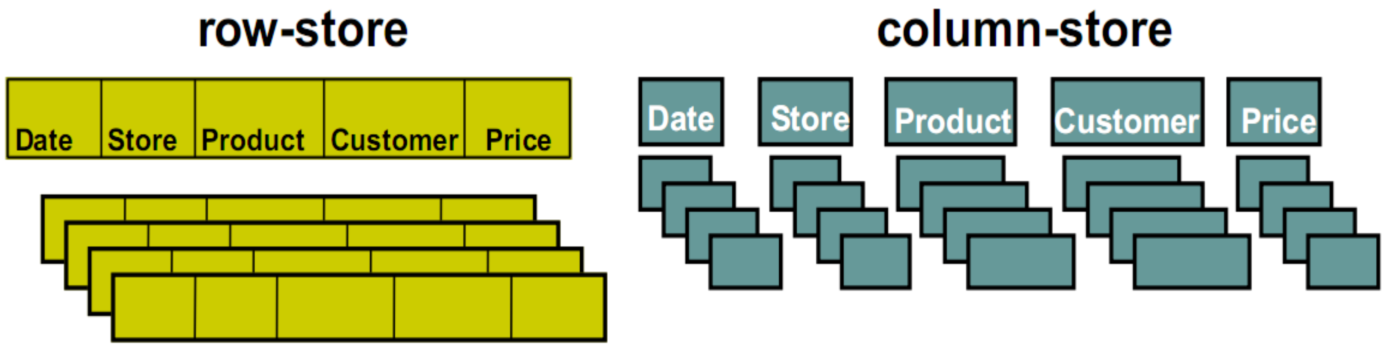
Column Store Tutorial

Xing Qu
https://github.com/tianjiqx

Contents

- Introduction
- Data model
- Encoding and Compression
- EM vs LM
- Vectorized process
- Main contribution factor Analysis
- Query execution on SSD

Row Store and Column Store



- In row store data are stored in the disk **tuple by tuple**
- Where in column store data are stored in the disk **column by column**

Why Column Stores?

- Can be significantly faster than row stores for some applications
  - Fetch only required columns for a query(telco example: 212 vs 7 columns)
  - Better compression (similar attribute values within a column)
    - Typical row-store compression ratio 1 : 3
    - Column-store 1 : 10
  - Sorting & indexing efficiency
    - Compression and dense-packing free up space
  - Block-tuple / vectorized processing,…
    - Better cache effects
    - Avoid decompression: operate directly on compressed
    - Delay decompression (and tuple reconstruction)
- But can be slower for other applications
  - OLTP with many row inserts(relieve:batch insert),…

Row Store and Column Store

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

- So column stores are suitable for **read-mostly(ad-hoc query)**, read-intensive, large data repositories such as Data Warehouse, CRM (Customer Relationship Management)…

Column Store System

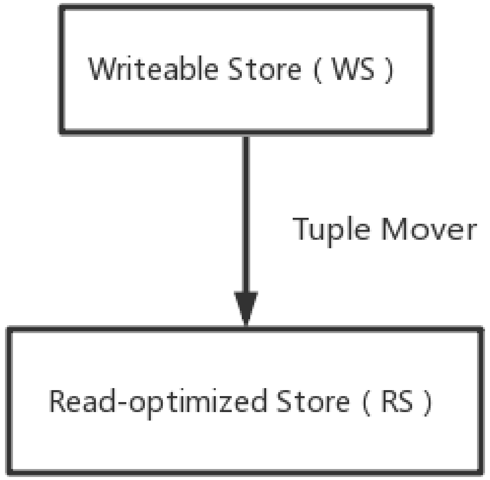
- Type demo:
  - C-Store
    - First comprehensive design description of a column-store
  - MonetDB/X100
    - “proper” disk-based column store
- Commercialized:
  - SybaseIQ
  - Vertica
  - Hive
  - SAP Business Accelerator
  - Greeplumn
  - Clickhouse
  - …

Data Model(C-Store)

- Standard relational logical data model
  - EMP(name, age, salary, dept)
  - DEPT(dname, floor)
- **Table** – covered by a set of projections
- **Projection** – set of columns
  - EMP1(name,age)
  - EMP2(dept,age,DEPT.floor)
  - EMP3(name,salary)
  - DEPT1(dname,floor)

C-Store:http://db.lcs.mit.edu/projects/cstore/

Data Model(C-Store)

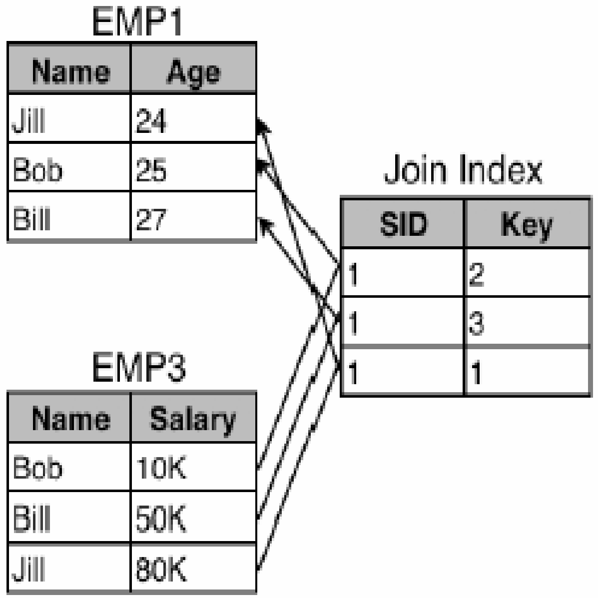


LSM-tree(Log-Structured Merge-Trees)

- **Sort key** – any column or columns in the projection (Self-order/Foreign-order)
  - 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
- **Storage Keys**:
  - Within a segment, every data value of every column is associated with a unique Skey
  - Values from different columns with matching Skey belong to the same logical row

Data Model(C-Store)

- **Join Indexes**:
  - T1 and T2 are projections on T
  - M segments in T1 and N segments in T2
  - Join Index from T1 to T2 is a table of the form:
    - (s: Segment ID in T2, k: Storage key in Segment s)
    - Each row in join index matches corresponding row in T
  - Join indexes are built such that T could be efficiently reconstructed from T1 and T2

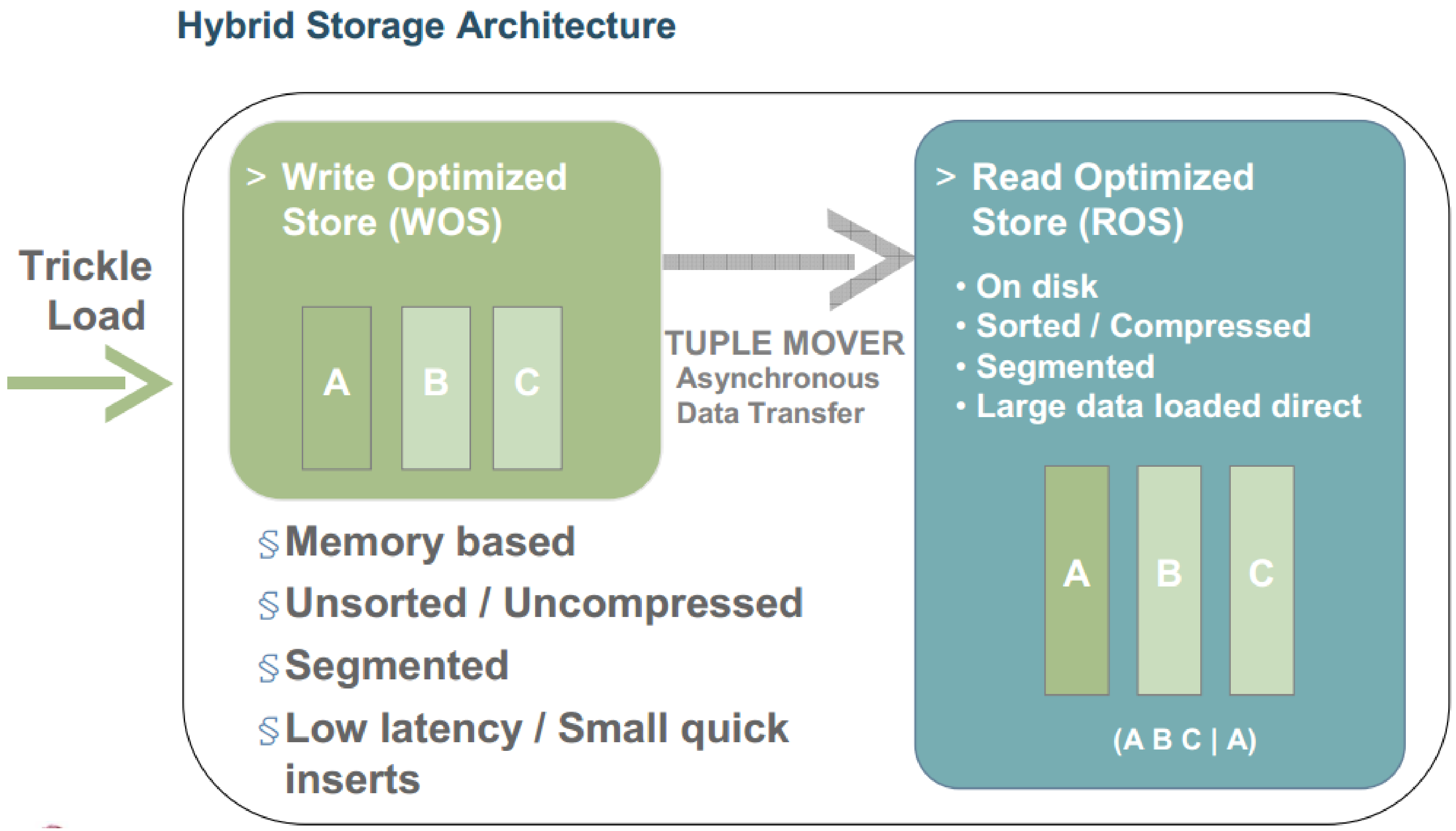


- Construct EMP(name, age, salary) from EMP1 and EMP3 using join index on EMP3 order by age

Data Model(Vertica)

- **Super projection** replace join index
  - One super projection containing every column of the anchoring table.
- Join index disadvantage:
  - Join indices were **complex to implemen**t and the runtime **cost of reconstructing** full tuples during distributed query execution was **very high**.
  - Explicitly storing row ids **consumed significant disk space** for large tables.

Architecture overview(Vertica)



Data Model(Vertica)

Original Data

sale_id	cid	cust	date	price
1	11	Andrew	01/01/06	$100
2	17	Chuck	01/05/06	$98
3	27	Nga	01/02/06	$90
4	28	Matt	01/03/06	$101
5	89	Ben	01/01/06	$103
1000	89	Ben	01/02/06	$103
1001	11	Andrew	01/03/06	$95

Split in two projections

date	price	cust	cid	sale_id

cust	price

Segmented on several nodes

Node 1

date	price	cid	cust	sale_id
01/02/06	$90.00	27	Nga	3
01/03/06	$95.00	11	Andrew	1001
01/03/06	$101.00	28	Matt	4

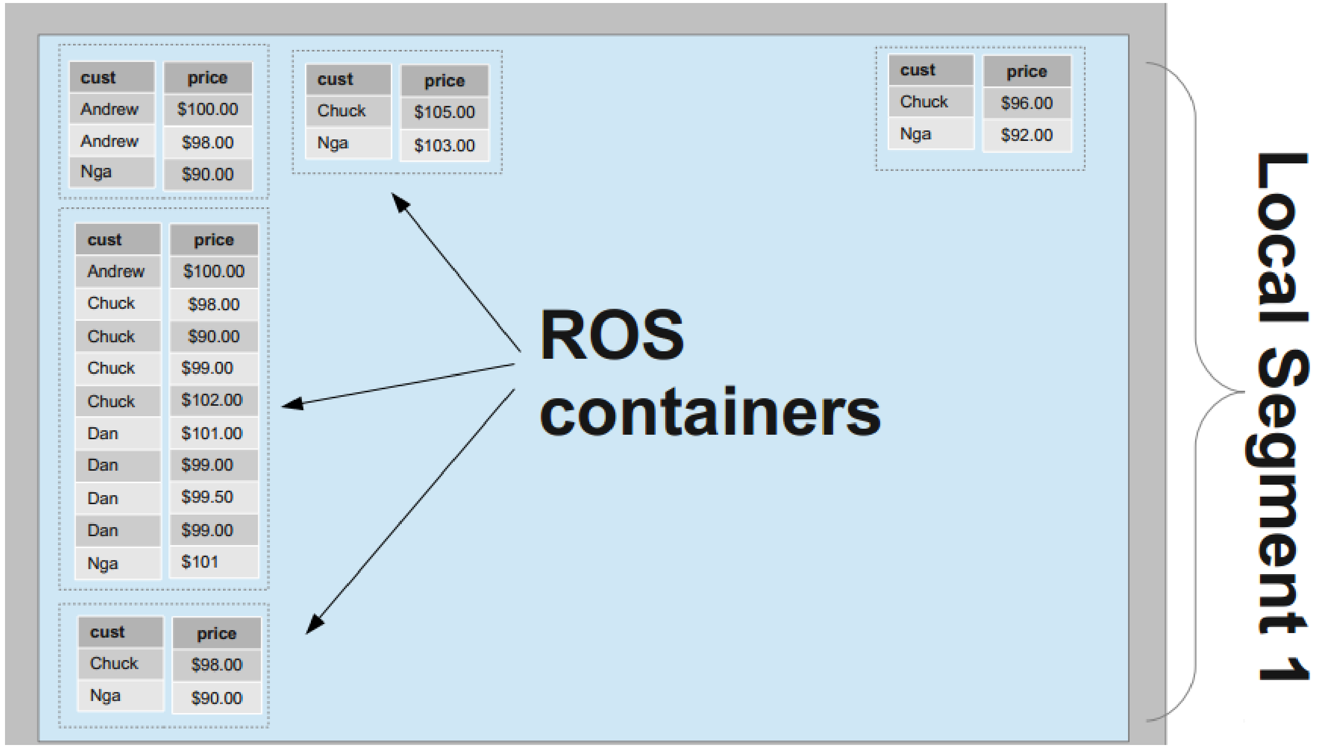
cust	price
Andrew	$95.00
Andrew	$100.00
Chuck	$98.00
Nga	$90.00

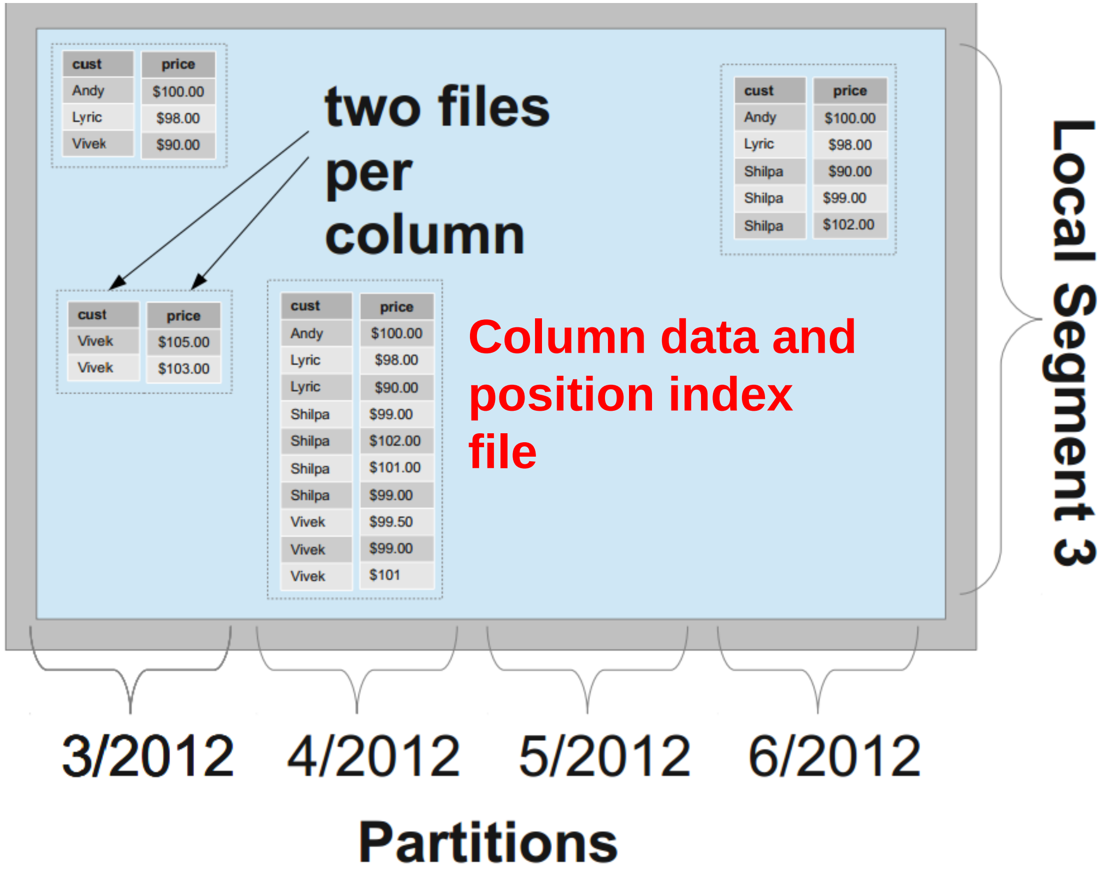
Node 2

date	price	cid	cust	sale_id
01/01/06	$100.00	11	Andrew	1
01/01/06	$103.00	89	Ben	5
01/02/06	$103.00	89	Ben	1000
01/05/06	$98.00	17	Chuck	2

cust	price
Ben	$103.00
Ben	$103.00
Matt	$101.00

Data Model(Vertica)





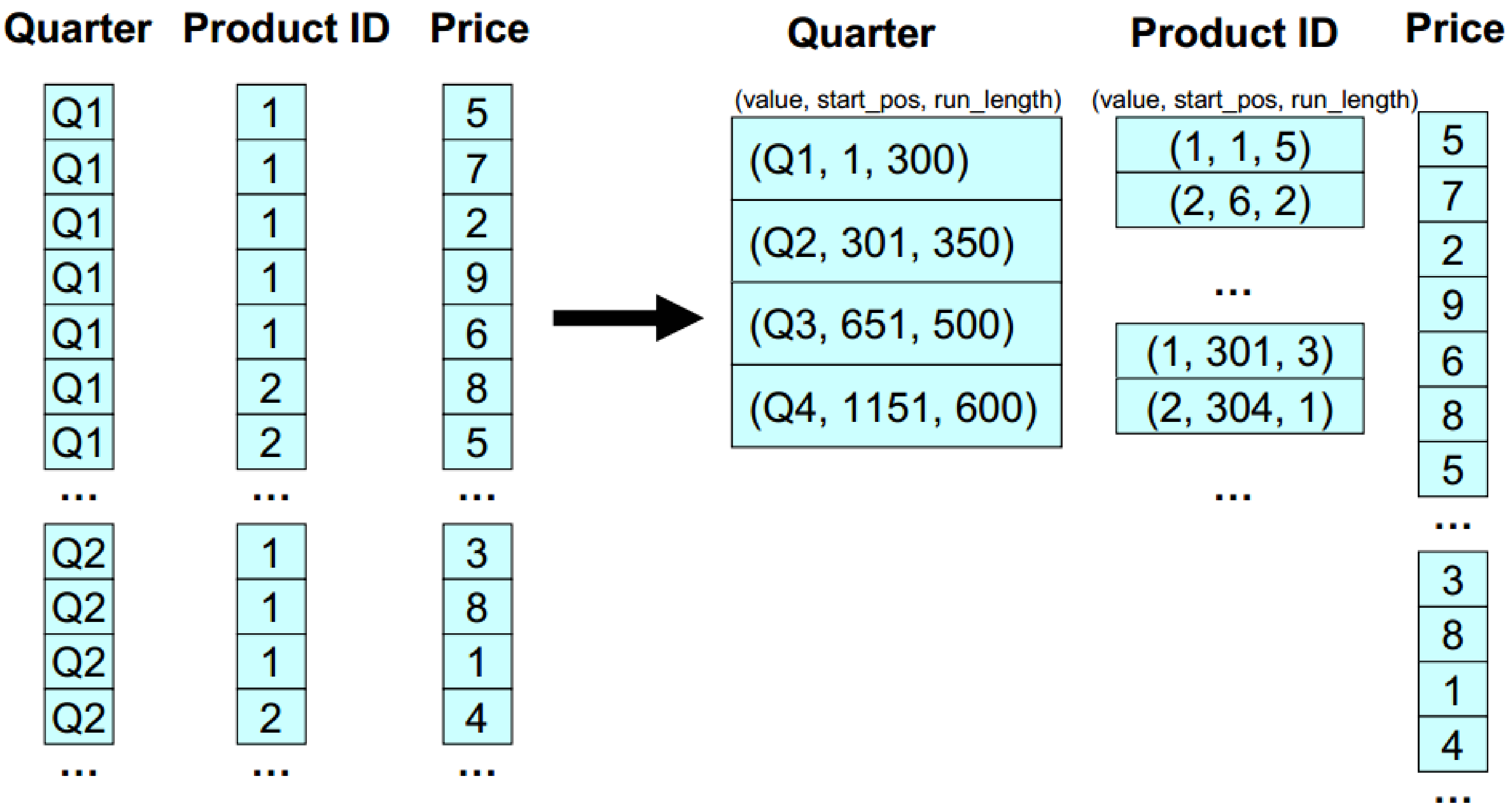
Read Optimized Store (ROS)
and a Write Optimized Store (WOS)

Encoding and Compression

- Trades I/O for CPU
- Increased column-store opportunities:
  - Higher data value locality in column stores
  - Techniques such as run length encoding far more useful
  - Can use extra space to store multiple copies of data in different sort orders

- Encoding Types(c-store/vertical):
  - Auto, RLE, Delta Value, Block Dictionary, Compressed Delta Range, Compressed Common Delta

Run-length Encoding(RLE)



Abadi, D. J., Madden, S. R., & Ferreira, M. C. (n.d.). **Integrating Compression and Execution in Column-Oriented Database Systems**.sigmod06.

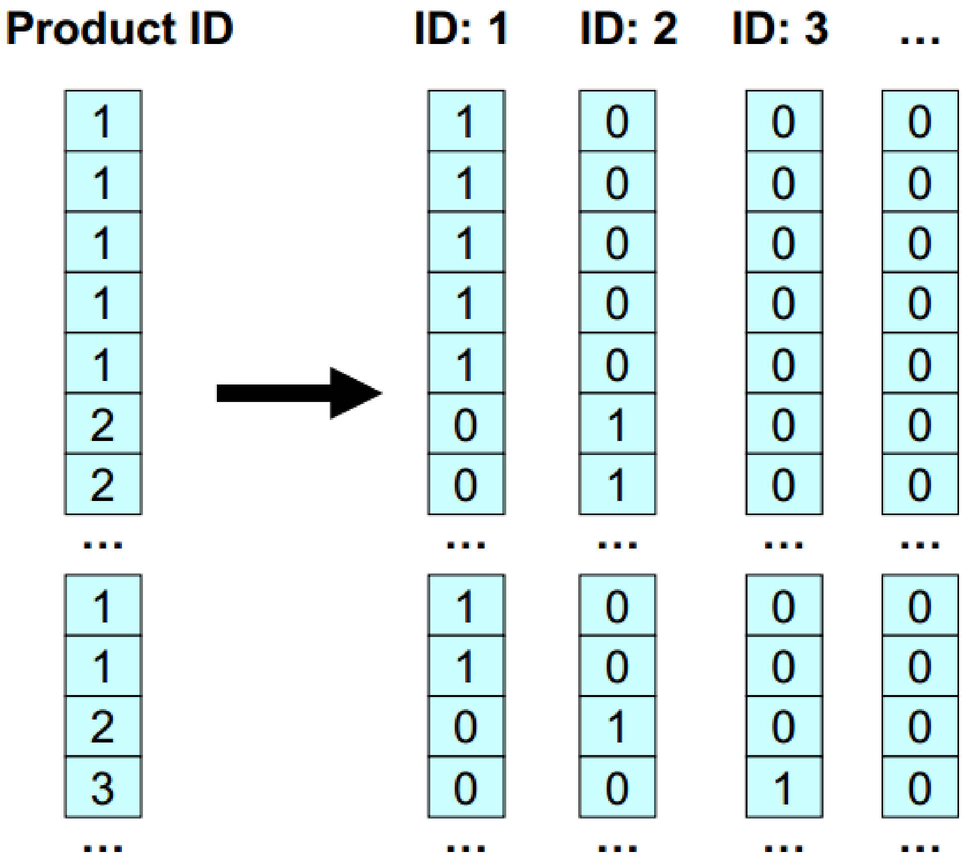
Bit-vector Encoding

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

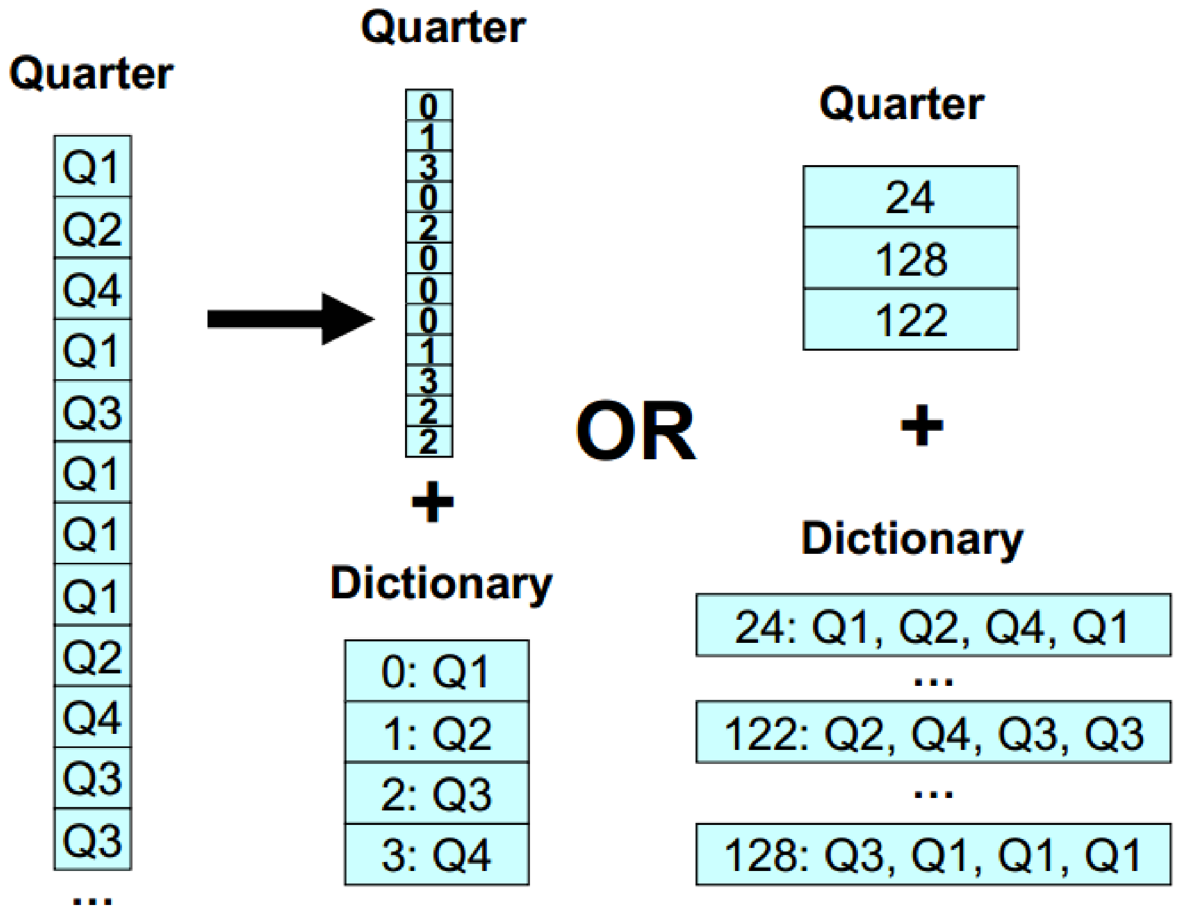
Each bit-vector can be further compressed if sparse



Abadi, D. J., Madden, S. R., & Ferreira, M. C. (n.d.). **Integrating Compression and Execution in Column-Oriented Database Systems**.sigmod06.

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

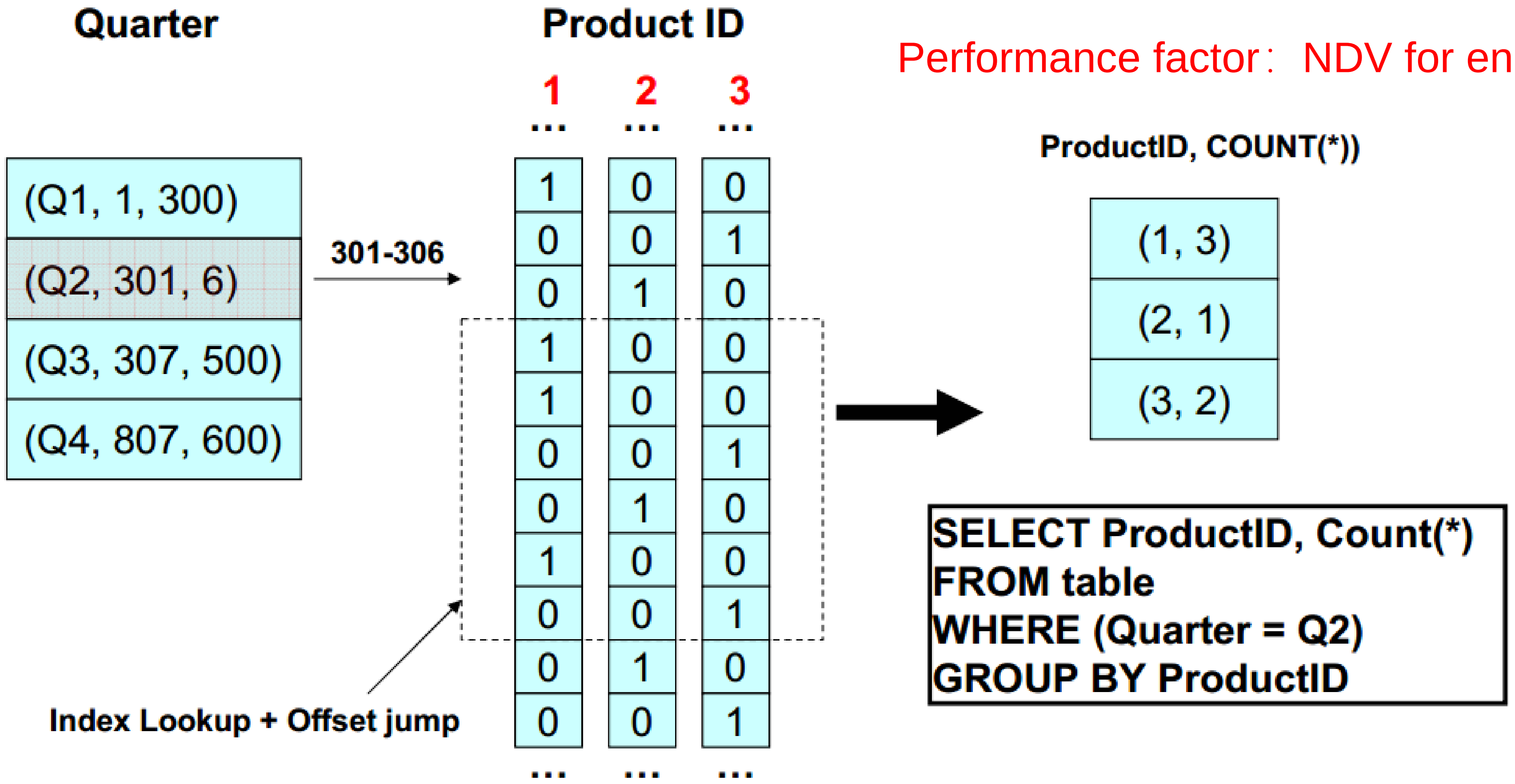


Abadi, D. J., Madden, S. R., & Ferreira, M. C. (n.d.). **Integrating Compression and Execution in Column-Oriented Database Systems**.sigmod06.

Operating Directly on Compressed Data

- I/O - CPU tradeoff is no longer a tradeoff
- Reduces memory–CPU bandwidth requirements
- Opens up possibility of operating on multiple records at once

Performance factor： NDV for encoding

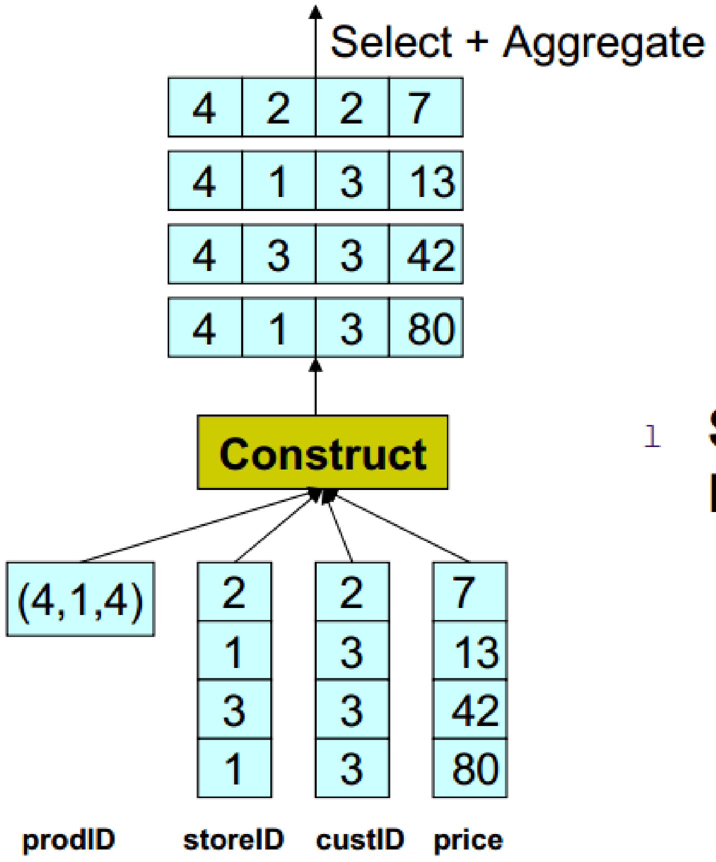


Abadi, D. J., Madden, S. R., & Ferreira, M. C. (n.d.). **Integrating Compression and Execution in Column-Oriented Database Systems**.sigmod06.

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
    - This process is called “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

When should tuples be constructed?

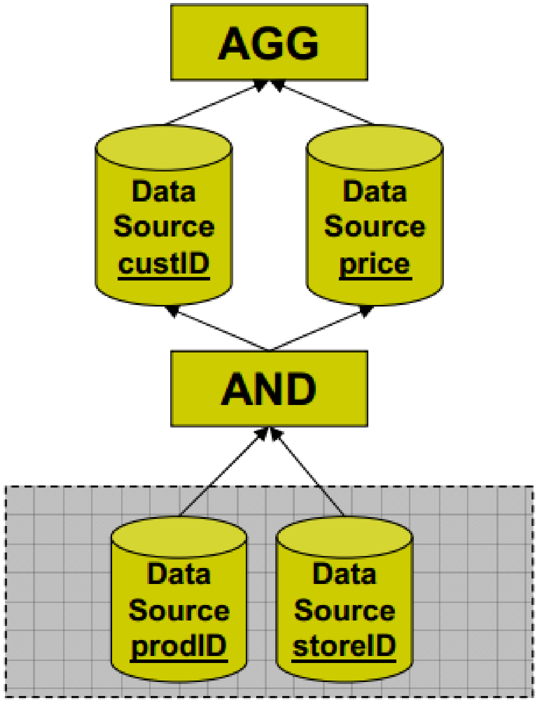


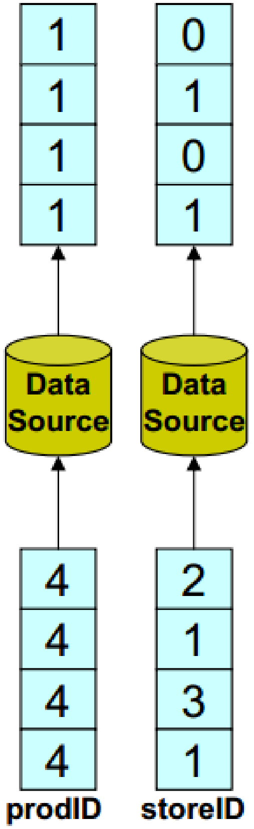
QUERY:

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

- **Solution 1: Create rows first (EM). But:**
  - **Need to construct ALL tuples**
  - **Need to decompress data**
  - **Poor memory bandwidth utilization**

Solution 2: Operate on columns



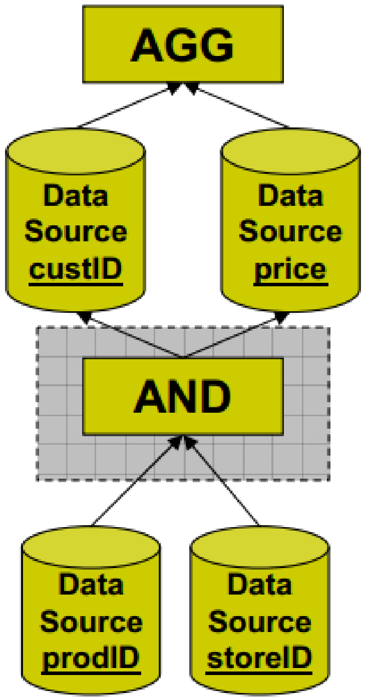


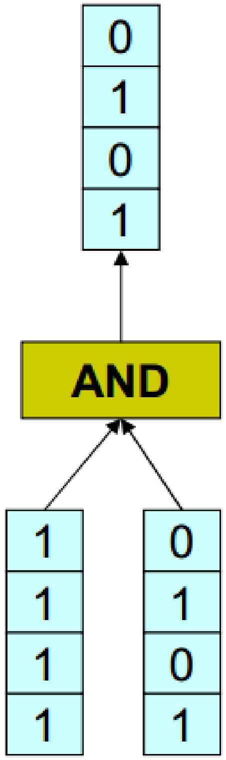
QUERY:

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

Solution 2: Operate on columns

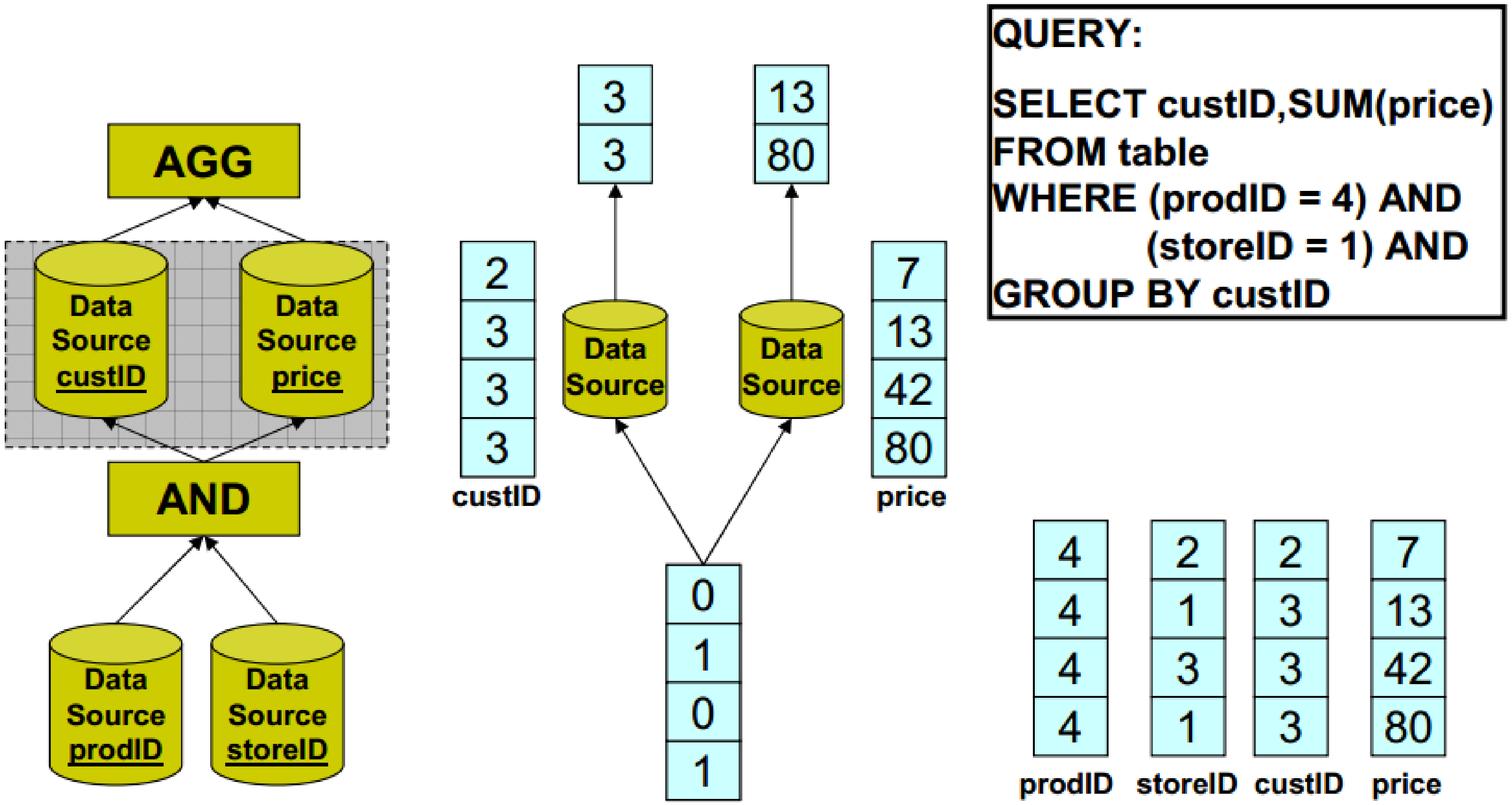




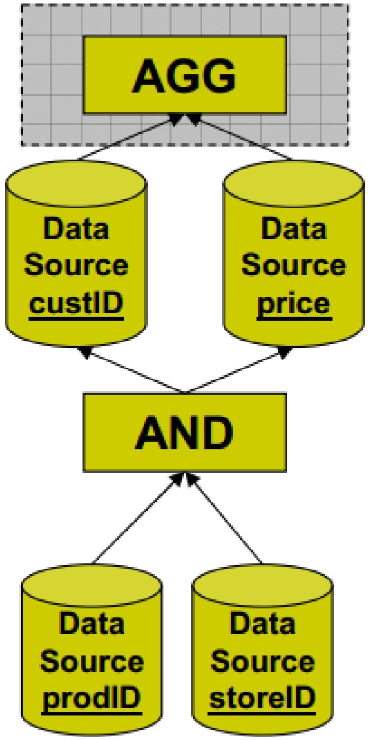

```
QUERY:

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

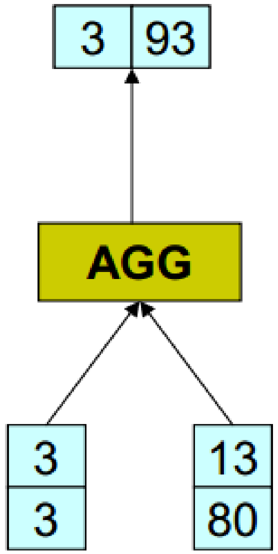
prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

Solution 2: Operate on columns



Solution 2: Operate on columns



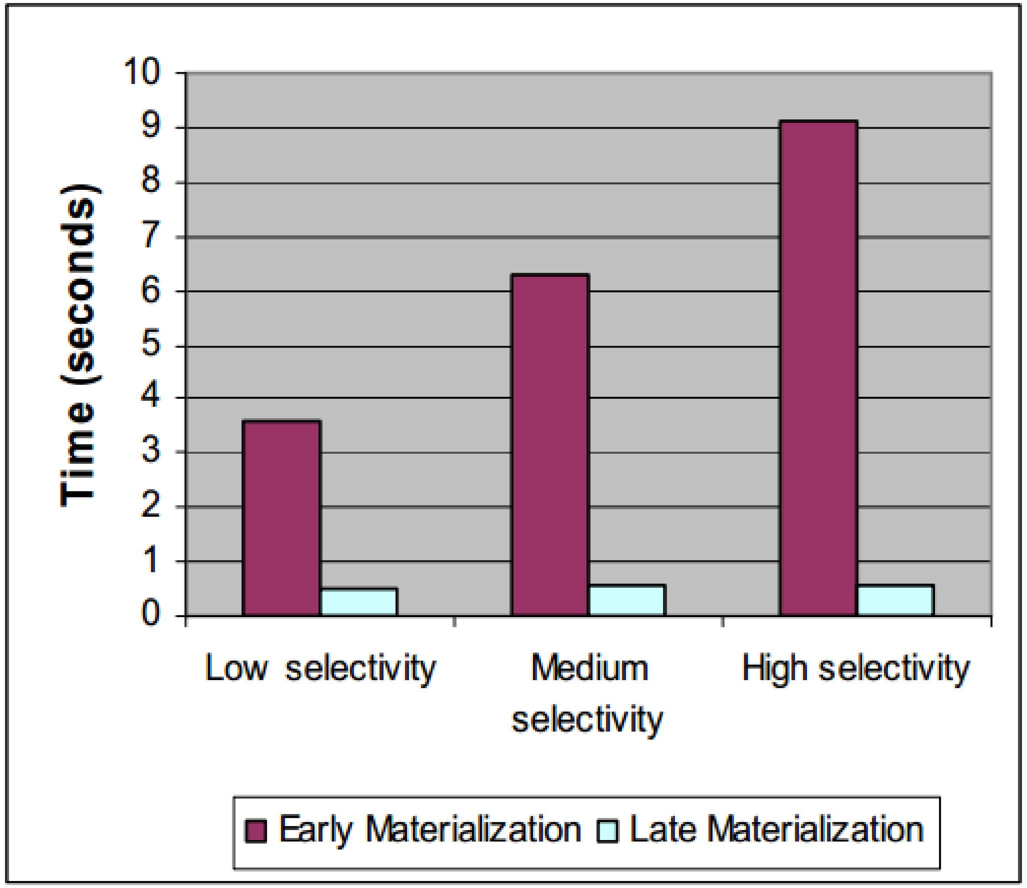


QUERY:

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

For plans without joins, late materialization is a win



QUERY:

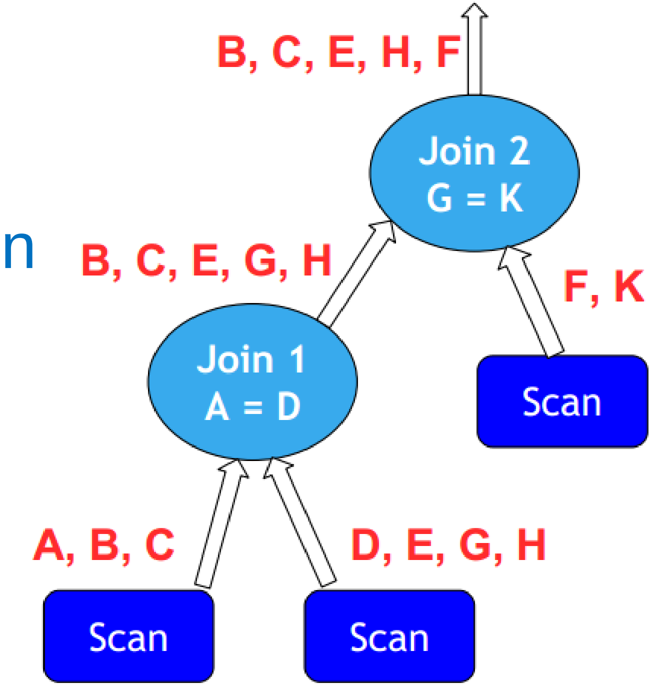
```
SELECT C1, SUM(C2)
FROM table
WHERE (C1 < CONST) AND
      (C2 < CONST)
GROUP BY C1
```

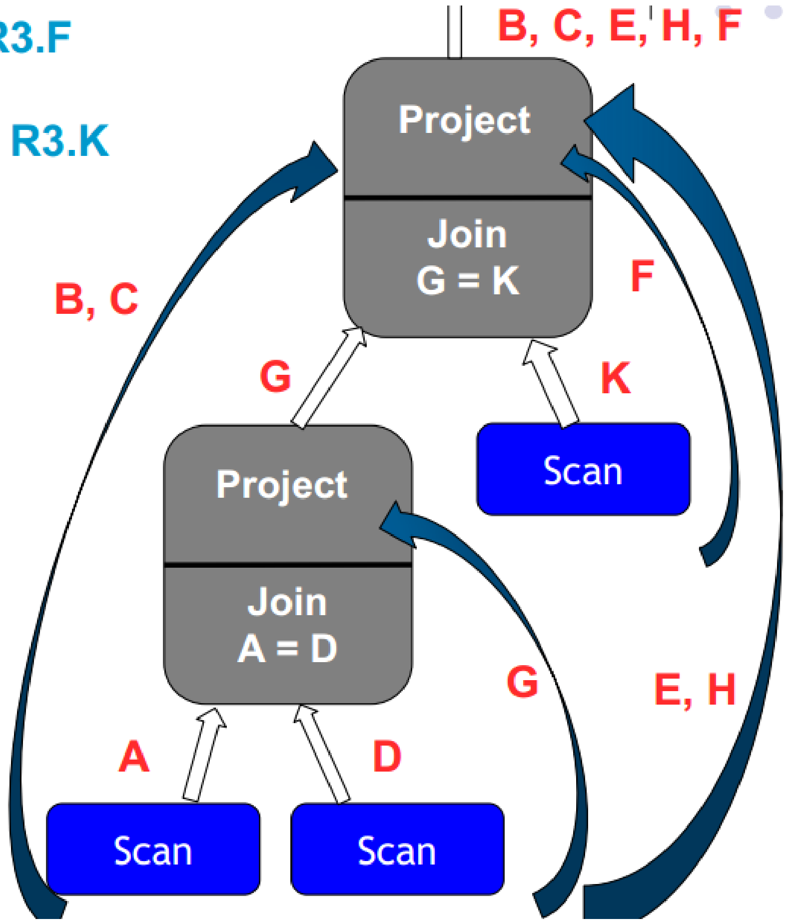
1 **Ran on 2 compressed columns from TPC-H scale 10 data**

What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K

Early materialization





Late materialization

Random IO!

Naive LM join about 2X slower than EM join on typical queries (due to random I/O)

Invisible Join

- Designed for typical joins when data is modeled using a star schema
  - One ("fact") table is joined with multiple dimension tables
- Typical query:

```
select c_nation, s_nation, d_year,
        sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
   and lo_suppkey = s_suppkey
   and lo_orderdate = d_datekey
   and c_region = 'ASIA'
   and s_region = 'ASIA'
   and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Abadi, D. J., Madden, S. R., & Hachem, N. (2008). **Column-stores vs. row-stores**. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08, 967.

Invisible Join

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	...

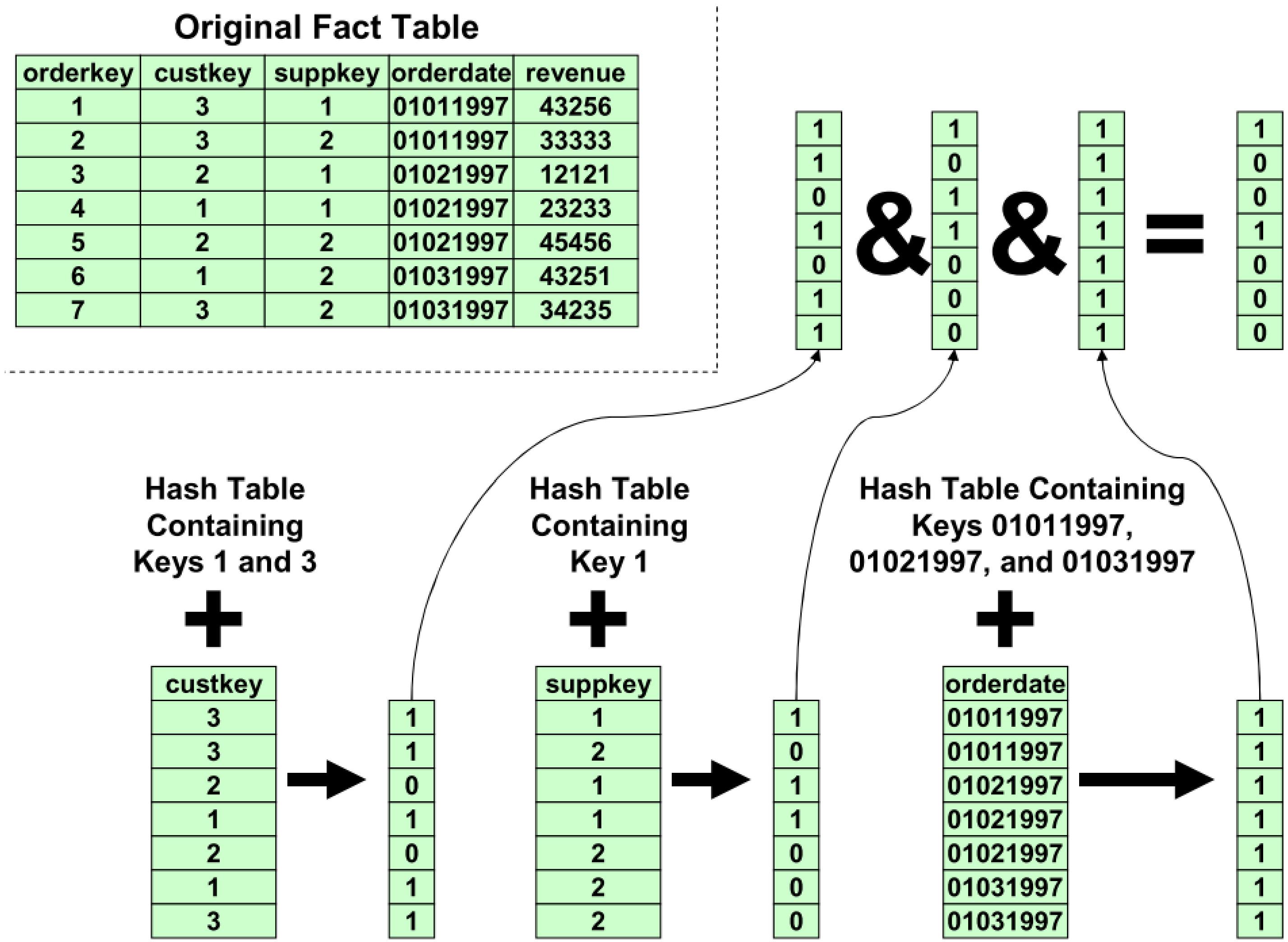
→ **Hash Table Containing Key 1**

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

dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

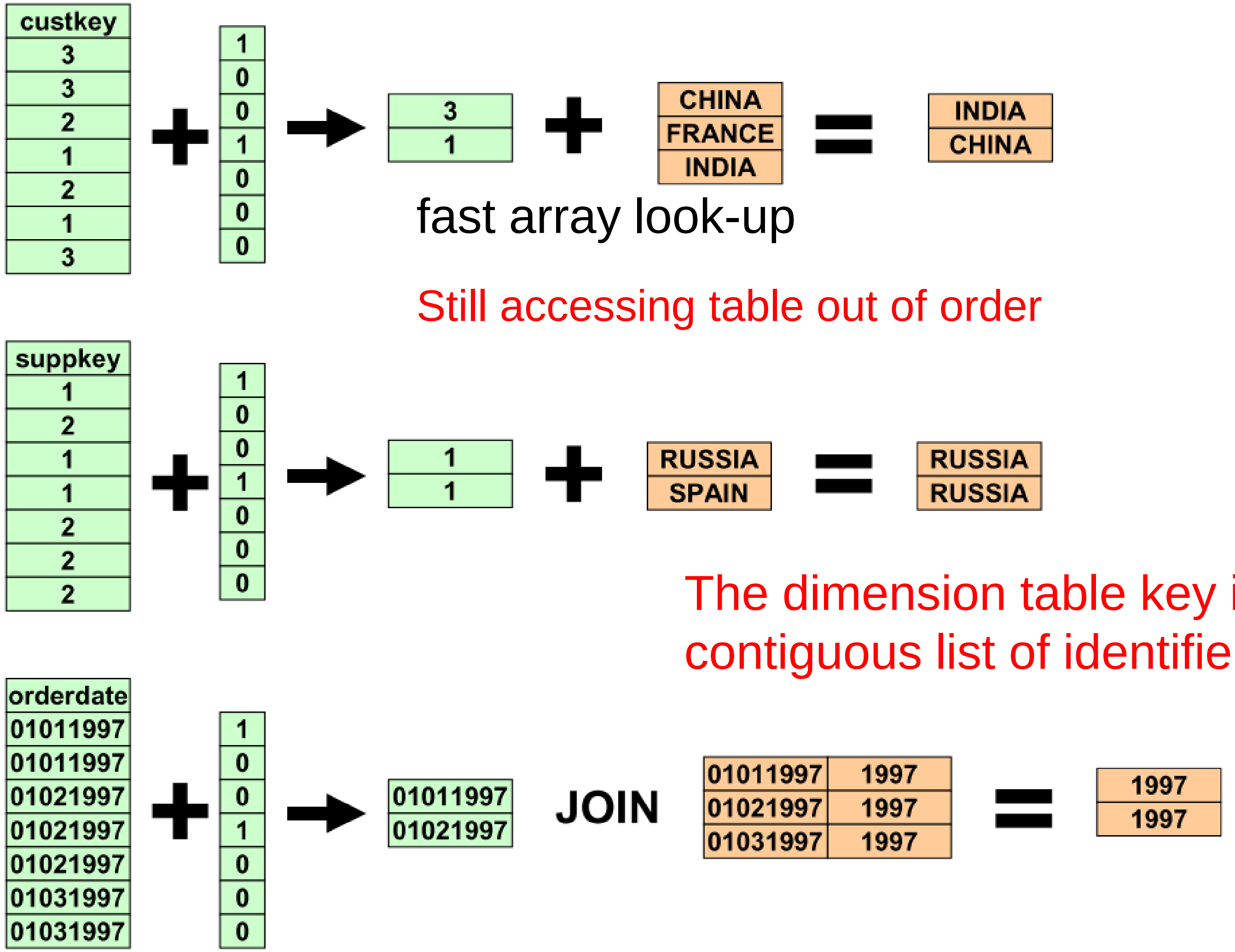
→ **Hash Table Containing Keys 01011997, 01021997, and 01031997**

Invisible Join



Invisible Join

2X EM join

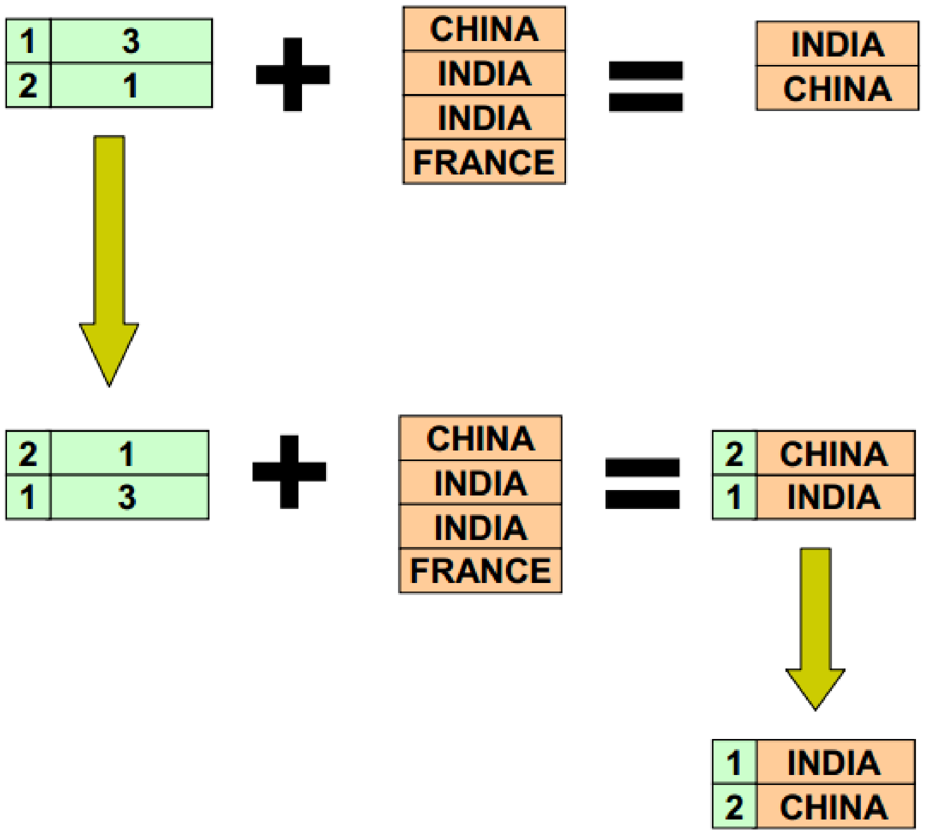


Still accessing table out of order

The dimension table key is a sorted, contiguous list of identifiers starting from 1

Jive/Flash Join

1. Add column with dense ascending integers from 1
2. Sort new position list by second column
3. Probe projected column in order using new sorted position list, keeping first column from position list around
4. Sort new result by first column



CPU or Disk?

- ~~“save disk I/O when scan-intensive queries need a few columns”~~
- “avoid an expression interpreter to improve computational efficiency”

- loop pipelining:
  - F(A[0])G(A[0]) F(A[1])G(A[1]) F(A[2])G(A[2]) F(A[3])G(A[3]) =>
  - F(A[0]) F(A[1]) F(A[2]) G(A[0]) G(A[1]) G(A[2]) F(A[3])…

Boncz, P., Zukowski, M., & Nes, N. (2005**). MonetDB/X100: Hyper-Pipelining Query Execution**. *CIDR ’05: Second Biennial Conference on Innovative Data Systems Research*, 225–237.

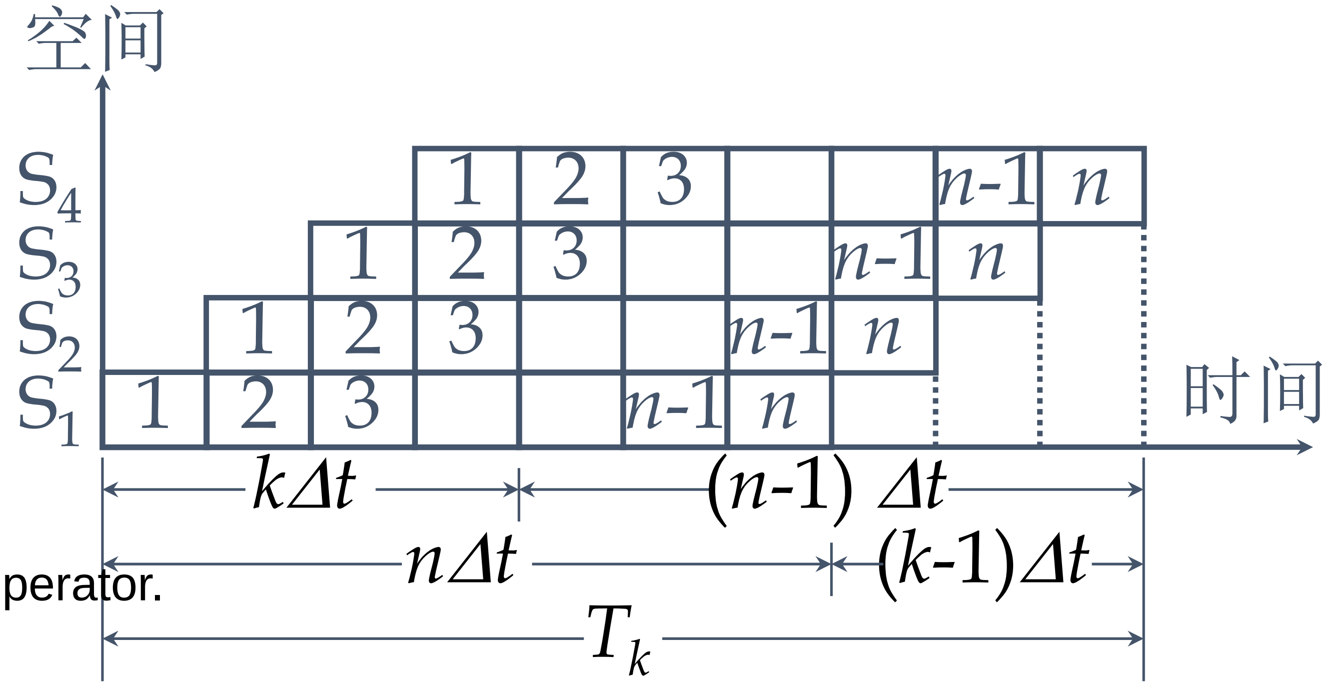
Loop pipeline

- CPU cache friendly for vectorized execution
  - A physical lower bound on memory latency of around **50 ns**.
  - This (ideal) minimum latency of 50ns already translates into **180 wait cycles** for a 3.6GHz CPU.

CPU cache vs memory
cache 5~10X

- Thoughput rate (1/ Δ t)
- Speedup ratio (k)
- Efficiency (1)

- Where multiplication in MonetDB/MIL was constrained by the RAM bandwidth of **500MB/s**, MonetDB/X100 exceeds **7.5GB/s** on the same operator.

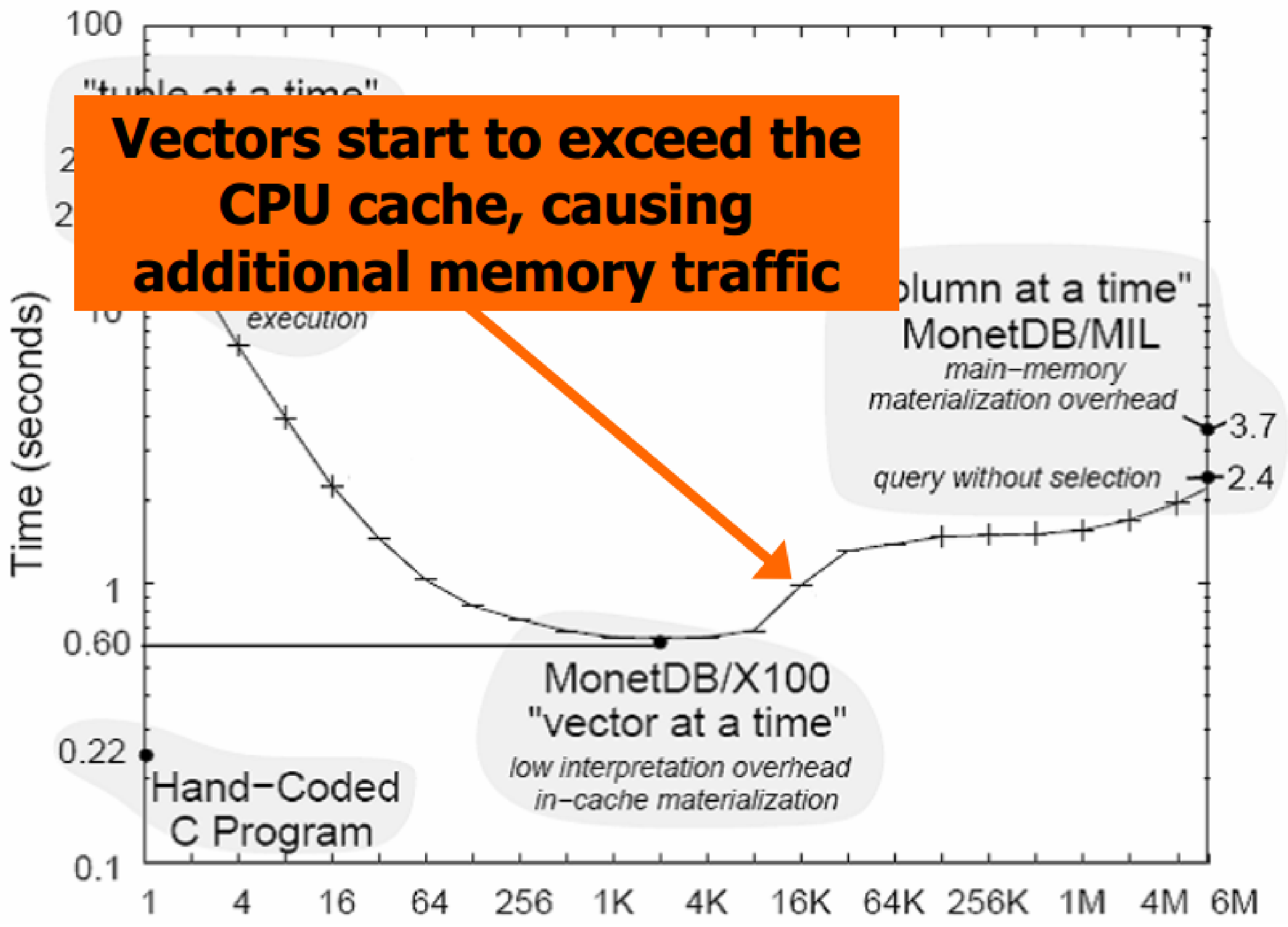


CPU latency: https://www.expreview.com/60725-5.html

pipeline

Varying the vector size

1024 row~L1+L2



Itanium 2 :
16KB L1
256KB L2
3MB L3

Benefits of vectorized processing

- **100x less function calls**
  - iterator.next().primitives
- No Instruction Cache Misses
- **Less Data Cache Misses**
- No Tuple Navigation
  - Primitives are record-oblivious, only see arrays
- Vectorization allows algorithmic optimization
  - Move activities out of the loop (“strength reduction”)
- **Compiler-friendly function bodies**
  - Loop-pipelining, automatic SIMD

Which of tech are most significant?

- Late materialization X3
- Block iteration(vectorized query processing)
- Column-specific compression X2
- invisible joins X2

Abadi, D. J., Madden, S. R., & Hachem, N. (2008). **Column-stores vs. row-stores**. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08, 967.

Column Store base on SSD

- Old designed:
  - the speed mismatch between random and sequential I/O on hard disks
  - and their algorithms and data struct currently **emphasize sequential accesses for disk-resident** data.
- Column store on SSD:
  - **<u>leverage fast random reads</u>** to speed up selection, projection, and join operations in relational query processing.(scan,join)
  - FlashJoin:
    - A general pipelined join algorithm that **minimizes accesses to base and intermediate relational data.**
    - FlashJoin's binary join kernel **accesses only the join attributes**, **fetch** kernel retrieves the attributes for **later** nodes in the query plan as they are needed.
    - Reduces memory and I/O requirements

Tsirogiannis, D., Harizopoulos, S., Shah, M. A., Wiener, J. L., & Graefe, G. (2009). **Query processing techniques for solid state drives**, 59.

Query optimizer(Vertica)

- Key Point:
  - joining projections with **highly compressed**
  - and **sorted** predicate
  - most **highly selective** dimensions
  - **Late materialization**
  - **Compression- aware costing** and planning, stream aggregation, sort elimi- nation, and merge joins
  - Distribution aware

References

- Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., … Zdonik, S. (2005). **C-store: a column-oriented DBMS**. *VLDB Conference*, 553–564.
- Abadi, D. J., Madden, S. R., & Hachem, N. (2008). **Column-stores vs. row-stores**. *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08*, 967.
- Boncz, P., Zukowski, M., & Nes, N. (2005). **MonetDB/X100: Hyper-Pipelining Query Execution**. *CIDR '05: Second Biennial Conference on Innovative Data Systems Research*, 225–237.
- Abadi, D. J., & Madden, S. (2008). **Query Execution in Column-Oriented Database Systems**.

- Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., & Bear, C. (2014). **The vertica analytic database**. *Proceedings of the VLDB Endowment*, *5*(12), 1790–1801.
- Tsirogiannis, D., Harizopoulos, S., Shah, M. A., Wiener, J. L., & Graefe, G. (2009). **Query processing techniques for solid state drives**, 59.
- 开源OLAP引擎测评报告:https://zhuanlan.zhihu.com/p/55197560
- Greenplum数据库文档:https://gp-docs-cn.github.io/docs/best_practices/schema.html