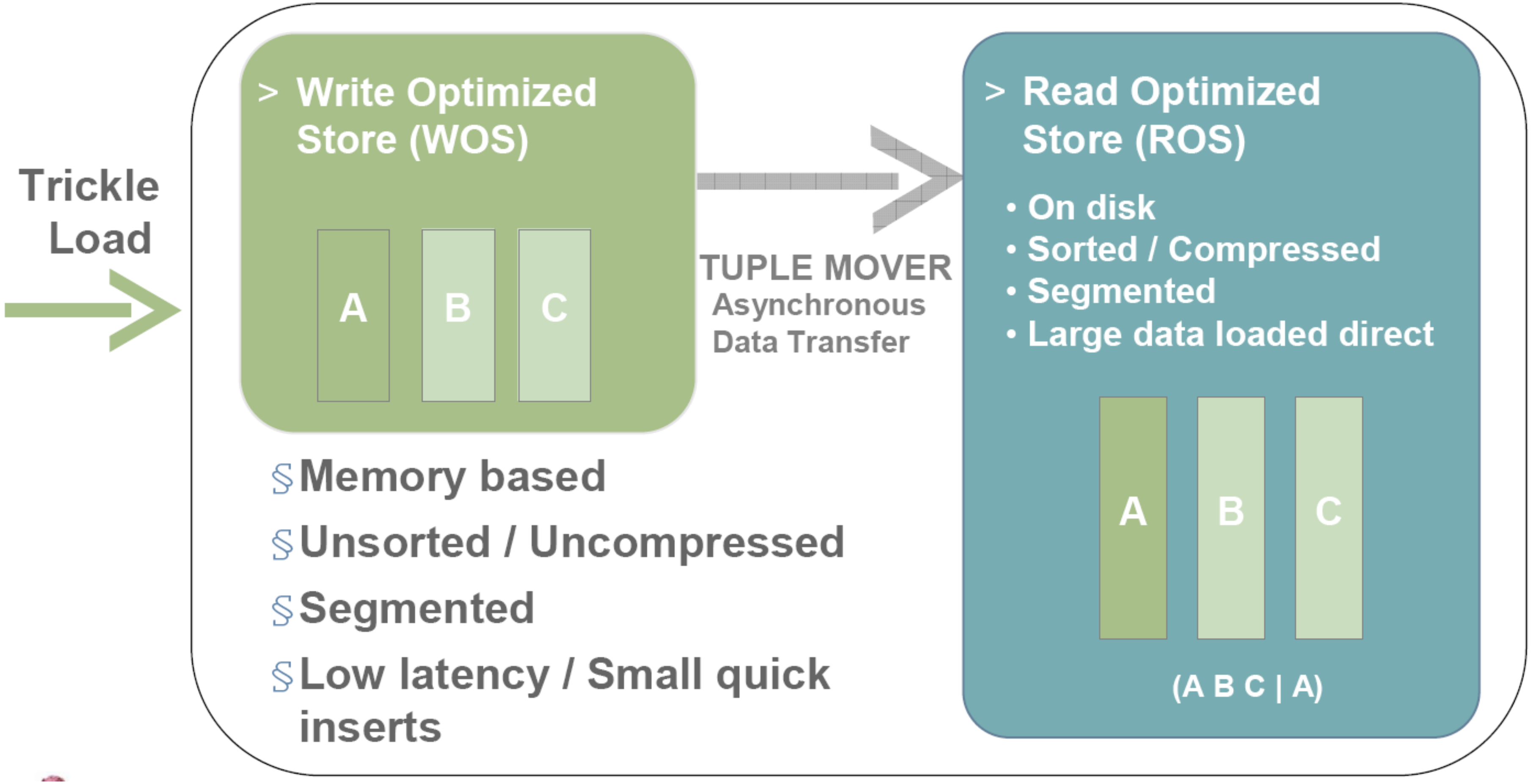
> C-Store

- Blocks of data are passed between operators (getNextBlock())
- In contrast to row-stores, operators in query plans do not form trees
- DS (Data Source) Operators
  - Responsible for reading columns of disk and filtering on one or more single-column predicates
  - Output: Vectors of positions or vectors of positions and values
  - LM (DS1, DS3) and EM (DS2, DS4)
- AND Operator
  - Merge several position lists into a single position list
- MERGE Operator
  - Tuple construction: combines multiple narrow tuples of positions and values into wider tuples

```
SELECT shipdate, linenum FROM lineitem
WHERE  shipdate < CONST1 AND linenum < CONST2
```

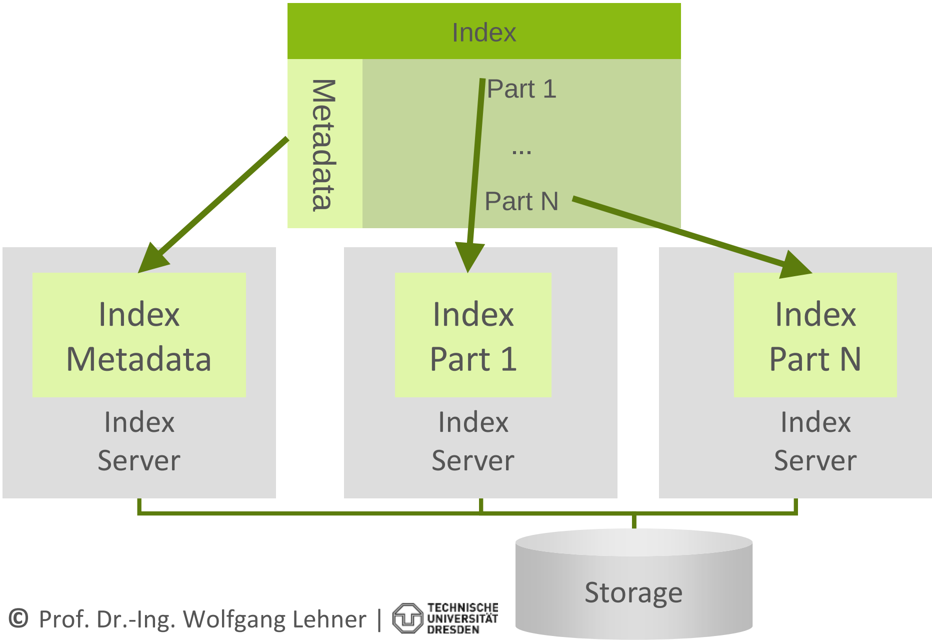


Hybrid Storage Architecture



> SAP Business Warehouse Accelerator

- Distributed In-Memory Column-Store
- Focus on Performance, Scalability and High-Availability
- Uses Dictionary Compression, and Bitlength encoding
- Strong compression of sparse attributes
- Horizontal and Vertical Data partitioning



DocId 1	JAN	INTEL	RED	€1
DocId 2	FEB	ABB	GREEN	€2
DocId 3	MAR	HP	BLUE	€2
DocId 4	APR	INTEL	RED	€4
DocId 5	MAY	IBM	WHITE	€5
DocId 6	JUN	IBM	BLACK	€5
DocId 7	JUL	SIEMENS	BROWN	€4
DocId 8	AUG	INTEL	BLUE	€3
...	...	...	...	...

Dictionary

ValueId	Value
1	ABB
2	HP
3	IBM
4	INTEL
5	SIEMENS

Attribute Table

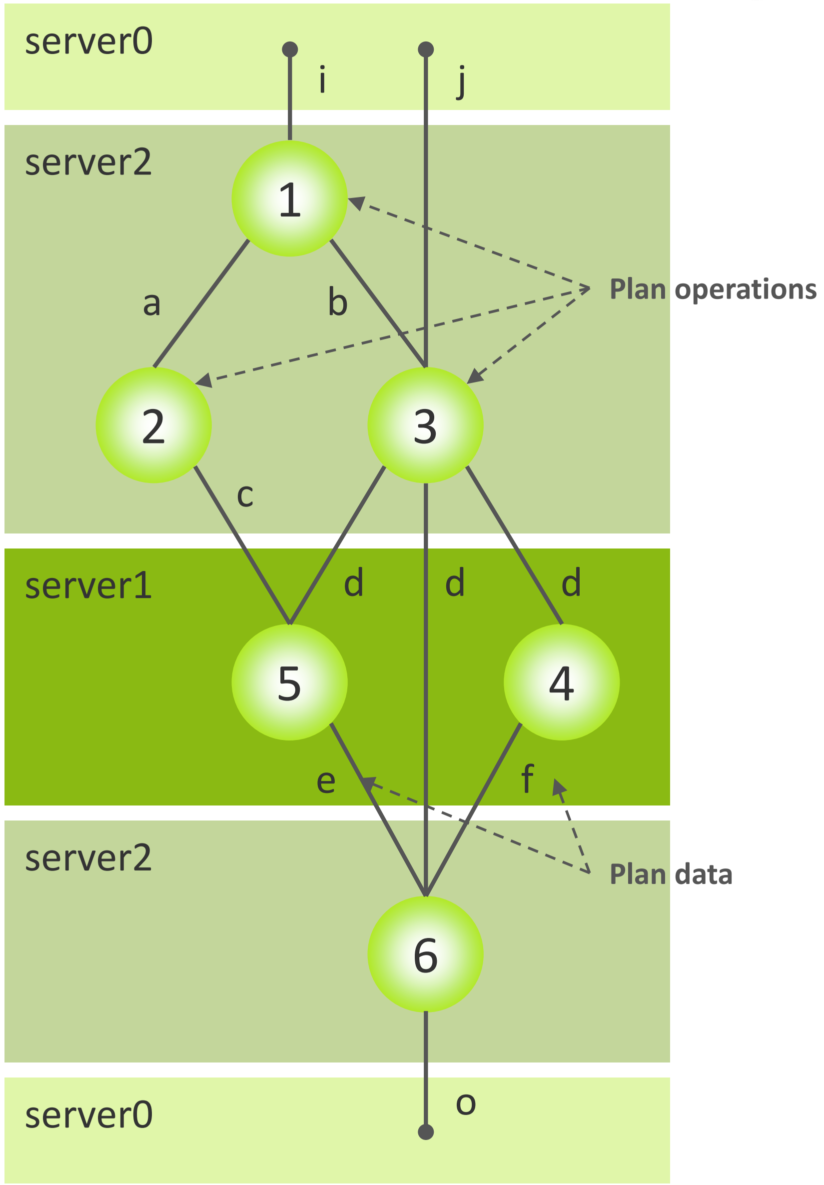
DocId	ValueId
1	4
2	1
3	2
4	4
5	3
6	3
7	5
8	4

Index

ValueId	DocIdList
1	2
2	3
3	5,6
4	1,4,8
5	7

> SAP Business Warehouse Accelerator

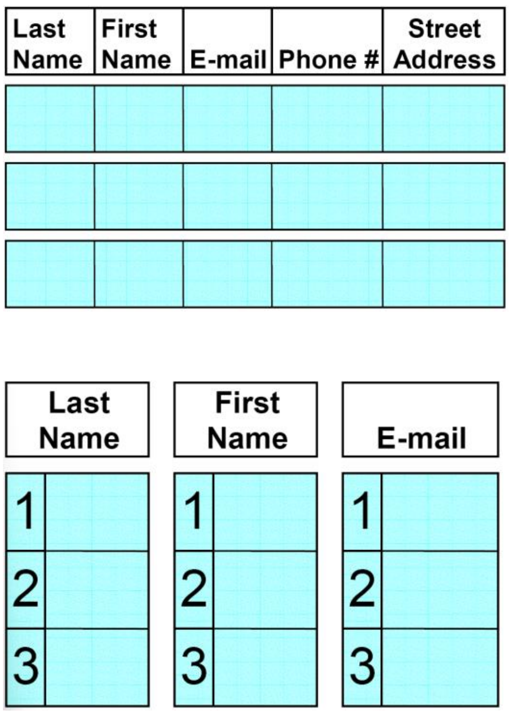
- Optimized for latest Hardware
  - Cache optimized
  - Use of SIMD instructions
- Plan executor implements a distributed execution framework
- Query plan is distributed over the landscape and executed in parallel
- Parallel aggregation on each data partition
- Small set of efficient plan operations
  - Search (Filter the predicates)
  - Join (join filterd rowIDs with F-table)
  - Aggregate (measures in the F-table)
  - Merge (the distributed aggregation results)
  - BuildResult (Build final result table)
- Near constant query execution time over arbitrary large data volumes



Simulate a Column-Store inside a Row-Store?

> Vertical Partioning (Row-Store)

- Technique to enhance performance on read-mostly data warehouse workloads
- Store an ***n***-column table in ***n*** new tables
- Each new table contains two columns - a tuple ID column and data value column
- Each table is clustered by tuple ID -> Joins are not expensive

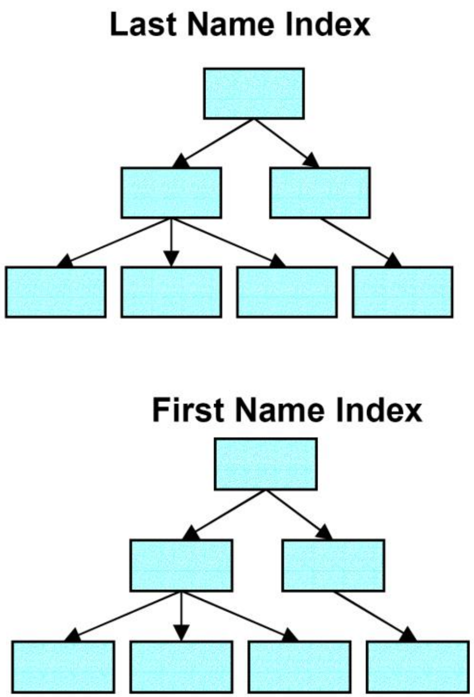


Problems

- Data sizes: tuple ID and tuple header for each row necessary
- Loss of row-store performance optimizations like horizontal partitioning
- Large number of partition joins
- No column-oriented compression or optimizations for fixed-width attributes

Index Every Column (Row-Store)

- Indexes reduce I/O costs by avoiding the need to perform table scans since they directly contain the data or pointers to the data
- Column-store only reads relevant columns and avoids a table scan
- Idea: Index every column in a row-store



Problems

- Result of lower part of query plan is a set of TIDs that passed all predicates
- Need to extract SELECT attributes at these TIDs
  - BUT: index maps value to TID
  - You really want to map TID to value
  - -> Tuple construction is slow
- Column-store is very different from an index

Physical Organization - Compression

> Compression Advantages

Compression improves I/O performance

- Reduces seek times, transfer times and increases buffer hit rate
- CPU overhead of decompression is often compensated for by the I/O improvements

Columns compress better than rows

- Rows contain values from different domains
- Consecutive entries in a column are quite similar
- Techniques such as run length encoding far more useful

Column-stores can store different columns in different sort-orders

- Sorted data is usually quite compressible

Operating Directly on Compressed Data

- IO – CPU tradeoff is no longer a tradeoff
- Reduces memory-CPU bandwith requirements
- Opens up possibility of operating on multiple records at once

Run-Length Encoding

Quarter	Product ID	Price
Q1	1	5
Q1	1	7
Q1	1	2
Q1	1	9
Q1	1	6
Q1	2	8
Q1	2	5
...	...	...
Q2	1	3
Q2	1	8
Q2	1	1
Q2	2	4
...	...	...

→

Quarter

(value, start_pos, run_length)

(Q1, 1, 300)
(Q2, 301, 350)
(Q3, 651, 500)
(Q4, 1151, 600)

Product ID

(value, start_pos, run_length)

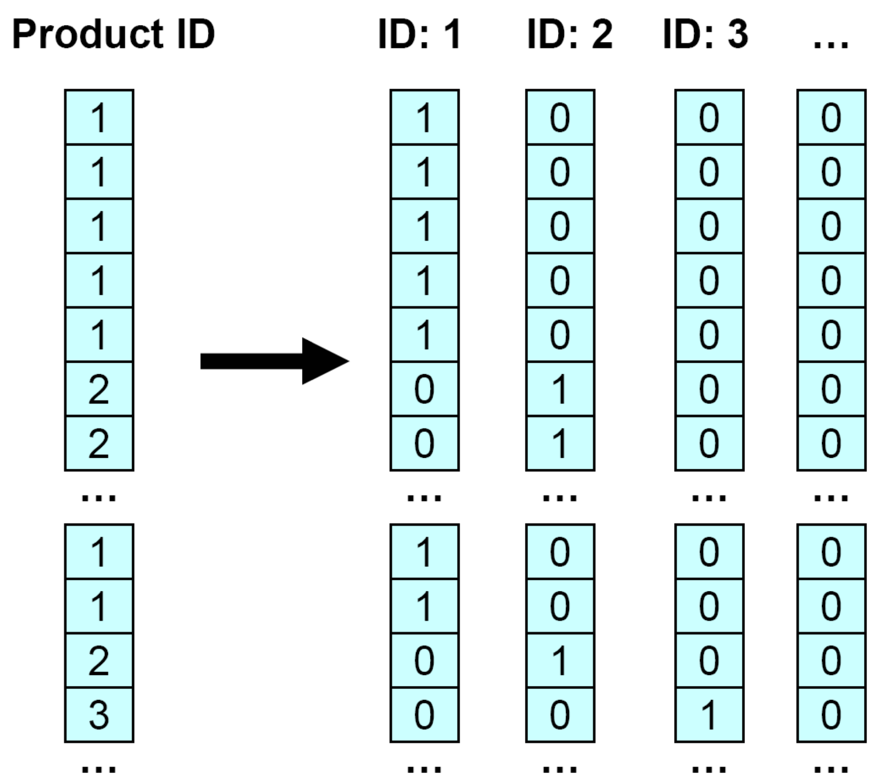
(1, 1, 5)
(2, 6, 2)
...
(1, 301, 3)
(2, 304, 1)
...

Price

5
7
2
9
6
8
5
...
3
8
1
4
...

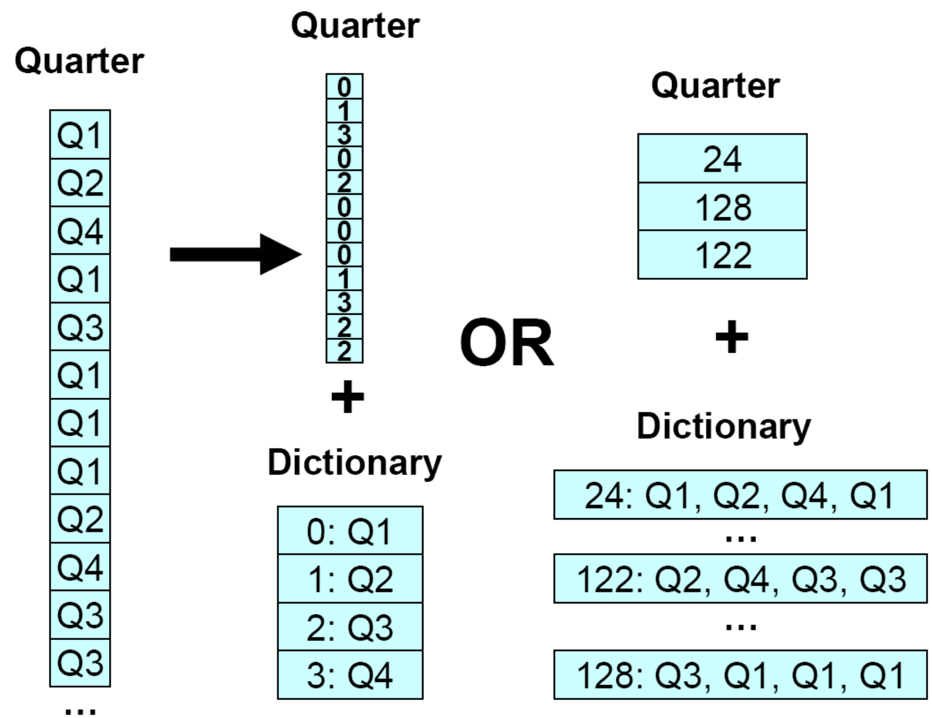
> Bit-Vector Encoding

- For each unique value v in column c, create bit-vector b
- b[i] = 1 if c[i] = v
- Good for columns with few unique values
- Each bit-vector can be further compressed if sparse



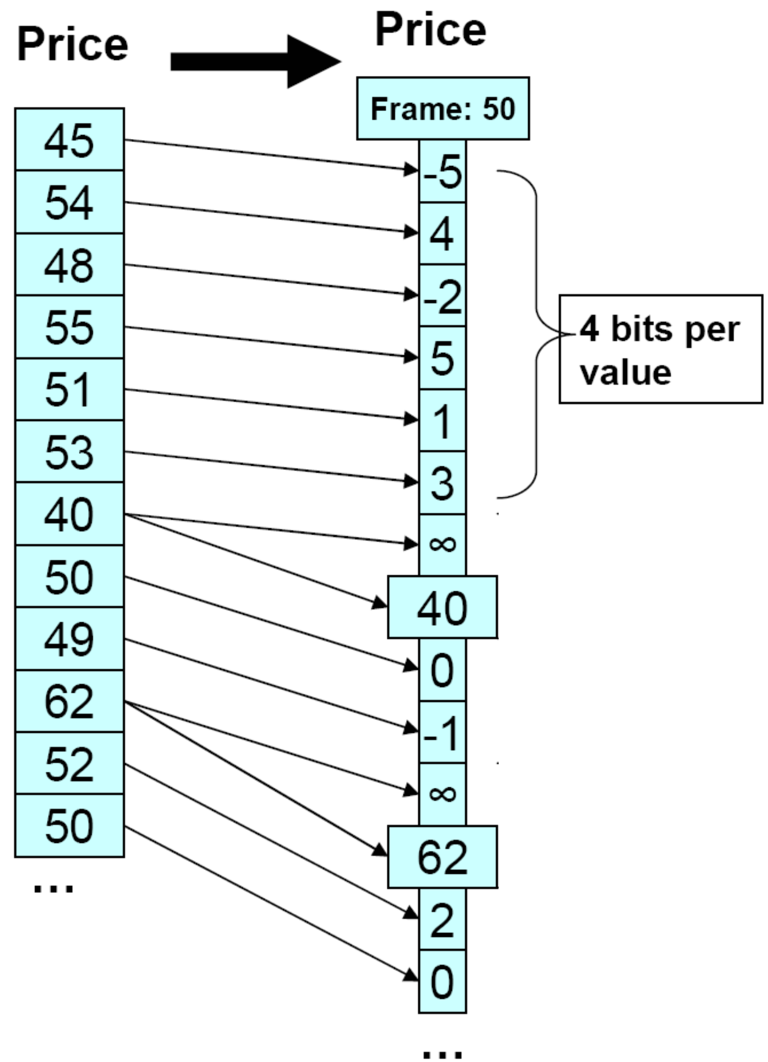
Dictionary Encoding

- For each unique value create dictionary entry
- Dictionary can be per-block or per-column
- Column-stores have the advantage that dictionary entries may encode multiple values at once



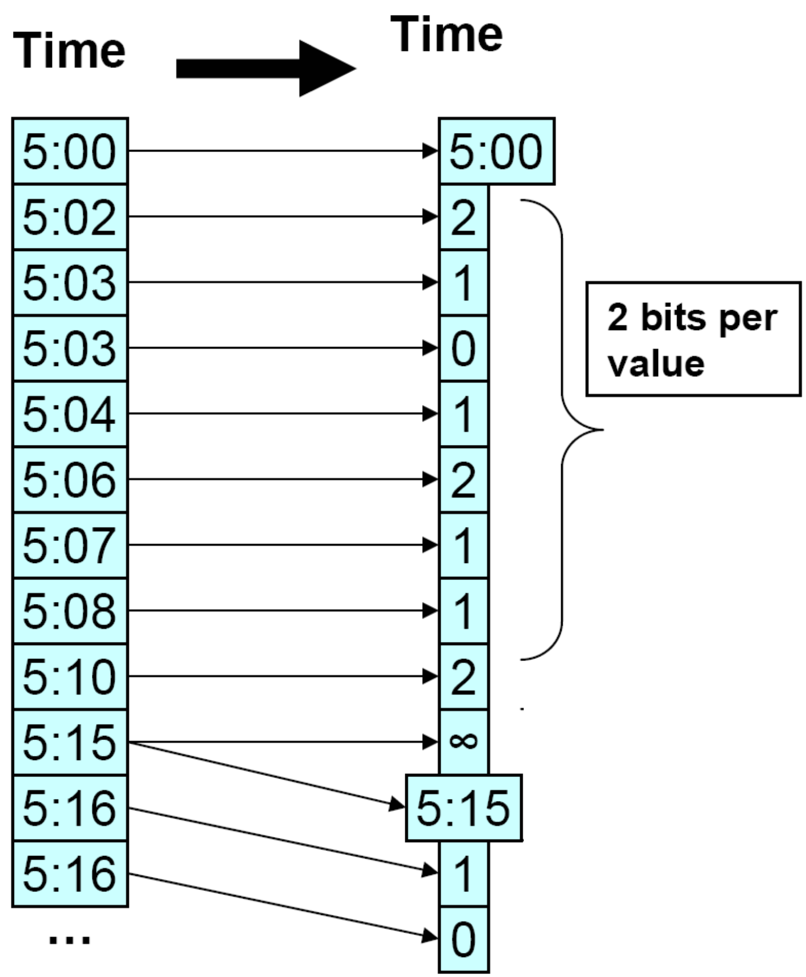
> Frame Of Reference Encoding

- Encodes values as b bit offset from chosen frame of reference
- Special escape code (e.g. all bits set to 1) indicates a difference larger than can be stored in b bits
- After escape code, original (uncompressed) value is written

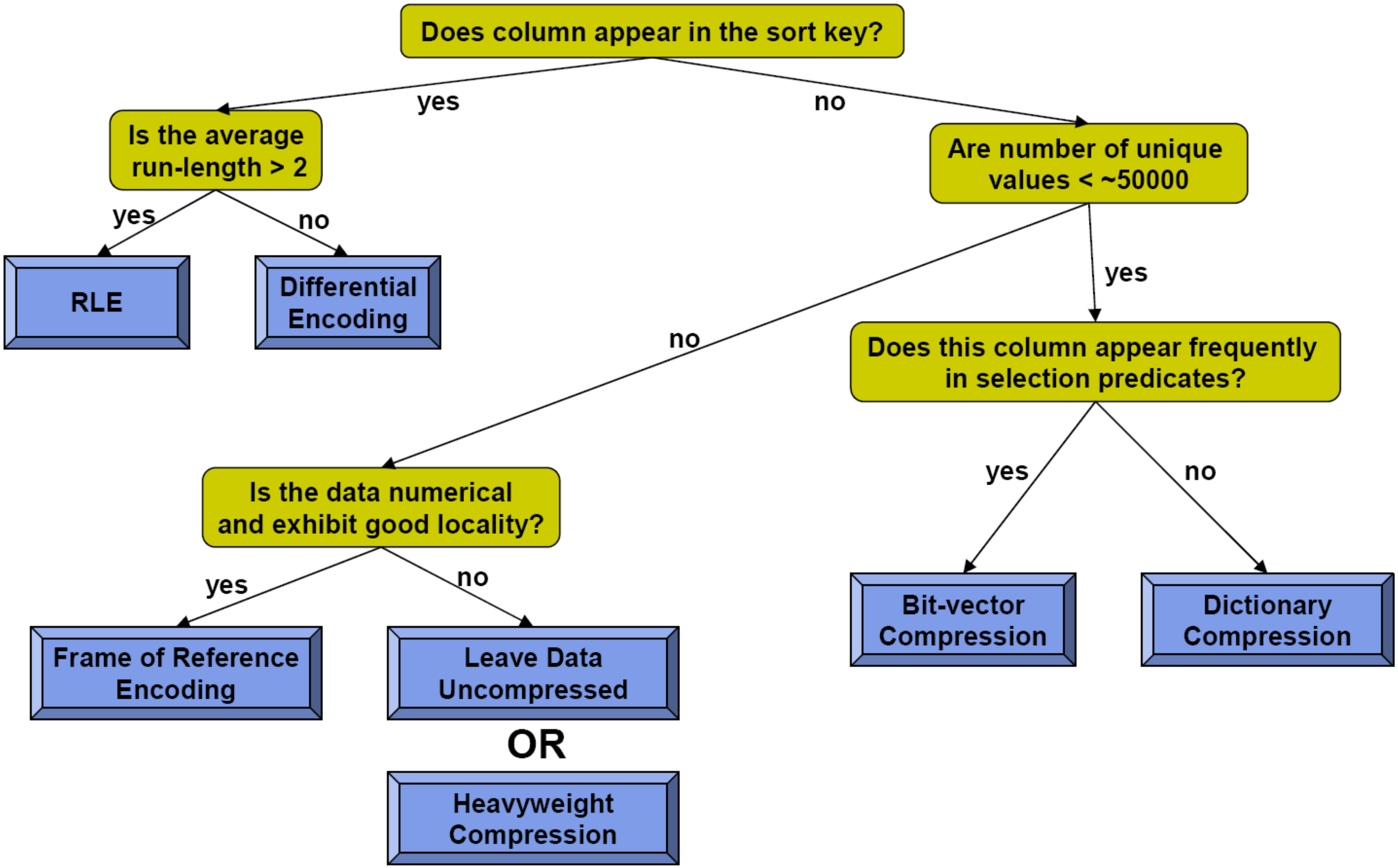


> Differential Encoding

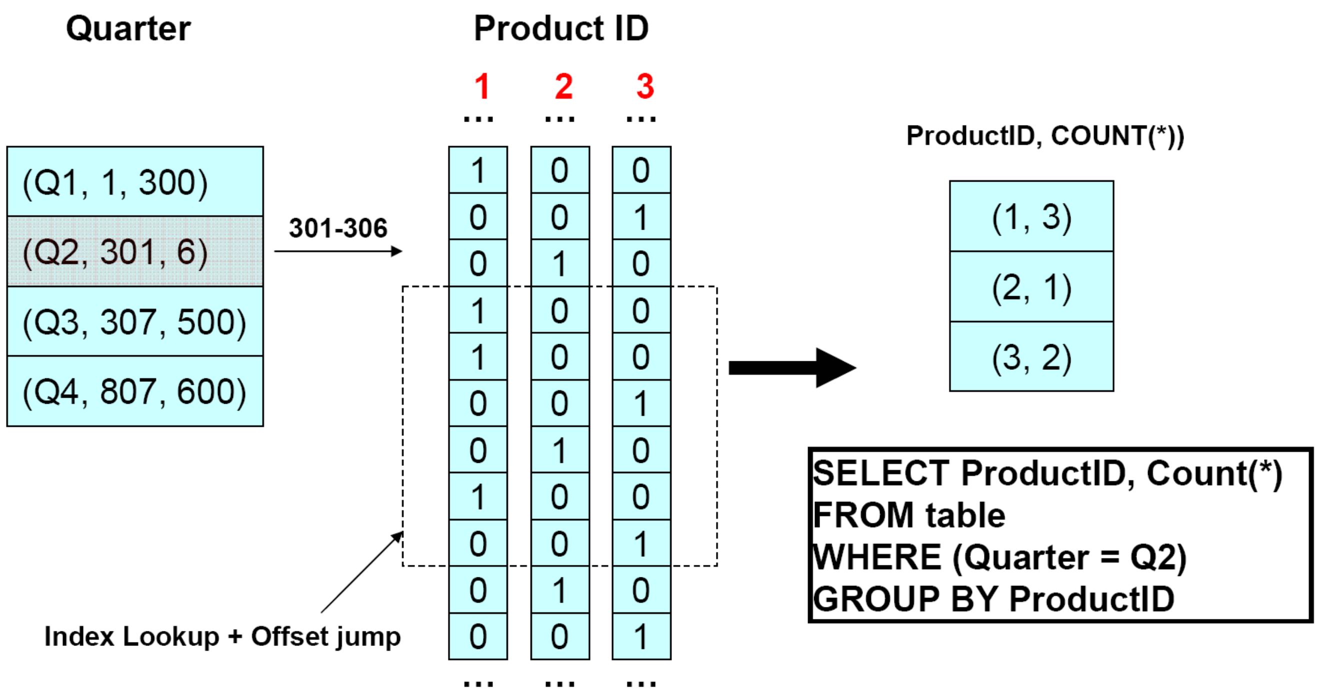
- Encodes values as b bit offset from previous value
- Special escape code (just like frame of reference encoding) indicates a difference larger than can be stored in b bits
- After escape code, original (uncompressed) value is written
- Unlike Frame of Reference Encoding the encoded column has to be sorted and only positive offsets are used
- Performs well on columns containing increasing/decreasing sequences
  - inverted lists
  - Timestamps
  - object Ids
  - sorted / clustered columns



> What Compression Scheme to Use?



> Operating Directly on Compressed Data



Example

- *Quarter* is compressed with Run Length Encoding
  - *(Quarter = Q2)* can be directly retrieved in compressed format
- *Product ID* is compressed with Bit Vector Encoding
  - Index Lookup with position information from *(Quarter = Q2)*
  - *Count(*)* represents the number of bits set in the corresponding bit vector

- Appropriate handling needed for each combination of compression types
- **C-Store** uses a generic API for a compression block
  - Compression block contains a buffer of the column data in compressed format
  - API is provided that allows accessing the buffer in several ways:

Properties	Iterator Access	Block Information
isOneValue()	getNext()	getSize()
isValueSorted()	asArray()	getStartValue()
isPosContig()		getEndPosition()

- **Iterator Access:** used when decompression cannot be avoided
  - *getNext():* Returns next decompressed {value, position} pair
  - *asArray():* Decompresses the entire buffer
- **Block Informations:** returns compressed data
  - Example Run Length Encoding:
    - Block consists of a single RLE triple of the form *(value; start pos; run length)*
    - *getSize()* returns run length, *getStartValue()* returns value, *getEndPosition()* returns *(start pos + run length – 1)*
- **Properties:** Used for operating on compressed columns (e.g. Join)
  - *isOneValue():* returns whether or not the block contains just one value (many positions for that value)
  - *isValueSorted():* returns whether or not the block's values are sorted
  - *isPosContig():* returns whether the block contains a consecutive subset of a column

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)

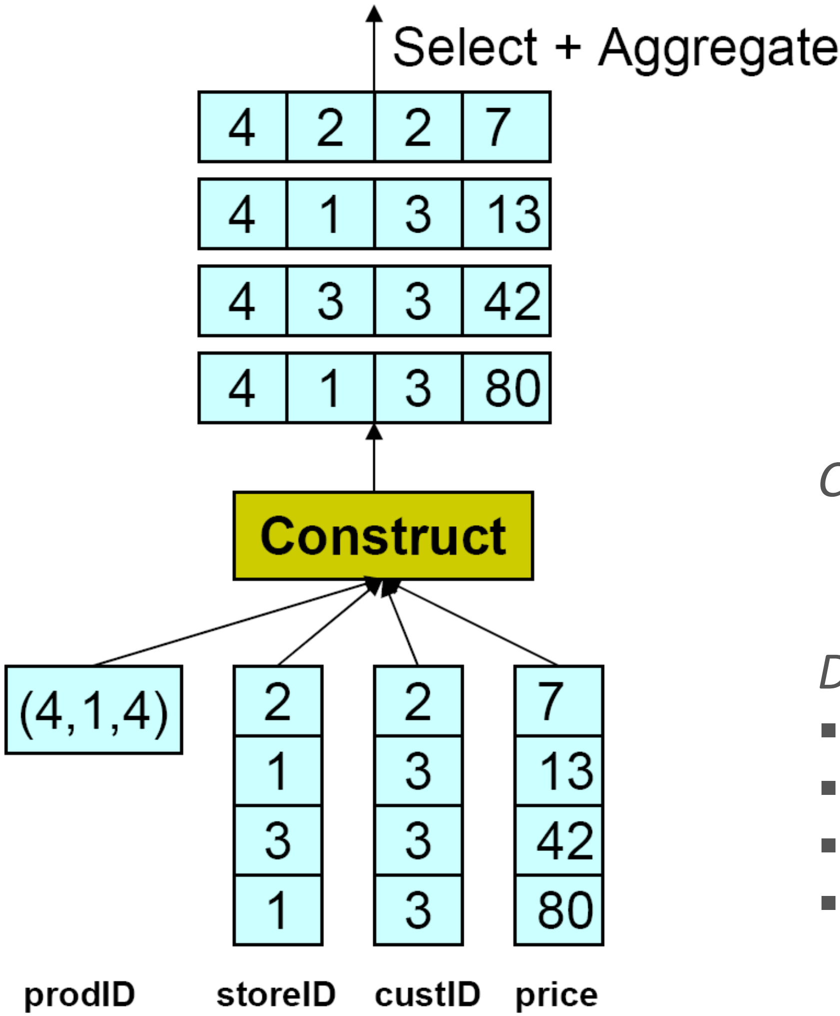
Early materialization

- 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:

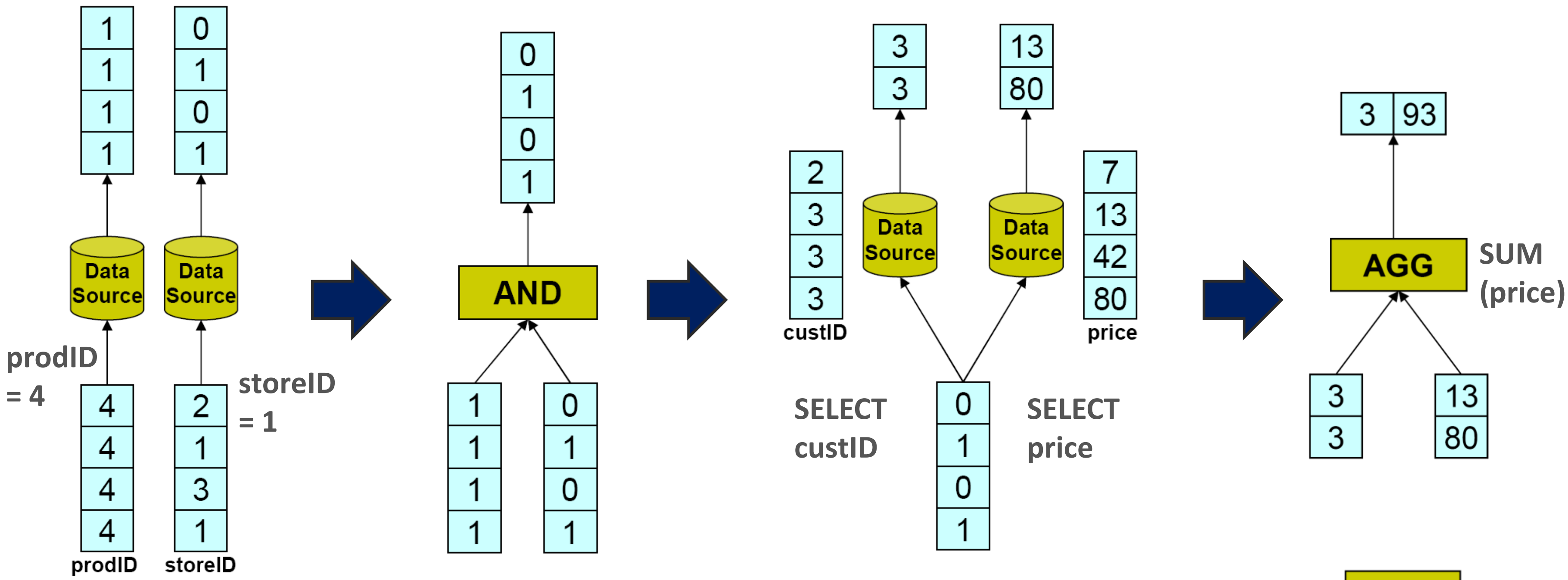
```
SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
          (storeID = 1) AND
GROUP BY custID
```

Create rows first

Disadvantages:

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

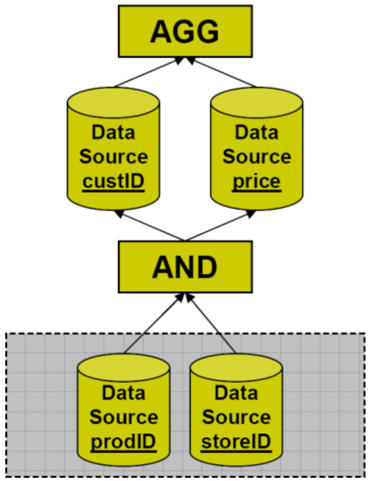
> Late Materialization Example



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

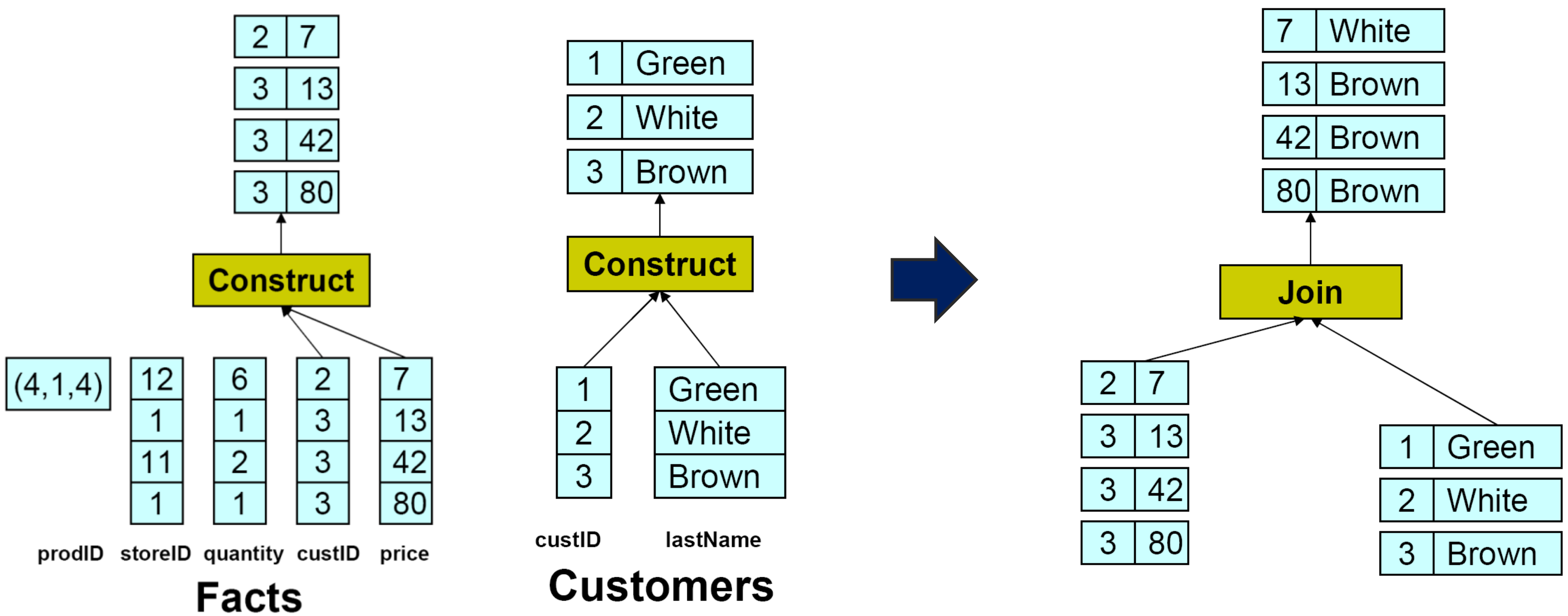
Advantages

- *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



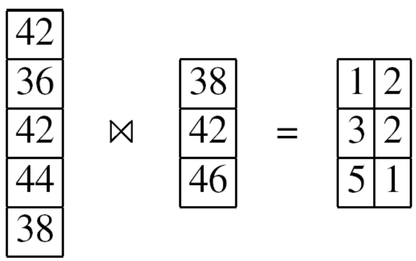
QUERY:

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

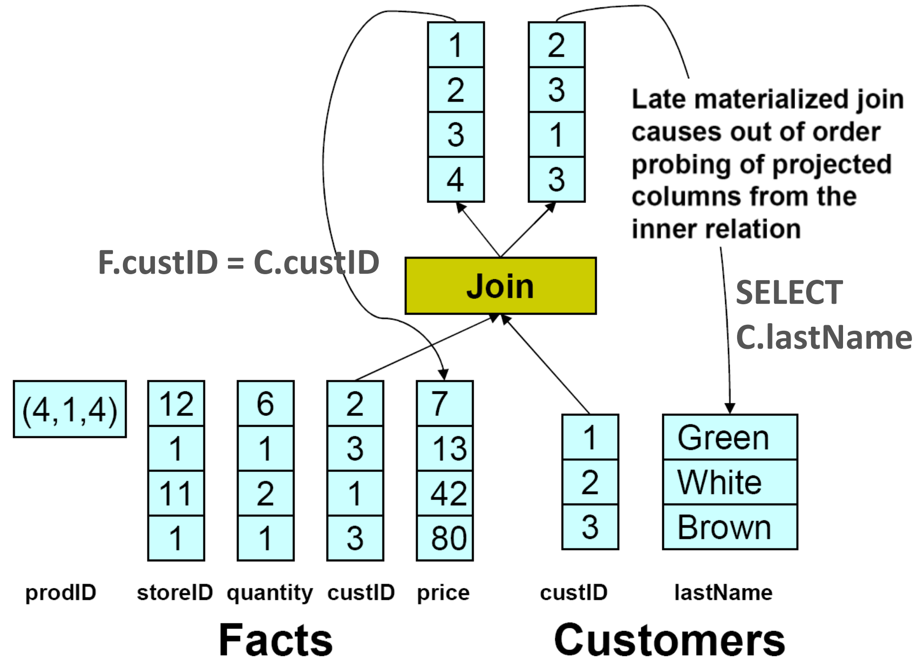
- Tuples have already been constructed before reaching the join operator
- Join functions as it would in a standard row-store system and outputs tuples

> 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
```

> Summary – Columnar Techniques

Storage

- Vertical partitioning of relations
- Header/ID elimination of columns
- Fast positional access
- Sparse indices, Multiple Sort orders of columns

Compression

- Operating on compressed data
- Lightweight, vectorized decompression

Late vs. Early materialization

- Non-join: LM always wins