Column-Oriented Database Systems

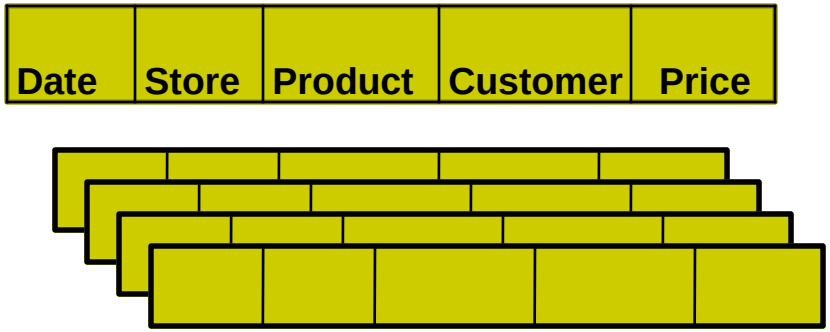
Tutorial

Peter Boncz (CWI)

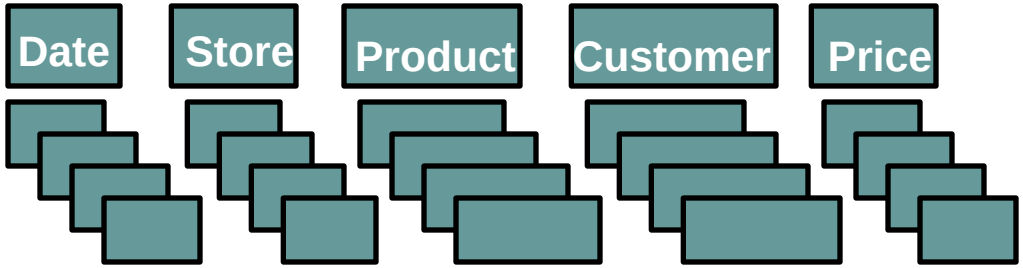
Adapted from VLDB 2009 Tutorial
Column-Oriented Database Systems
with
Daniel Abadi (Yale)
Stavros Harizopuolos (HP Labs)

What is a column-store?

row-store



column-store



+ easy to add/modify a record

- might read in unnecessary data

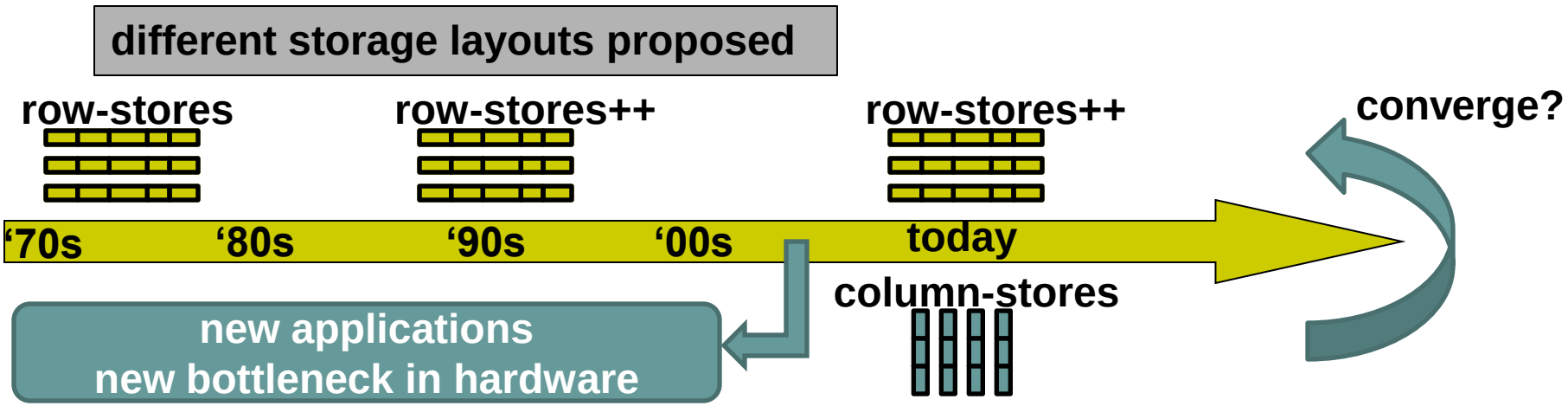
+ only need to read in relevant data

- tuple writes require multiple accesses

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

Are these two fundamentally different?

- The only fundamental difference is the storage layout
- However: we need to look at the big picture



- How did we get here, and where we are heading
- What are the column-specific optimizations?
- How do we improve CPU efficiency when operating on Cs

Part 1

Part 2

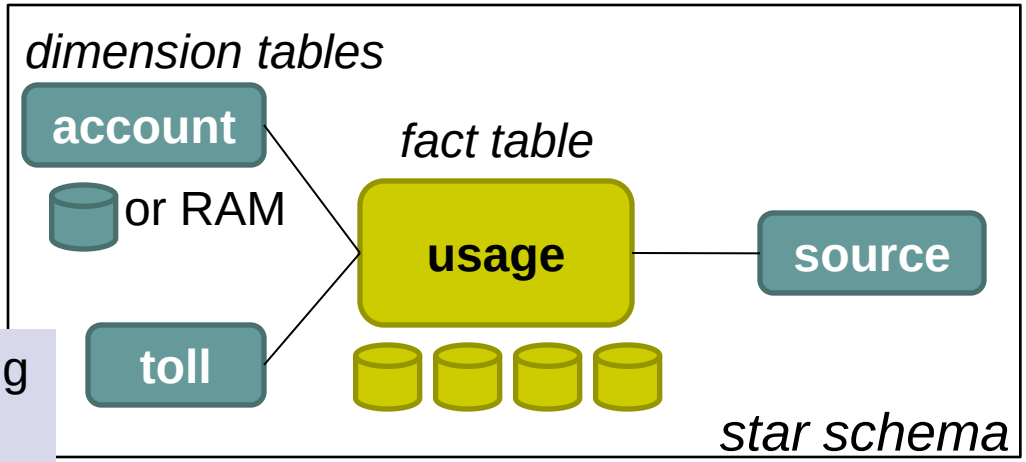
Part 3

Outline

- Part 1: Basic concepts
  - Introduction to key features
  - From DSM to column-stores and performance tradeoffs
  - Column-store architecture overview
  - Will rows and columns ever converge?
- Part 2: Column-oriented execution
- Part 3: MonetDB/X100 (“VectorWise”) and CPU efficiency

Telco Data Warehousing example

- Typical DW installation
- Real-world example



"One Size Fits All? - Part 2: Benchmarking Results" Stonebraker et al. CIDR 2007

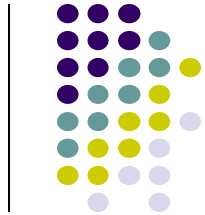
QUERY 2

```
SELECT account.account_number,
sum (usage.toll_airtime),
sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_ind in ('AE'. 'AA')
AND usage.toll_price > 0
AND source.type != 'CIBER'
AND toll.rating_method = 'IS'
AND usage.invoice_date = 20051013
GROUP BY account.account_number
```

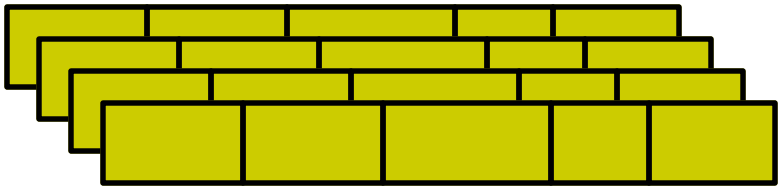
	Column-store	*Row-store*
Query 1	*2.06*	*300*
Query 2	*2.20*	*300*
Query 3	*0.09*	*300*
Query 4	*5.24*	*300*
Query 5	*2.88*	*300*

Why? Three main factors (next slides)

Telco example explained (1/3): *read efficiency*



row store



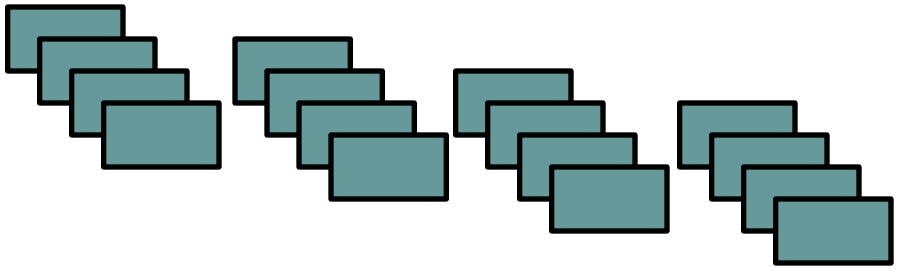
read pages containing entire rows

one row = 212 columns!

is this typical? (it depends)

What about vertical partitioning?
(it does not work with ad-hoc queries)

column store



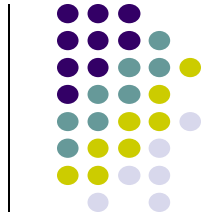
read only columns needed

in this example: 7 columns

caveats:

- “select * ” not any faster
- clever disk prefetching
- clever tuple reconstruction

Telco example explained (2/3): *compression efficiency*



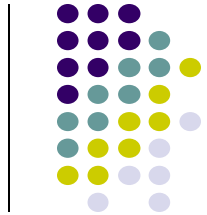
- Columns compress better than rows
  - Typical row-store compression ratio 1 : 3
  - Column-store 1 : 10

- Why?
  - Rows contain values from different domains
    => more entropy, difficult to dense-pack
  - Columns exhibit significantly less entropy
  - Examples:

Male, Female, Female, Female, Male
1998, 1998, 1999, 1999, 1999, 2000

  - Caveat: CPU cost (use lightweight compression)

Telco example explained (3/3): *sorting & indexing efficiency*



- Compression and dense-packing free up space
  - Use multiple overlapping column collections
  - Sorted columns compress better
  - Range queries are faster
  - Use sparse clustered indexes

What about heavily-indexed row-stores?
(works well for single column access,
cross-column joins become increasingly expensive)

Additional opportunities for column-stores

- Block-tuple / vectorized processing
  - Easier to build block-tuple operators
    - Amortizes function-call cost, improves CPU cache performance
  - Easier to apply vectorized primitives
    - Software-based: bitwise operations
    - Hardware-based: SIMD

Part 3

- Opportunities with compressed columns
  - *Avoid* decompression: operate directly on compressed
  - *Delay* decompression (and tuple reconstruction)
    - Also known as: *late materialization*

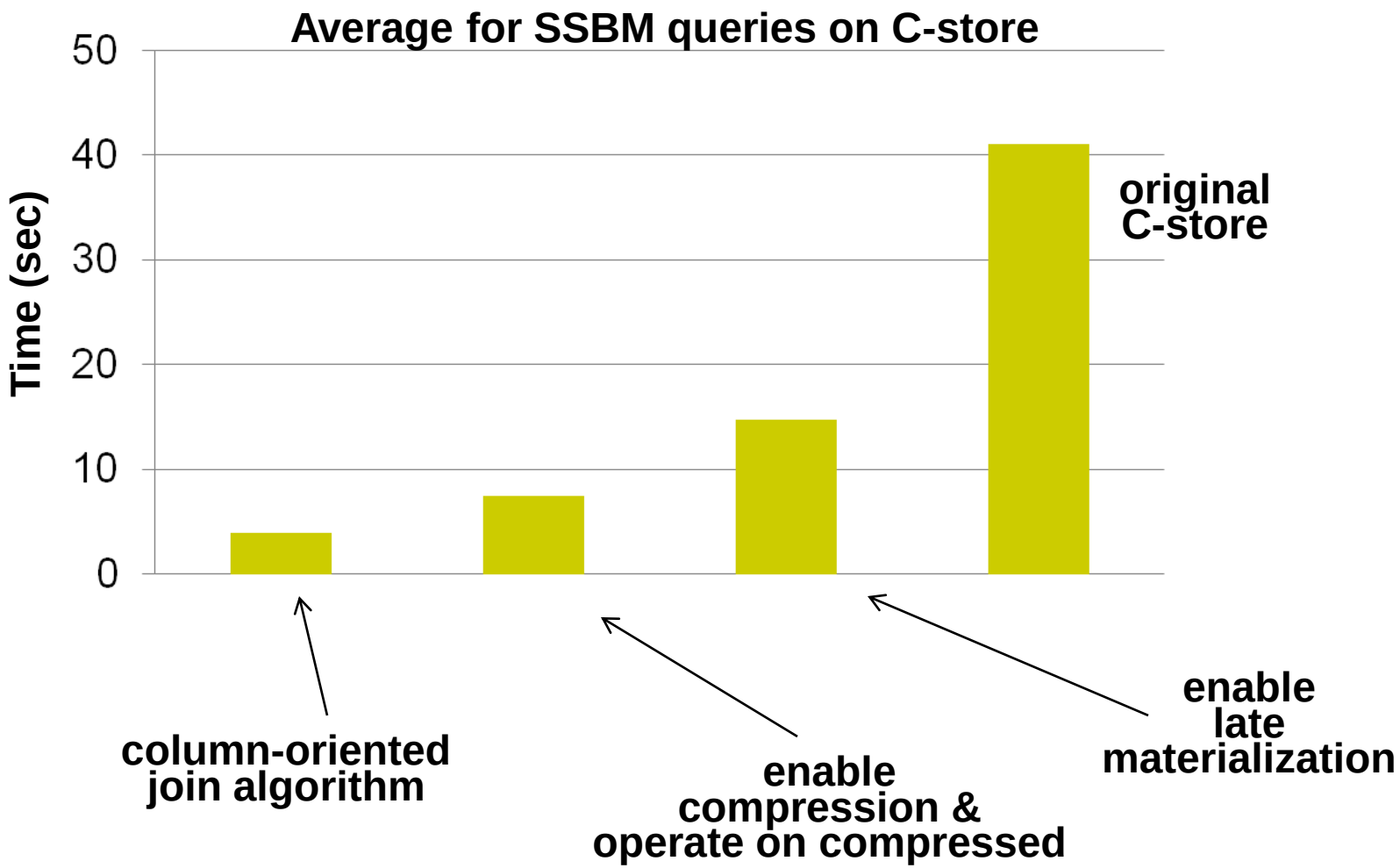
more in Part 2

- Exploit columnar storage in other DBMS components
  - Physical design (both static and dynamic)

See: *Database Cracking*, from CWI

Effect on C-Store performance

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Hachem, and Madden. SIGMOD 2008.



Summary of column-store key features

- Storage layout
  - columnar storage
  - header/ID elimination — Part 1
  - compression — Part 2, Part 3
  - multiple sort orders
- Execution engine
  - column operators — Part 1, Part 2
  - avoid decompression
  - late materialization — Part 2
  - vectorized operations — Part 3
- Design tools, optimizer

Outline

- Part 1: Basic concepts
  - Introduction to key features
  - From DSM to column-stores and performance tradeoffs
  - Column-store architecture overview
  - Will rows and columns ever converge?
- Part 2: Column-oriented execution
- Part 3: MonetDB/X100 (“VectorWise”) and CPU efficiency

From DSM to Column-stores

70s -1985:

TOD: Time Oriented Database – Wiederhold et al.
"A Modular, Self-Describing Clinical Databank System,"
Computers and Biomedical Research, 1975
More 1970s: Transposed files, Lorie, Batory, Svensson.

"An overview of cantor: a new system for data analysis"
Karasalo, Svensson, SSDBM 1983

1985: DSM paper

"A decomposition storage model"
Copeland and Khoshafian. SIGMOD 1985.

1990s: Commercialization through SybaseIQ

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

- DSM "on steroids" [1997 – now] CWI: MonetDB
- Hybrid DSM/NSM [2001 – 2004] Wisconsin: PAX, Fractured Mirrors

Michigan: Data Morphing CMU: Clotho

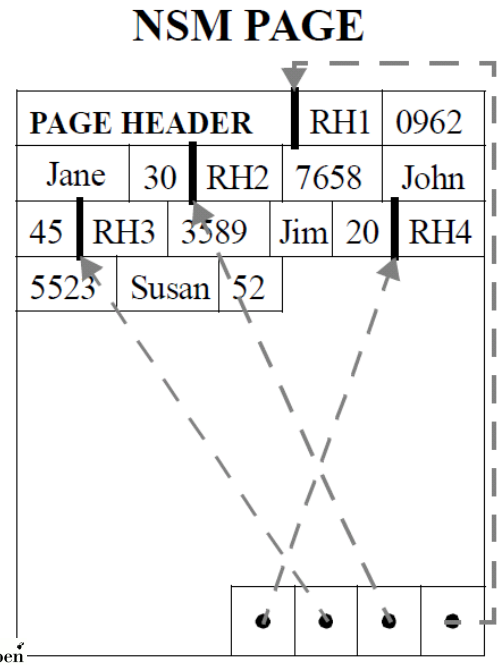
2005 – : Re-birth of read-optimized DSM as "column-store"

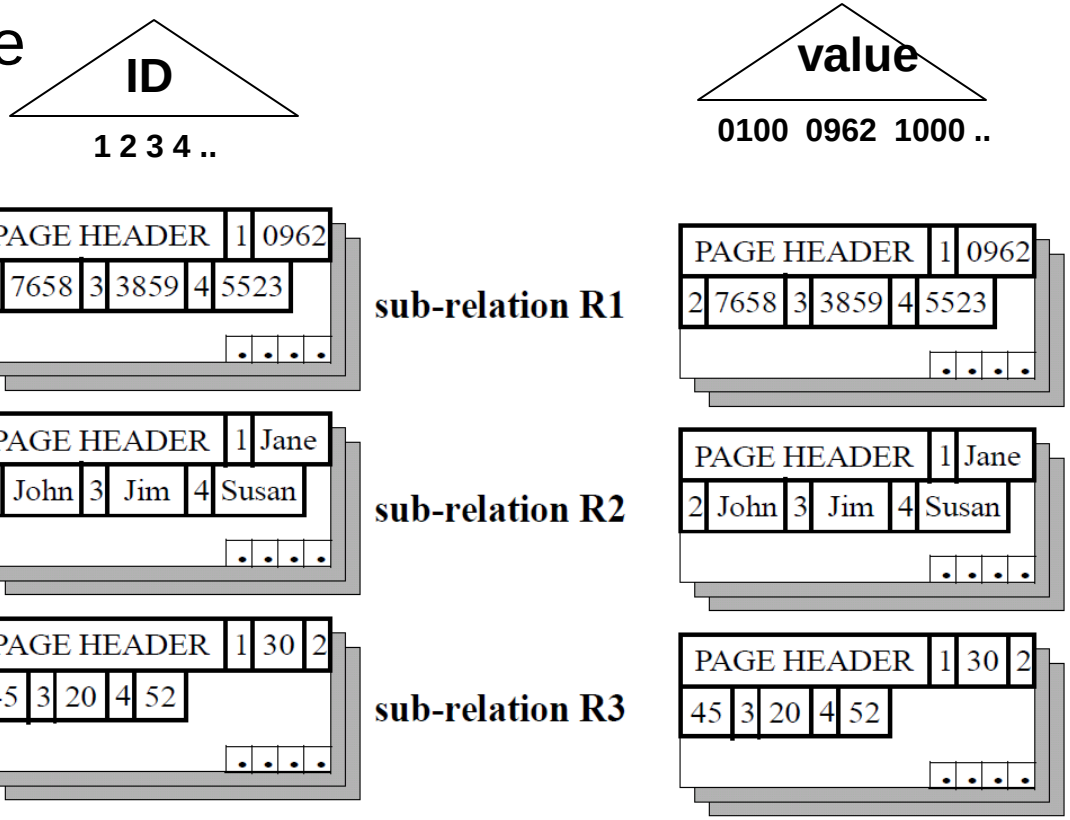
MIT: C-Store CWI: X100➔VectorWise 10+ startups

"A decomposition storage model"
Copeland and Khoshafian. SIGMOD 1985.

The original DSM paper

- Proposed as an alternative to NSM
- 2 indexes: clustered on ID, non-clustered on value
- Speeds up queries projecting few columns
- Requires more storage





Memory wall and PAX

- 90s: Cache-conscious research

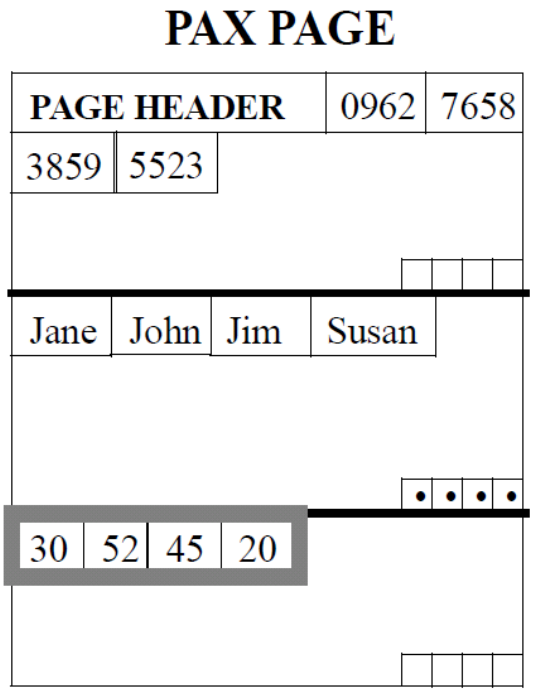
from: "Cache Conscious Algorithms for Relational Query Processing." Shatdal, Kant, Naughton. VLDB 1994.

to: "Database Architecture Optimized for the New Bottleneck: Memory Access." Boncz, Manegold, Kersten. VLDB 1999.

and: "DBMSs on a modern processor: Where does time go?" Ailamaki, DeWitt, Hill, Wood. VLDB 1999.

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

"Weaving Relations for Cache Performance." Ailamaki, DeWitt, Hill, Skounakis, VLDB 2001.



More hybrid NSM/DSM schemes

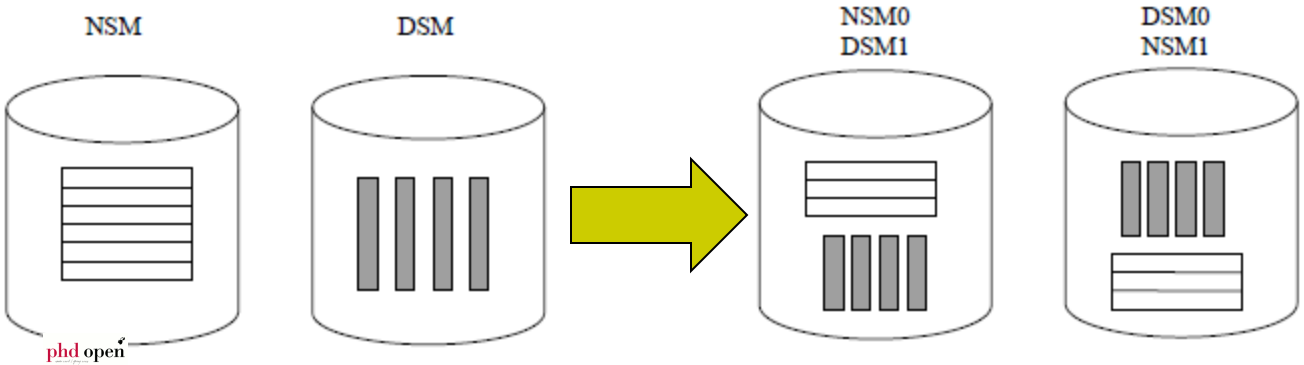
- Dynamic PAX: Data Morphing

"Data morphing: an adaptive, cache-conscious storage technique." Hankins, Patel, VLDB 2003.

- Clotho: custom layout using scatter-gather I/O

"Clotho: Decoupling Memory Page Layout from Storage Organization." Shao, Schindler, Schlosser, Ailamaki, and Ganger. VLDB 2004.

- Fractured mirrors
  - Smart mirroring with both NSM/DSM copies

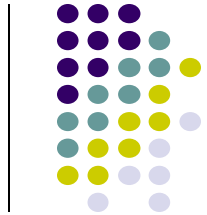


"A Case For Fractured Mirrors." Ramamurthy, DeWitt, Su, VLDB 2002.

MonetDB (more in Part 3)

- Late 1990s, CWI: Boncz, Manegold, and Kersten
- Motivation:
  - Main-memory
  - Improve computational efficiency by avoiding expression interpreter
  - DSM with virtual IDs natural choice
  - Developed new query execution algebra
- Initial contributions:
  - Pointed out memory-wall in DBMSs
  - Cache-conscious projections and joins
  - …

2005: the (re)birth of column-stores



- New hardware and application realities
  - Faster CPUs, larger memories, disk bandwidth limit
  - Multi-terabyte Data Warehouses
- New approach: combine several techniques
  - Read-optimized, fast multi-column access, disk/CPU efficiency, light-weight compression
- C-store paper:
  - First comprehensive design description of a column-store
- MonetDB/X100
  - “proper” disk-based column store
- Explosion of new products

Performance tradeoffs: columns vs. rows

DSM traditionally was not favored by technology trends
How has this changed?

- Optimized DSM in “Fractured Mirrors,” 2002
- “Apples-to-apples” comparison

“Performance Tradeoffs in Read-Optimized Databases” Harizopoulos, Liang, Abadi, Madden, VLDB’06

- Follow-up study

“Read-Optimized Databases, In-Depth” Holloway, DeWitt, VLDB’08

- Main-memory DSM vs. NSM

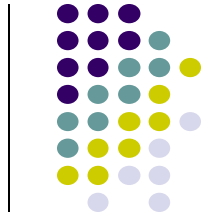
“DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing” Boncz, Zukowski, Nes, DaMoN’08

- Flash-disks: a come-back for PAX?

“Fast Scans and Joins Using Flash Drives” Shah, Harizopoulos, Wiener, Graefe. DaMoN’08

“Query Processing Techniques for Solid State Drives” Tsirogiannis, Harizopoulos, Shah, Wiener, Graefe, SIGMOD’09

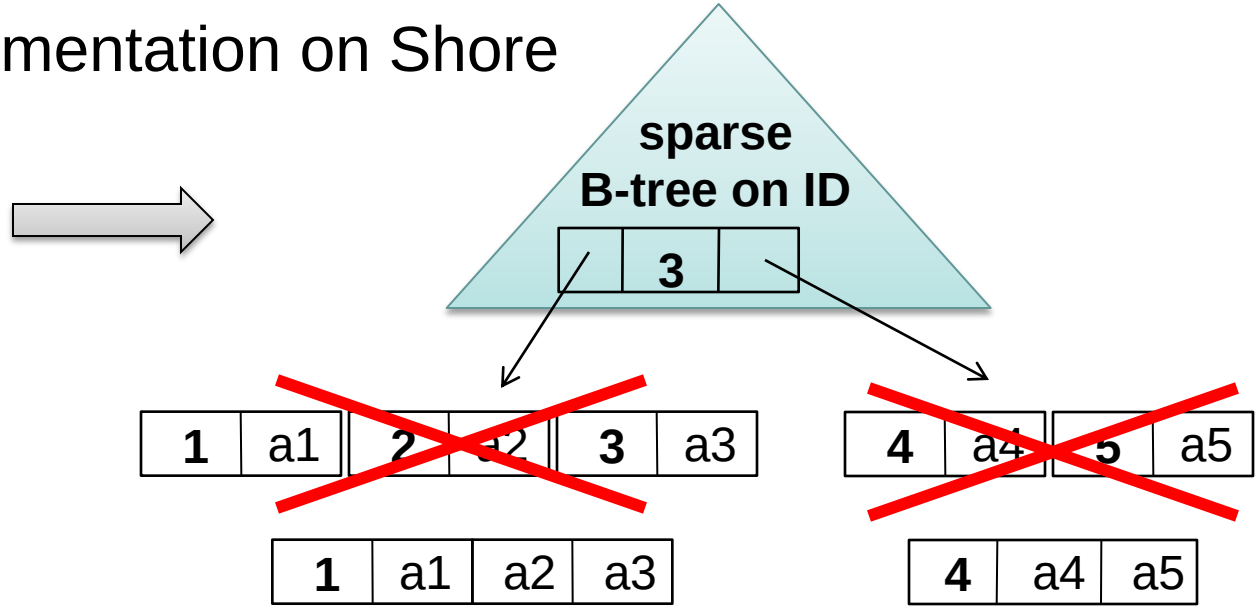
Fractured mirrors: a closer look



- Store DSM relations inside a B-tree
  - Leaf nodes contain values
  - Eliminate IDs, amortize header overhead
  - Custom implementation on Shore

"A Case For Fractured Mirrors" Ramamurthy, DeWitt, Su, VLDB 2002.

Tuple Header	TID	Column Data
	1	a1
	2	a2
	3	a3
	4	a4
	5	a5

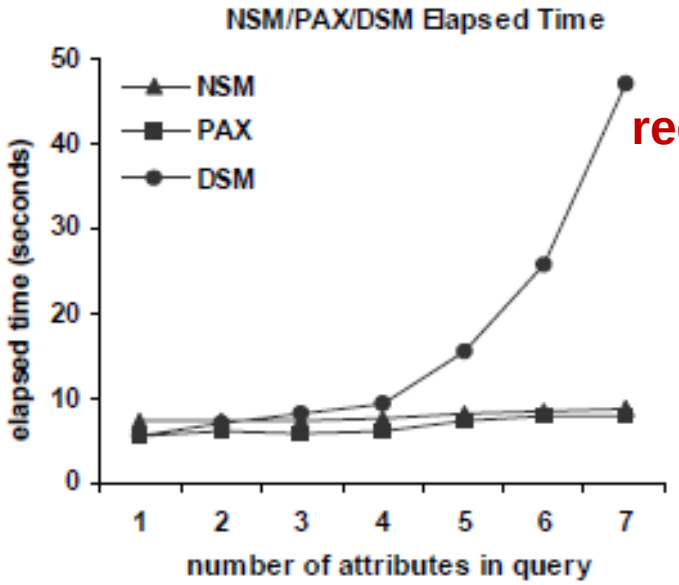


Similar: storage density comparable to column stores

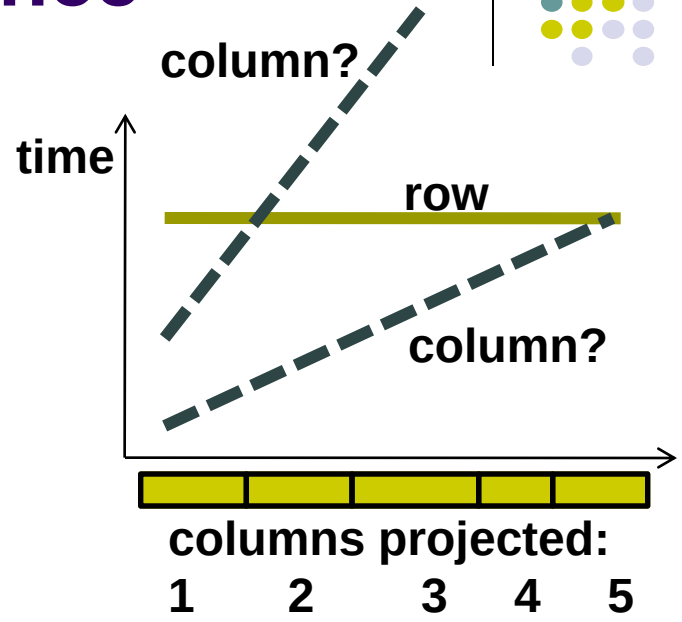
"Efficient columnar storage in B-trees" Graefe. Sigmod Record 03/2007.

Fractured mirrors: performance

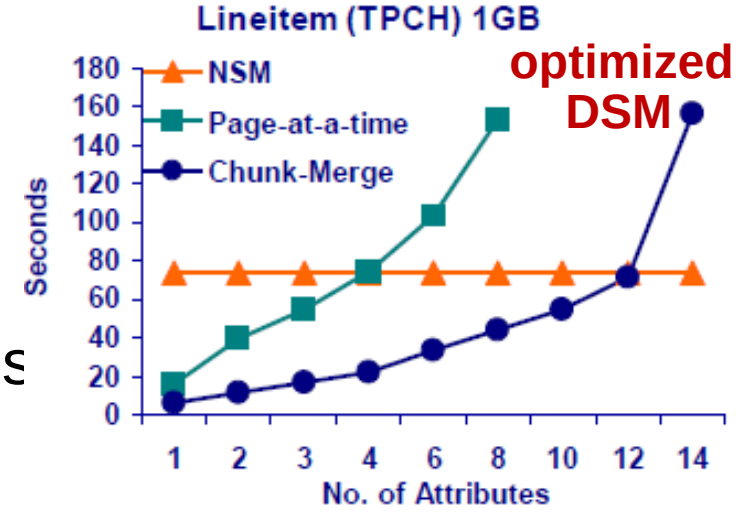
From PAX paper:



regular DSM



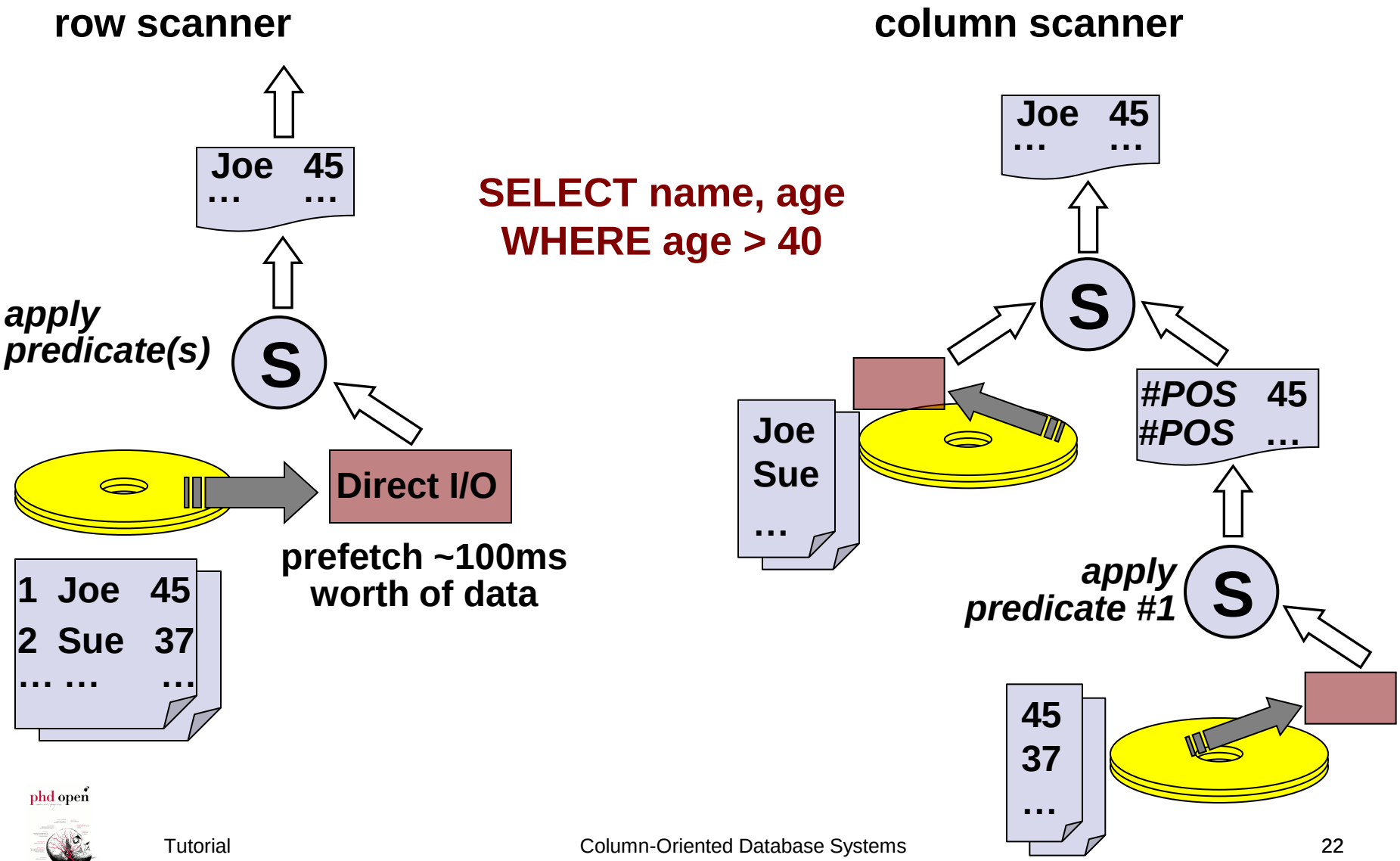
- Chunk-based tuple merging
  - Read in segments of M pages
  - Merge segments in memory
  - Becomes CPU-bound after 5 pages



optimized DSM

Column-scanner implementation

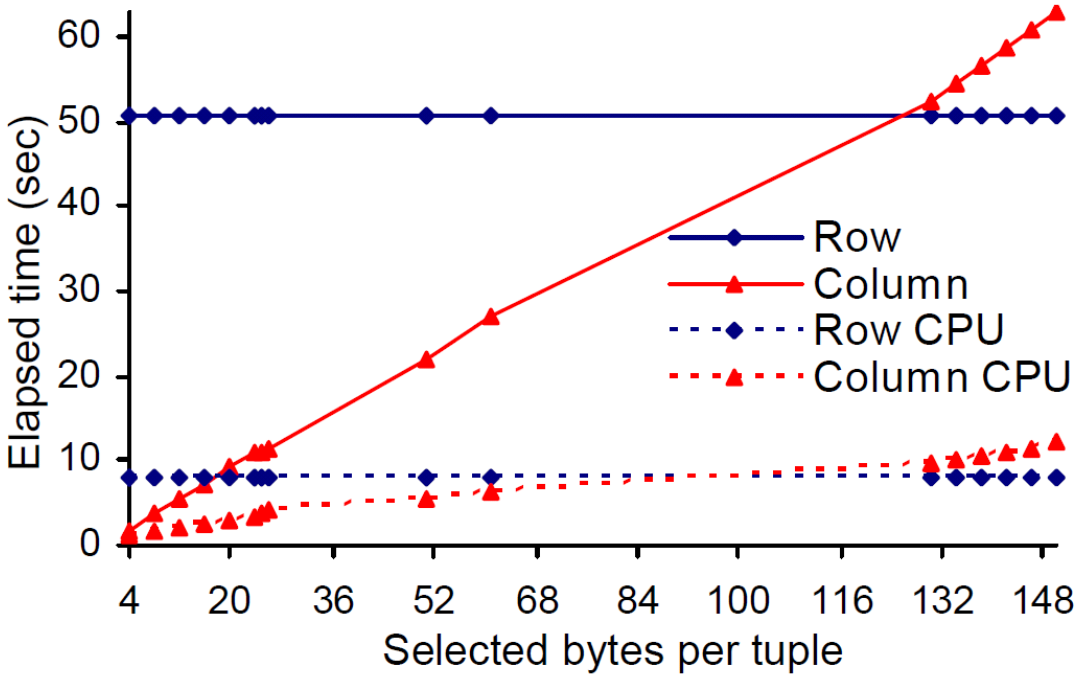
"Performance Tradeoffs in Read-Optimized Databases" Harizopoulos, Liang, Abadi, Madden, VLDB'06



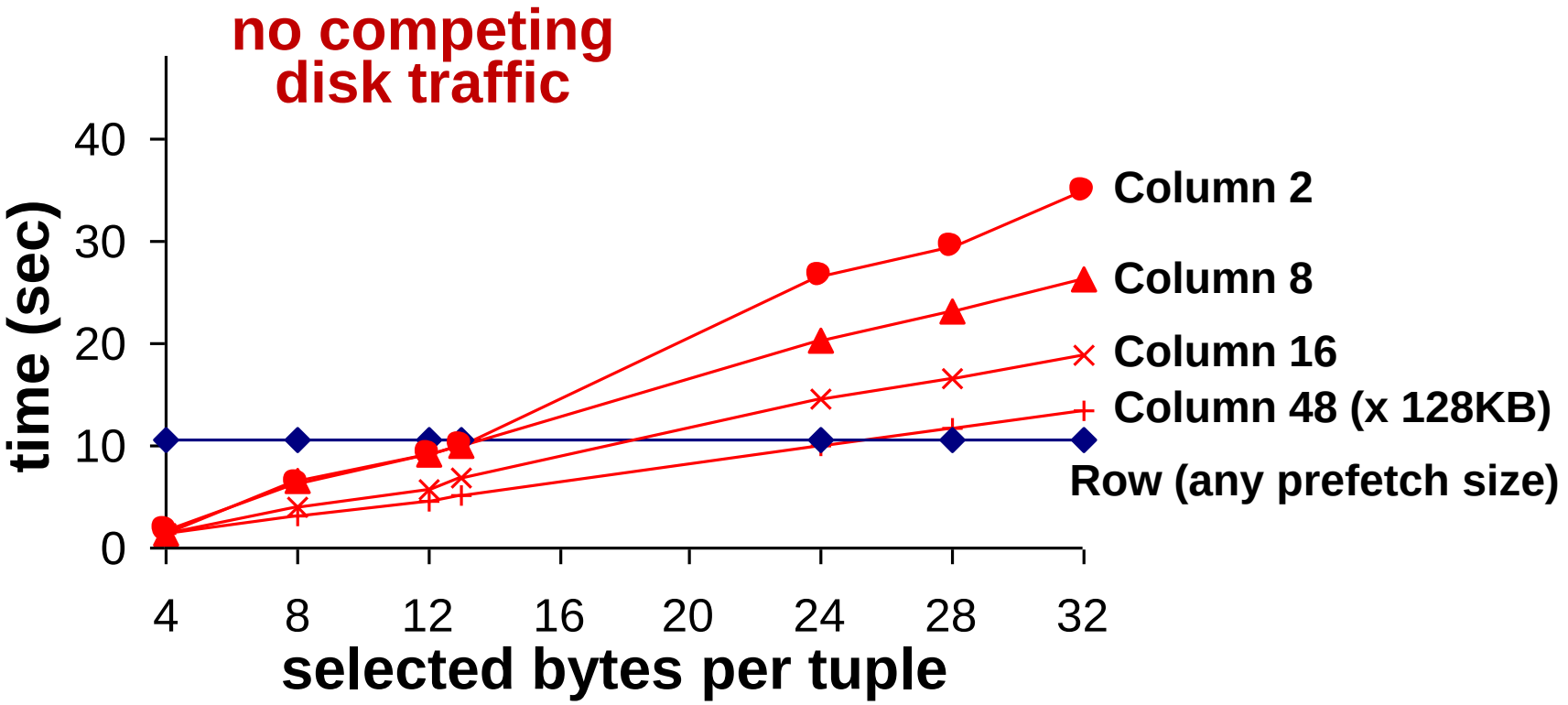
Scan performance

- Large prefetch hides disk seeks in columns
- Column-CPU efficiency with lower selectivity
- Row-CPU suffers from memory stalls
- Memory stalls disappear in narrow tuples
- Compression: similar to narrow

not shown, details in the paper

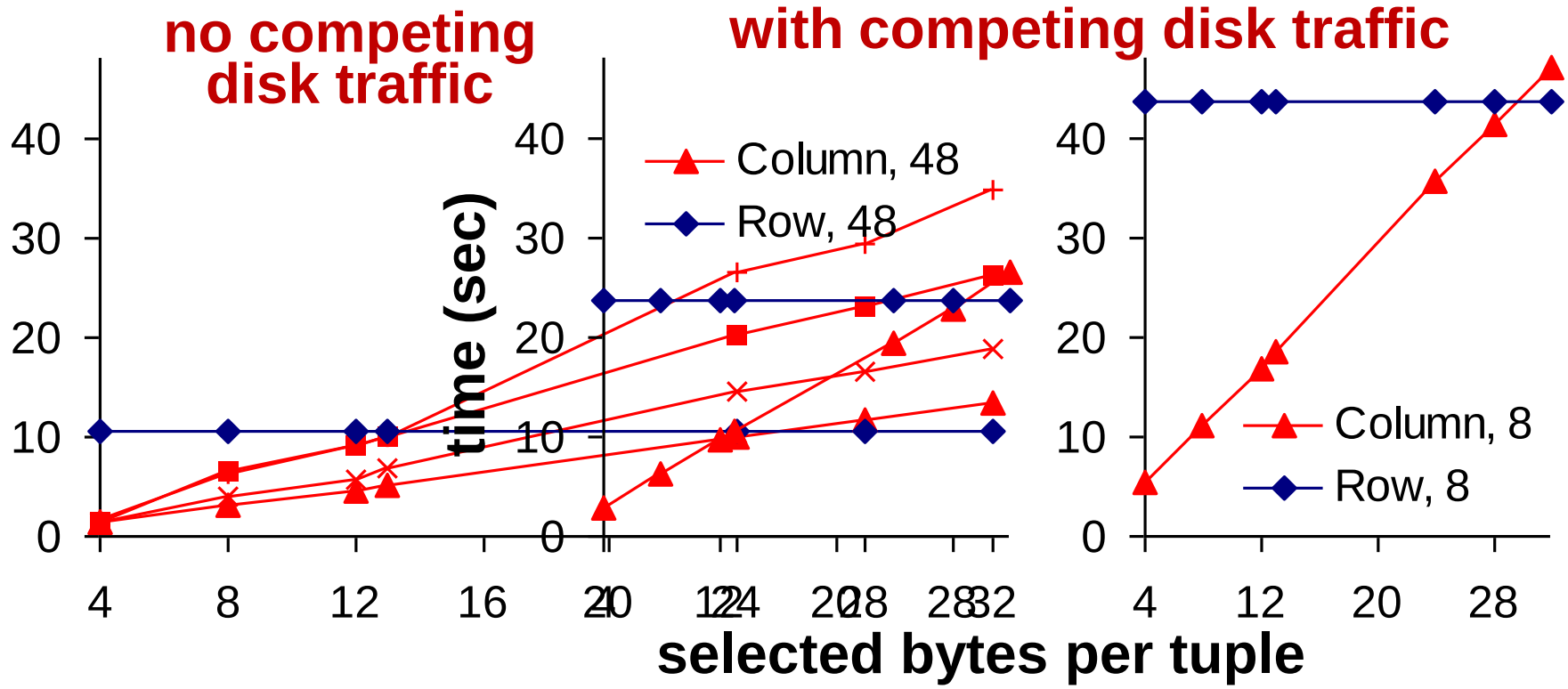


Varying prefetch size



- No prefetching hurts columns in single scans

Varying prefetch size



- No prefetching hurts columns in single scans
- Under competing traffic, columns outperform rows for any prefetch size

CPU Performance

"DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing" Boncz, Zukowski, Nes, DaMoN'08

- Playing field: in-flight tuples, regardless input storage
  - How is data stored in temporary in-memory buffers?
- Idea: on-the-fly conversion between NSM and DSM
  - Insert data storage conversion operators

Convert the entire in-flight tuple, or part of it

- Effectively hybrid layout
- Should be planned by query optimizer

  "data layout planning"

Winners:

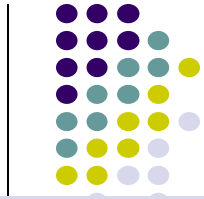
- DSM: sequential access (block fits in L2), random in L1
- NSM: random access, SIMD for grouped Aggregation

New storage technology: Flash SSDs

- Performance characteristics
  - very fast random reads, slow random writes
  - fast sequential reads and writes
- Price per bit (capacity follows)
  - cheaper than RAM, order of magnitude more expensive than Disk
- Flash Translation Layer introduces unpredictability
  - avoid random writes!
- Form factors not ideal yet
  - SSD (➔ small reads still suffer from SATA overhead/OS limitations)
  - PCI card (➔ high price, limited expandability)

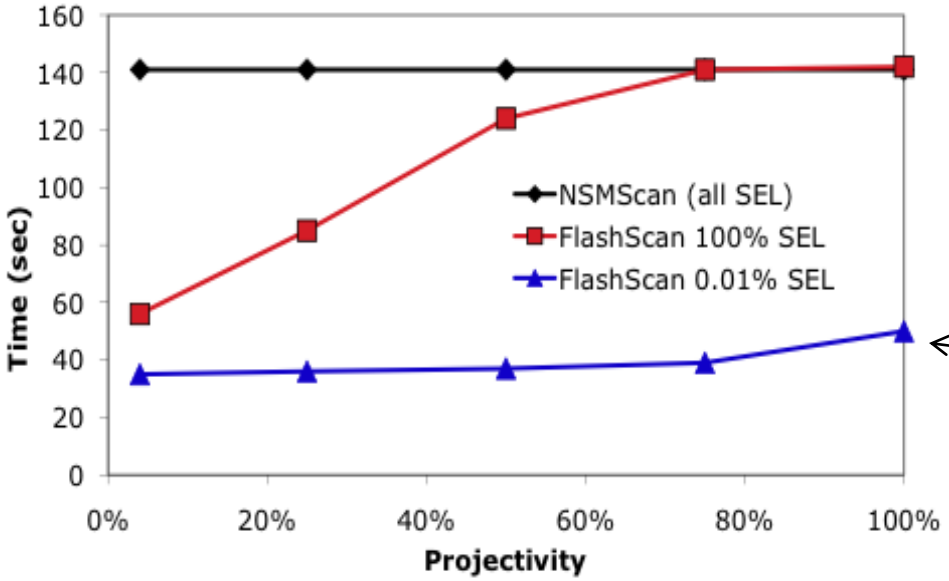
- Boost Sequential I/O in a simple package
  - Flash RAID: very tight bandwidth/cm^3 packing (4GB/sec inside the box)
- Column Store Updates
  - useful for delta structures and logs
- Random I/O on flash fixes unclustered index access
  - still suboptimal if I/O block size > record size
  - therefore column stores profit mush less than horizontal stores
- Random I/O useful to exploit secondary, tertiary table orderings
  - the larger the data, the deeper clustering one can exploit

Even faster column scans on flash SSDs



30K Read IOps, 3K Write Iops
250MB/s Read BW, 200MB/s Write

- New-generation SSDs
  - Very fast random reads, slower random writes
  - Fast sequential RW, comparable to HDD arrays
- No expensive seeks across columns
- FlashScan and Flashjoin: PAX on SSDs, inside Postgres

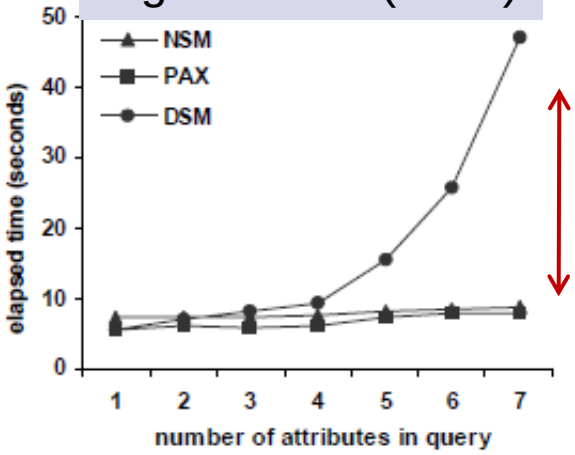


"Query Processing Techniques for Solid State Drives" Tsirogiannis, Harizopoulos, Shah, Wiener, Graefe, SIGMOD'09

mini-pages with no qualified attributes are not accessed

Column-scan performance over time

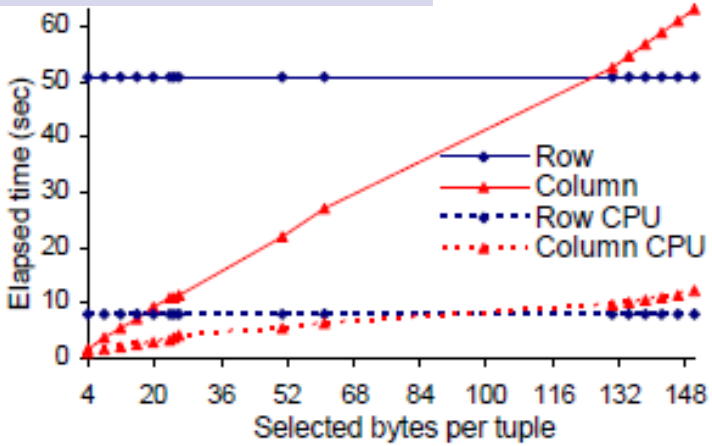
regular DSM (2001)



from 7x slower

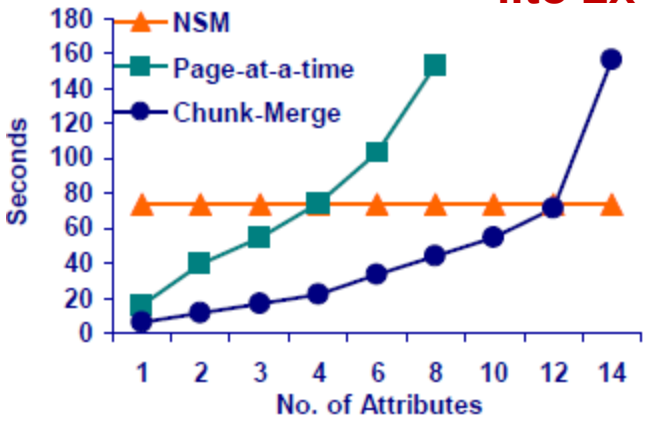
column-store (2006)

..to 1.2x slower



Lineitem (TPCH) 1GB

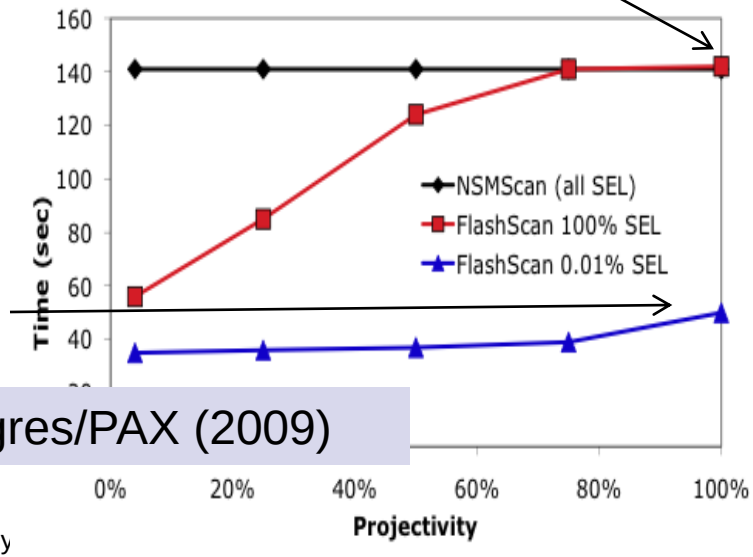
..to 2x slower



optimized DSM (2002)

..to same

and 3x faster!



SSD Postgres/PAX (2009)

Outline

- Part 1: Basic concepts
  - Introduction to key features
  - From DSM to column-stores and performance tradeoffs
  - Column-store architecture overview
  - Will rows and columns ever converge?
- Part 2: Column-oriented execution
- Part 3: MonetDB/X100 (“VectorWise”) and CPU efficiency

Architecture of a column-store

storage layout

- read-optimized: dense-packed, compressed
- organize in extends, batch updates
- multiple sort orders
- sparse indexes

engine

- **block-tuple operators**
- new access methods
- **work on compressed data**
- optimized relational operators

system-level

- system-wide column support
- loading / **updates**
- scaling through multiple nodes
- **transactions** / redundancy

Continuous Load and Query (Vertica)

Hybrid Storage Architecture

Trickle Load

> Write Optimized Store (WOS)

A B C

- Memory based
- Unsorted / Uncompressed
- Segmented
- Low latency / Small quick inserts

TUPLE MOVER
Asynchronous Data Transfer

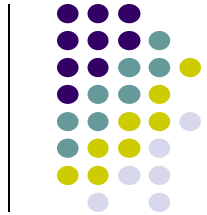
> Read Optimized Store (ROS)

- On disk
- Sorted / Compressed
- Segmented
- Large data loaded direct

A B C

(A B C | A)

Differential Updates in Column Stores

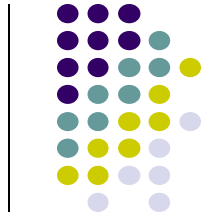


- Problem Definition:
  - A table T is kept ***ordered*** on a sortkey SK
    - table is stored column-wise, compressed
  - Keep updates (INS,MOD,DEL) in a differential structure
    - “write store”, in C-Store/Vertica

- Questions:
  - What data structures/algorithms to use for updates?
  - How are read-only queries impacted?
    - Must **merge** differential structure with base table
  - Maximal update throughput that can be sustained?
    - Given constraints on
      - RAM for differences
      - IO bandwidth
      - per-modification memory cost

    ➔ Currently viable approach for many scenarios

Value-based Delta Tables (VDTs)



- How is the row-store organized?

- Simple approach:
  - INS(C1..Cn) table
    - Keeps full record
  - DEL(SK1..SKm)
    - Only stores keys of deleted tuples
  - Modifications are DEL+INS

SQL: T=inventory SK=[store,prod]

```
SELECT * FROM ins
 UNION (SELECT * FROM inventory WHERE NOT EXISTS
(SELECT * FROM del
 WHERE inventory.store = del.store AND
       inventory.prod = del.prod))
```

Physical Relational Algebra:

```
MergeUnion[store,prod](Scan(ins),
 MergeDiff[store,prod](Scan(inventory),Scan(del)))
```

Problem: overhead on queries

- Always a MergeUnion (and MergeDiff) ➔CPU cost
- Union/Diff need the SKs
  - Many queries do not need these by themselves
  - Now the column-store must do IO for them anyway ☹

Positional Updates

Remember the **position** of an update rather than its SK

- less CPU cost (especially in block-oriented processing)
  - Scan passes tuples through unmodified until the next modification position
- no need to scan SK columns

SID	store	prod	new	qty	RID
0	London	chair	N	30	0
1	London	stool	N	10	1
2	London	table	N	20	2
3	Paris	rug	N	1	3
4	Paris	stool	N	5	4

Figure 1: TABLE$_0$

the SK=(store,prod) so inserts go “in between”!!

```
INSERT INTO inventory
  VALUES ('Berlin','table','Y',10)
INSERT INTO inventory
  VALUES ('Berlin','cloth','Y',5)
INSERT INTO inventory
  VALUES ('Berlin','chair','Y',20)
```

Figure 2: BATCH$_1$

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	5	1
0	Berlin	table	Y	10	2
0	London	chair	N	30	3
1	London	stool	N	10	5
2	London	table	N	20	4
3	Paris	rug	N	1	6
4	Paris	stool	N	5	7

Figure 5: TABLE$_1$

TABLEx = state of TABLE at time x

SID(t)=StableID=position of tuple t in columns=RID$_0$(t) ← Stable

RIDx(t) = RowID of tuple t at time x ← HIGHLY VOLATILE!!

RIDs and SIDs

TABLEx = state of TABLE at time x

SID(t)=StableID=position of tuple t in columns=$RID_0(t)$ ← Stable

RIDx(t) = RowID of tuple t at time x ← HIGHLY VOLATILE!!

The difference between RID and SID:

$$\Delta_{t_2}^{t_1}(\tau) \quad = \mathrm{RID}(\tau)_{t_2} - \mathrm{RID}(\tau)_{t_1} \tag{2}$$

which in the common case when we have one PDT on top of the stable table ($t_1 = 0$) is RID minus SID:[3]

$$\Delta_t(\tau) \quad = \mathrm{RID}(\tau)_t - \mathrm{SID}(\tau) \tag{3}$$

If we define the SK-based Table time-wise difference as:

$$\mathrm{MINUS}_{t_2}^{t_1} \quad = \{\tau \in \mathrm{TABLE}_{t_1} : \nexists \gamma \in \mathrm{TABLE}_{t_2} : \tau.\mathrm{SK} = \gamma.\mathrm{SK}\} \tag{4}$$

then we can compute Δ as the number of inserts minus the number of deletes before τ:

$$\begin{aligned} \Delta_{t_2}^{t_1}(\tau) \quad = \quad & |\{\gamma \in \mathrm{MINUS}_{t_1}^{t_2} : \mathrm{RID}(\gamma)_{t_2} < \mathrm{RID}(\tau)_{t_2}\}| - \\ & |\{\gamma \in \mathrm{MINUS}_{t_2}^{t_1} : \mathrm{SID}(\gamma) < \mathrm{SID}(\tau)\}| \end{aligned} \tag{5}$$

Enter Positional Delta Trees

- A counting B-Tree that keeps track of cumulative tuple shifts
- DIFF^ta_tb maintains a monotonic mapping between RID_x(ta) and RID_y(tb)
- DIFF = (PDT,VALS), VALS = value space
- Merge Operator ← used by each query to get up-todate table

$$\mathrm{TABLE}_{t_2} = \mathrm{TABLE}_{t_1}.\mathrm{Merge}(\mathrm{DIFF}_{t_2}^{t_1}) \qquad (1)$$

$$Aligned(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_a = t_c$$

$$Consecutive(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_b = t_c$$

$$Overlapping(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_a < t_d \leq t_b \vee t_c < t_b \leq t_d$$

Enter Positional Delta Trees

- A counting B-Tree that keeps track of cumulative tuple shifts
- DIFF^ta_tb maintains a monotonic mapping between RID_x(ta) and RID_y(tb)
- DIFF = (PDT,VALS), VALS = value space
- Merge Operator ⇚ used by each query to get up-to-date table

SID	store	prod	new	qty	RID
0	London	chair	N	30	0
1	London	stool	N	10	1
2	London	table	N	20	2
3	Paris	rug	N	1	3
4	Paris	stool	N	5	4

Figure 1: $TABLE_0$

```
INSERT INTO inventory
  VALUES ('Berlin','table','Y',10)
INSERT INTO inventory
  VALUES ('Berlin','cloth','Y',5)
INSERT INTO inventory
  VALUES ('Berlin','chair','Y',20)
```

Figure 2: $BATCH_1$

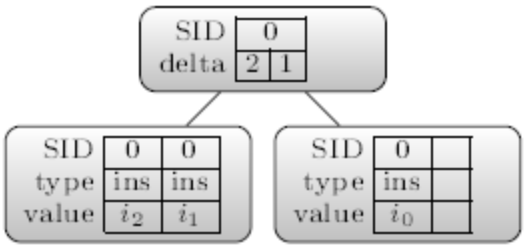


Figure 3: PDT_1

ins	store	prod	new	qty
i_0	Berlin	table	Y	10
i_1	Berlin	cloth	Y	5
i_2	Berlin	chair	Y	20

del store prod

new new qty qty

Figure 4: $VALS_1$

Enter Positional Delta Trees

- A counting B-Tree that keeps track of cumulative tuple shifts
- DIFF^ta_tb maintains a monotonic mapping between RID_x(ta) and RID_y(tb)
- DIFF = (PDT,VALS), VALS = value space
- Merge Operator ← used by each query to get up-to-date table

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	5	1
0	Berlin	table	Y	10	2
0	London	chair	N	30	3
1	London	stool	N	10	5
2	London	table	N	20	4
3	Paris	rug	N	1	6
4	Paris	stool	N	5	7

Figure 5: $TABLE_1$

```
UPDATE inventory SET qty=1
WHERE store='Berlin' and prod='cloth'
UPDATE inventory SET qty=9
WHERE store='London' and prod='stool'
DELETE FROM inventory
WHERE store='Berlin' and prod='table'
DELETE FROM inventory
WHERE store='Paris' and prod='rug'
```

Figure 6: $BATCH_2$

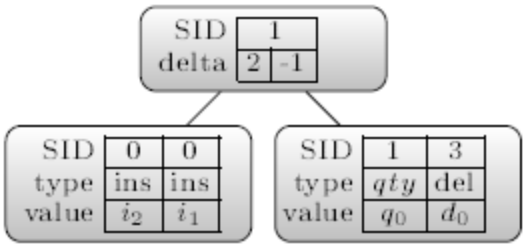


Figure 7: PDT_2

ins

	store	prod	new	qty
i_0				
i_1	Berlin	cloth	Y	1
i_2	Berlin	chair	Y	20

del

	store	prod
d_0	Paris	rug

new

new

qty

	qty
q_0	9

Figure 8: $VALS_2$

Enter Positional Delta Trees

- A counting B-Tree that keeps track of cumulative tuple shifts
- DIFF^ta_tb maintains a monotonic mapping between RID_x(ta) and RID_y(tb)
- DIFF = (PDT,VALS), VALS = value space
- Merge Operator ← used by each query to get up-to-date table

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	1	1
0	London	chair	N	30	2
1	London	stool	N	9	3
2	London	table	N	20	4
3	Paris	rug	N	1	5
4	Paris	stool	N	5	5

Figure 9: $TABLE_2$

```
INSERT INTO inventory
  VALUES ('Paris','rack','Y',4)
INSERT INTO inventory
  VALUES ('London','rack','Y',4)
INSERT INTO inventory
  VALUES ('Berlin','rack','Y',4)
```

Figure 10: $BATCH_3$

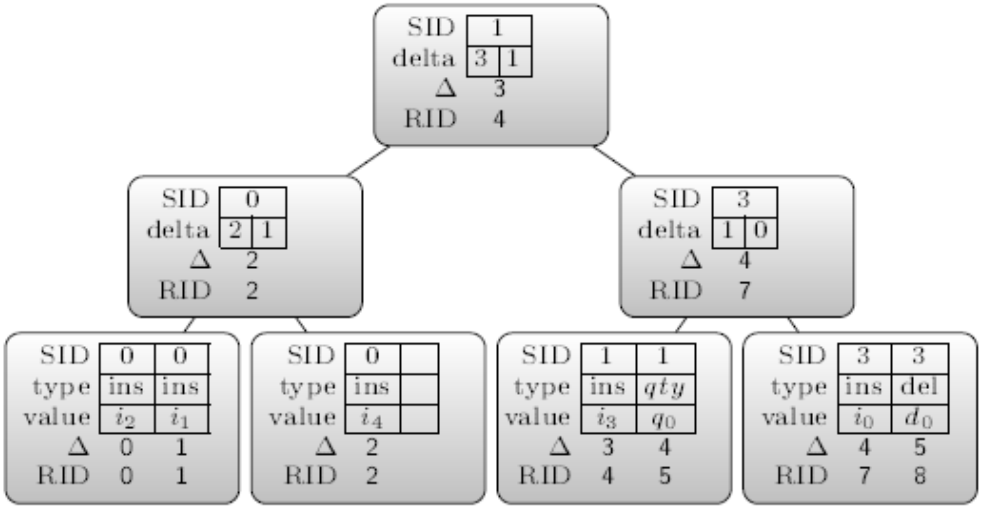
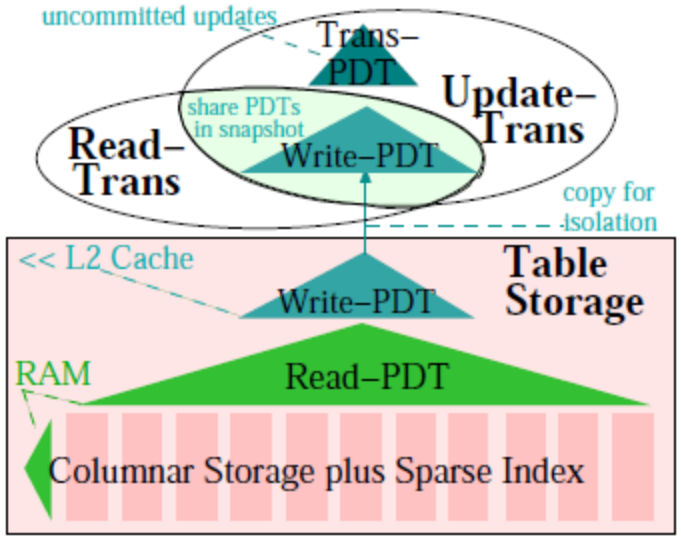


Figure 11: PDT_3 with annotated Δ,RID=SID+Δ

PDT-based Transactions

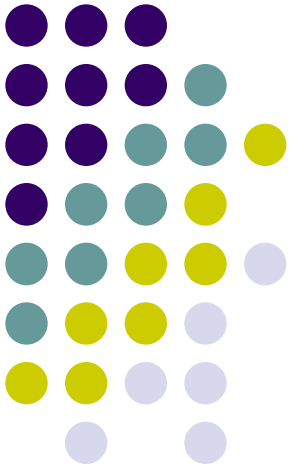


Snapshot Isolation with Layered PDTs

- Propagate(PDT^t2_t1,PDT^t3_t2)
  - Requires **Consecutive** PDTs
- Serialize(PDT^t2_t0,PDT^t3_t1) ➔ PDT^t3_t2
  - Makes **Overlapping** PDTs **Aligned**
  - **May Fail!!** ➔ transactional conflict detection

Column-Oriented Database Systems

Tutorial



Compression

"Super-Scalar RAM-CPU Cache Compression" Zukowski, Heman, Nes, Boncz, ICDE'06

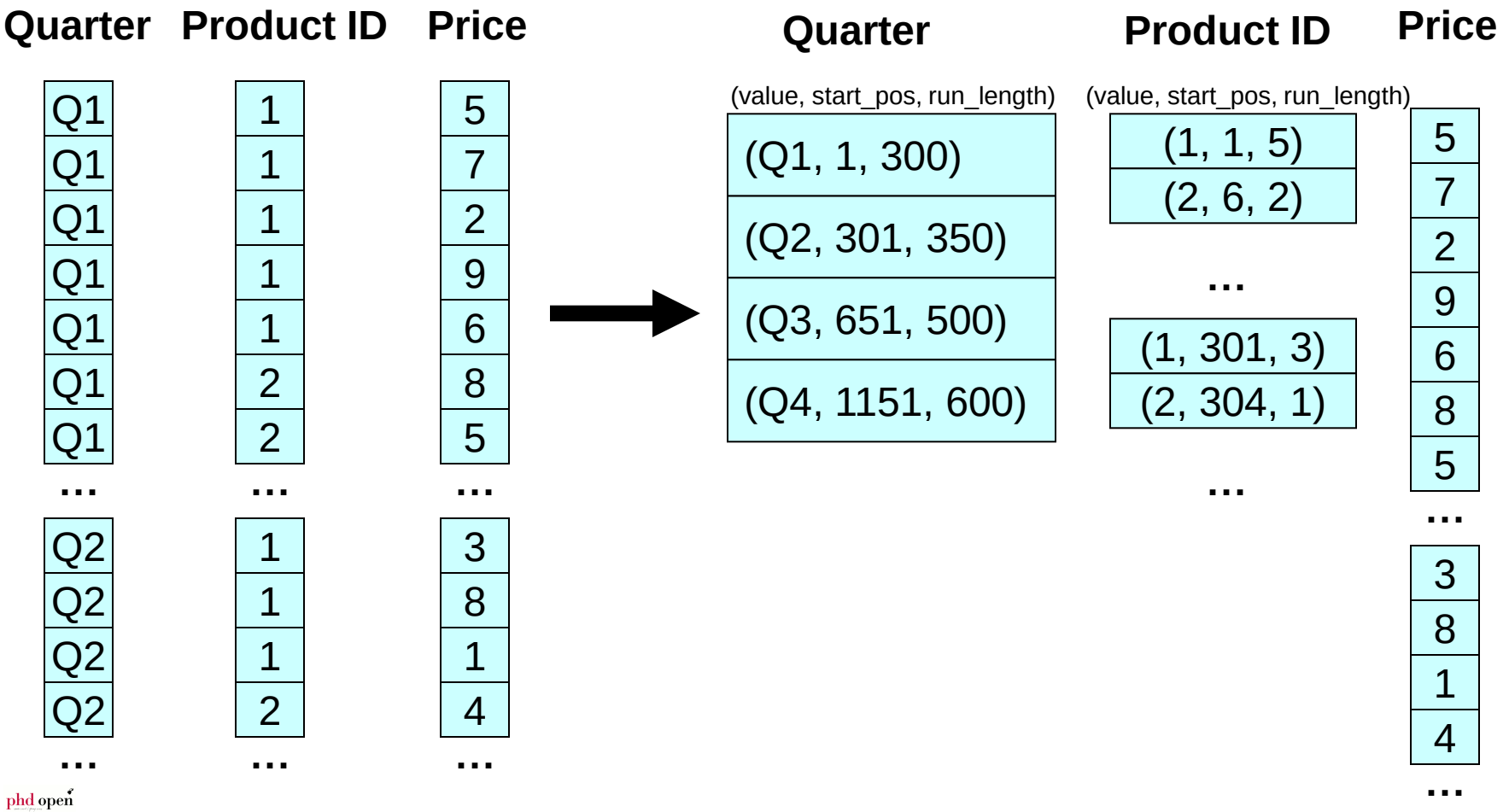
"Integrating Compression and Execution in Column-Oriented Database Systems" Abadi, Madden, and Ferreira, SIGMOD '06

•Query optimization in compressed database systems" Chen, Gehrke, Korn, SIGMOD'01

Compression

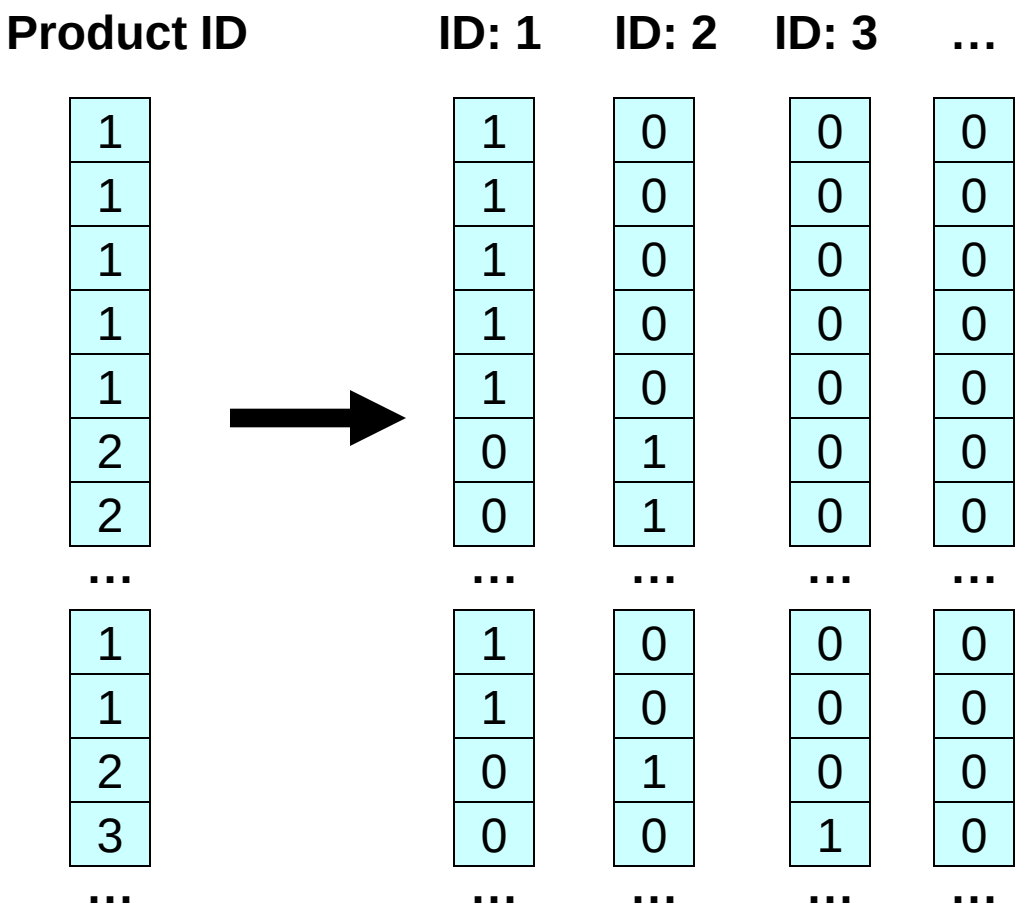
- Trades I/O for CPU
- Increased column-store opportunities:
  - Higher data value locality in column stores
  - Techniques such as run length encoding far more useful
  - Can use extra space to store multiple copies of data in different sort orders

Run-length Encoding



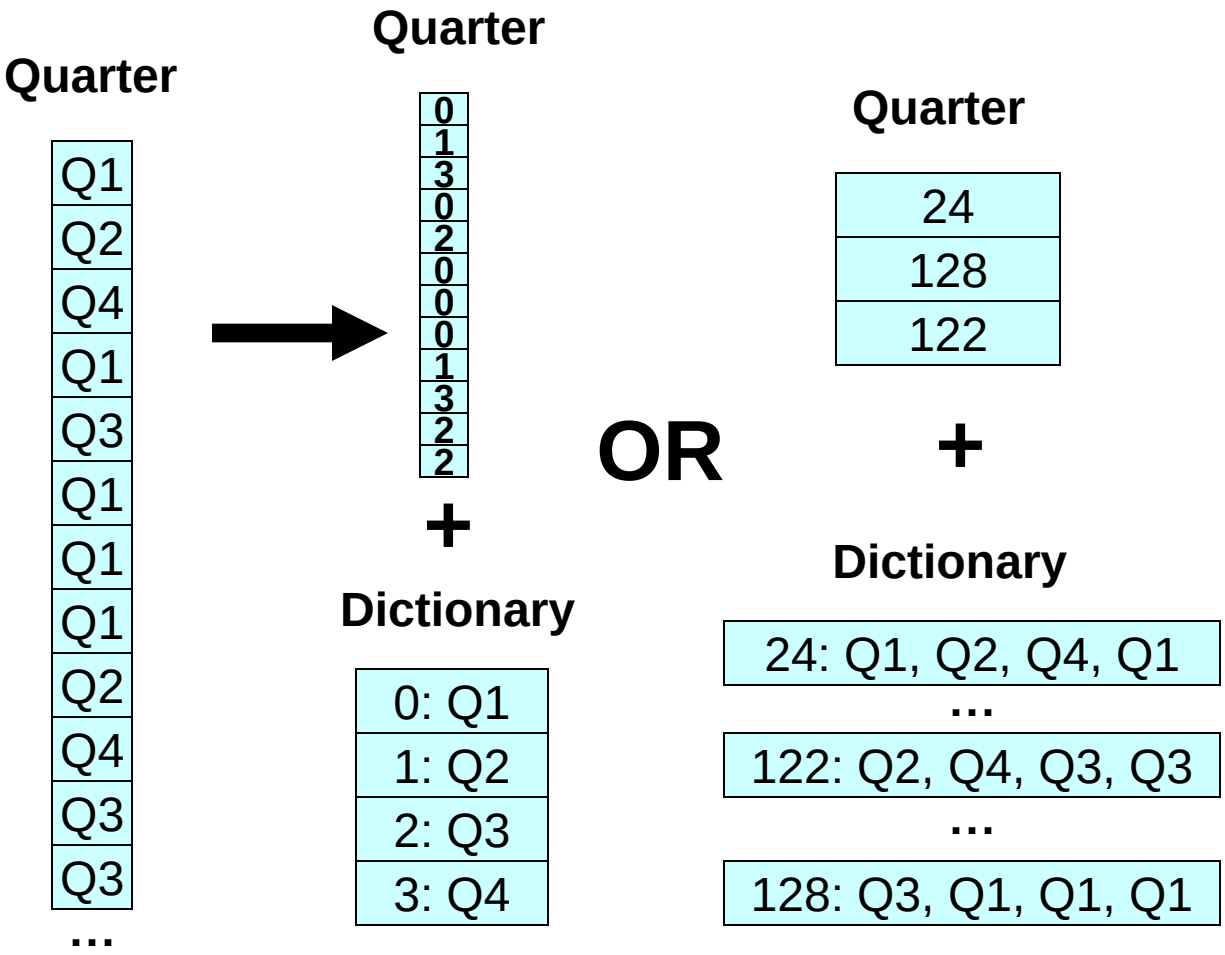
Bit-vector Encoding

- 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



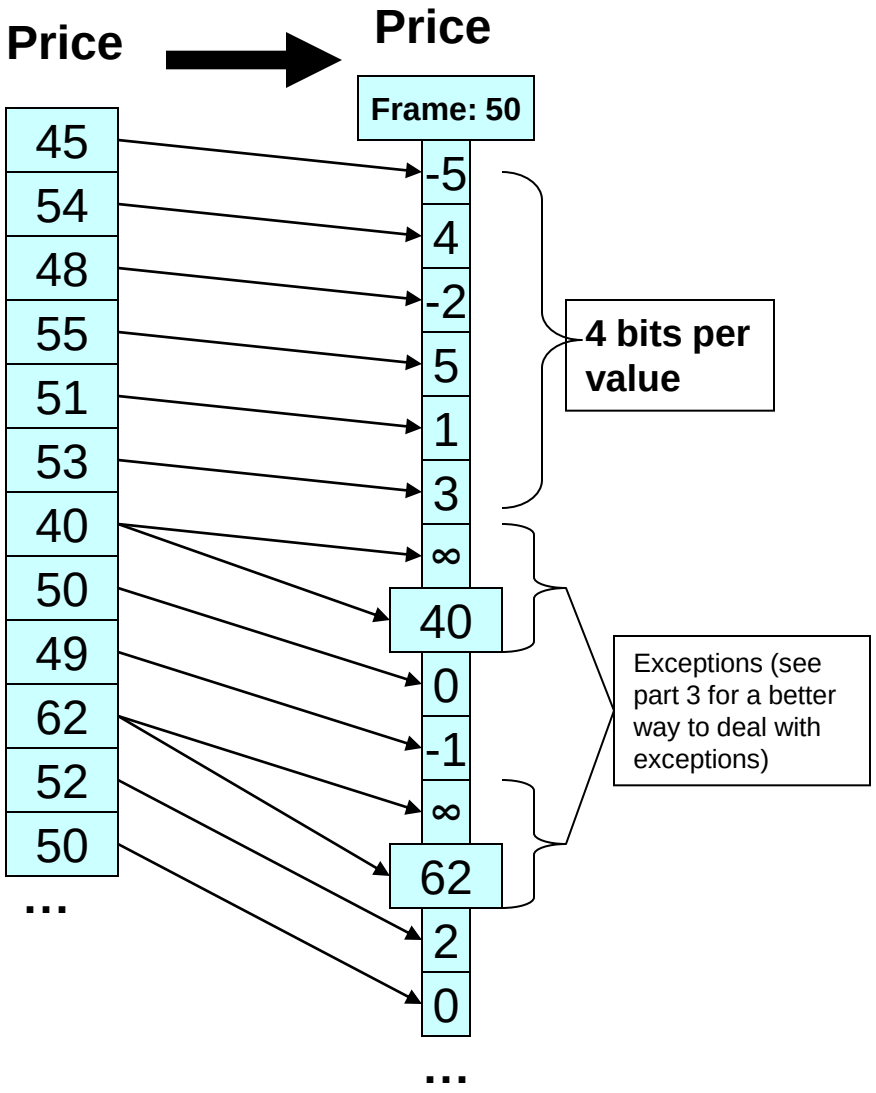
Dictionary Encoding

- 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

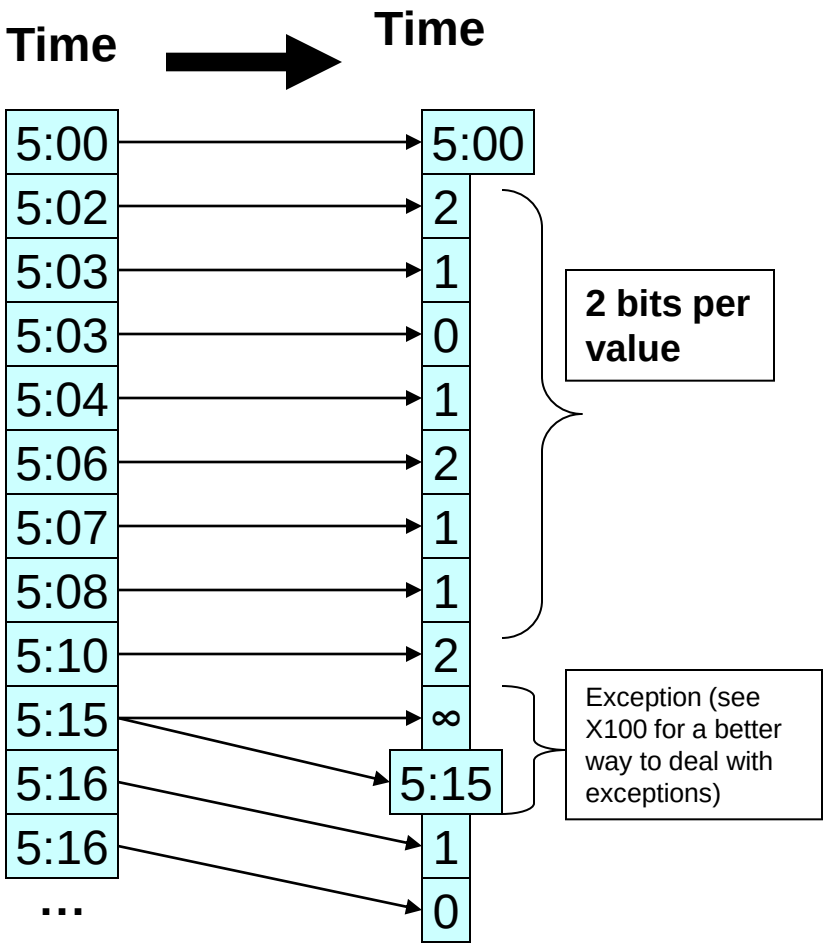


"Compressing Relations and Indexes" Goldstein, Ramakrishnan, Shaft, ICDE'98

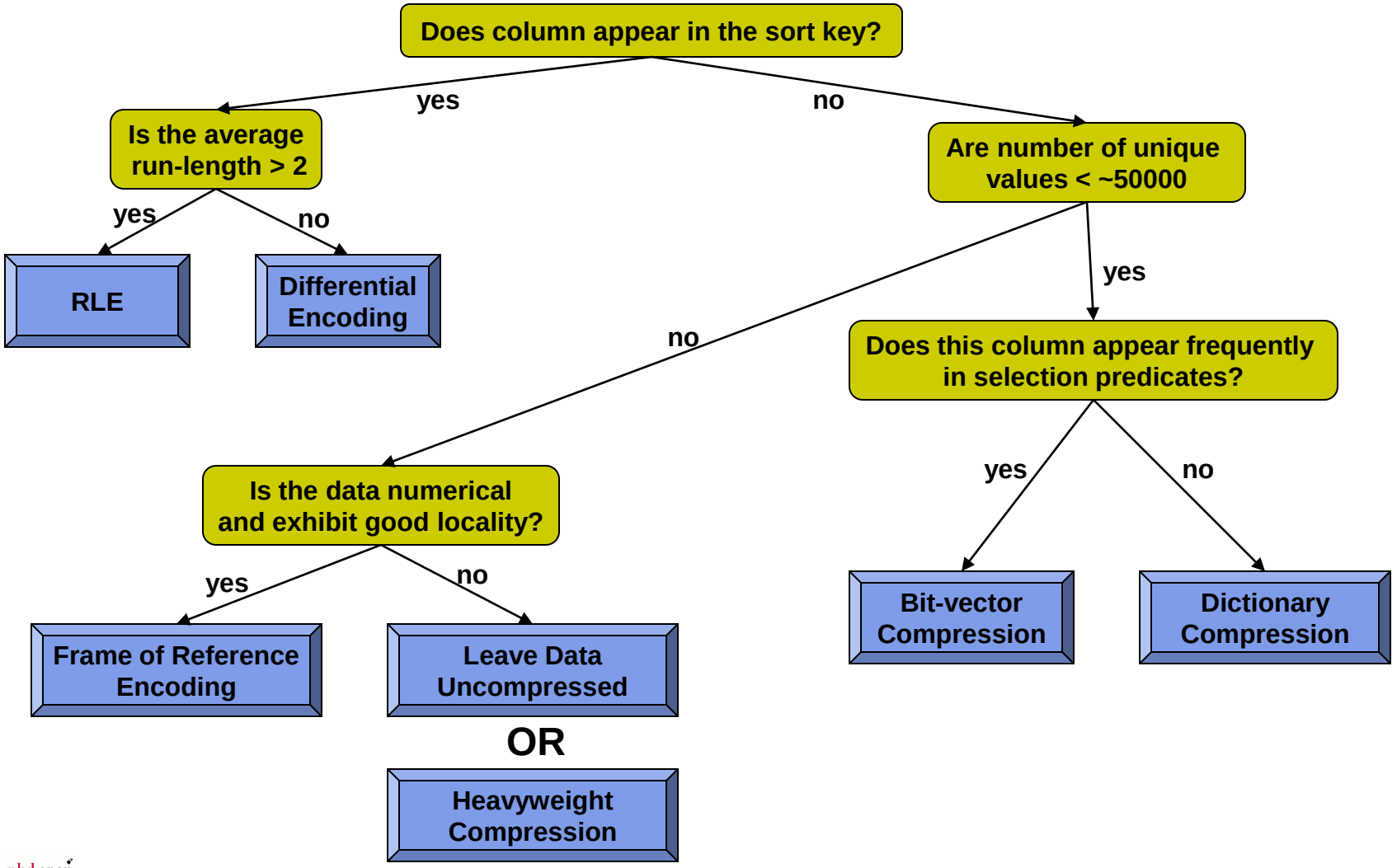
Differential Encoding

- 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
- Performs well on columns containing increasing/decreasing sequences
  - inverted lists
  - timestamps
  - object IDs
  - sorted / clustered columns

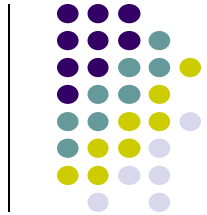
"Improved Word-Aligned Binary Compression for Text Indexing" Ahn, Moffat, TKDE'06



What Compression Scheme To Use?



"Super-Scalar RAM-CPU Cache Compression"
Zukowski, Heman, Nes, Boncz, ICDE'06



Heavy-Weight Compression Schemes

Algorithm	Decompression Bandwidth
BZIP	10 MB/s
ZLIB	80 MB/s
LZO	300 MB/s

- Modern disk arrays can achieve > 1GB/s
- 1/3 CPU for decompression ➔ 3GB/s needed

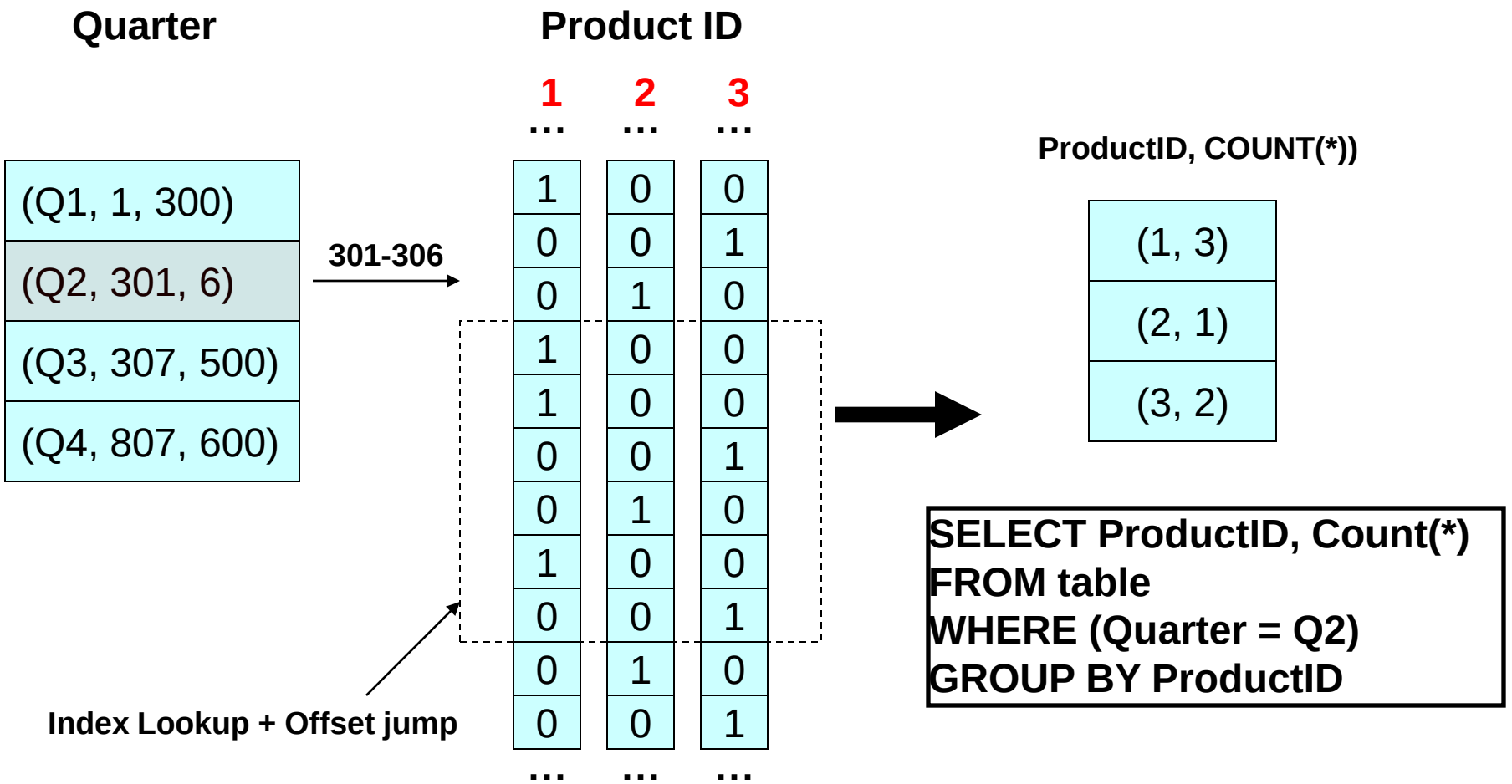
➔ **Lightweight compression schemes are better**

➔ **Even better: operate directly on compressed data**

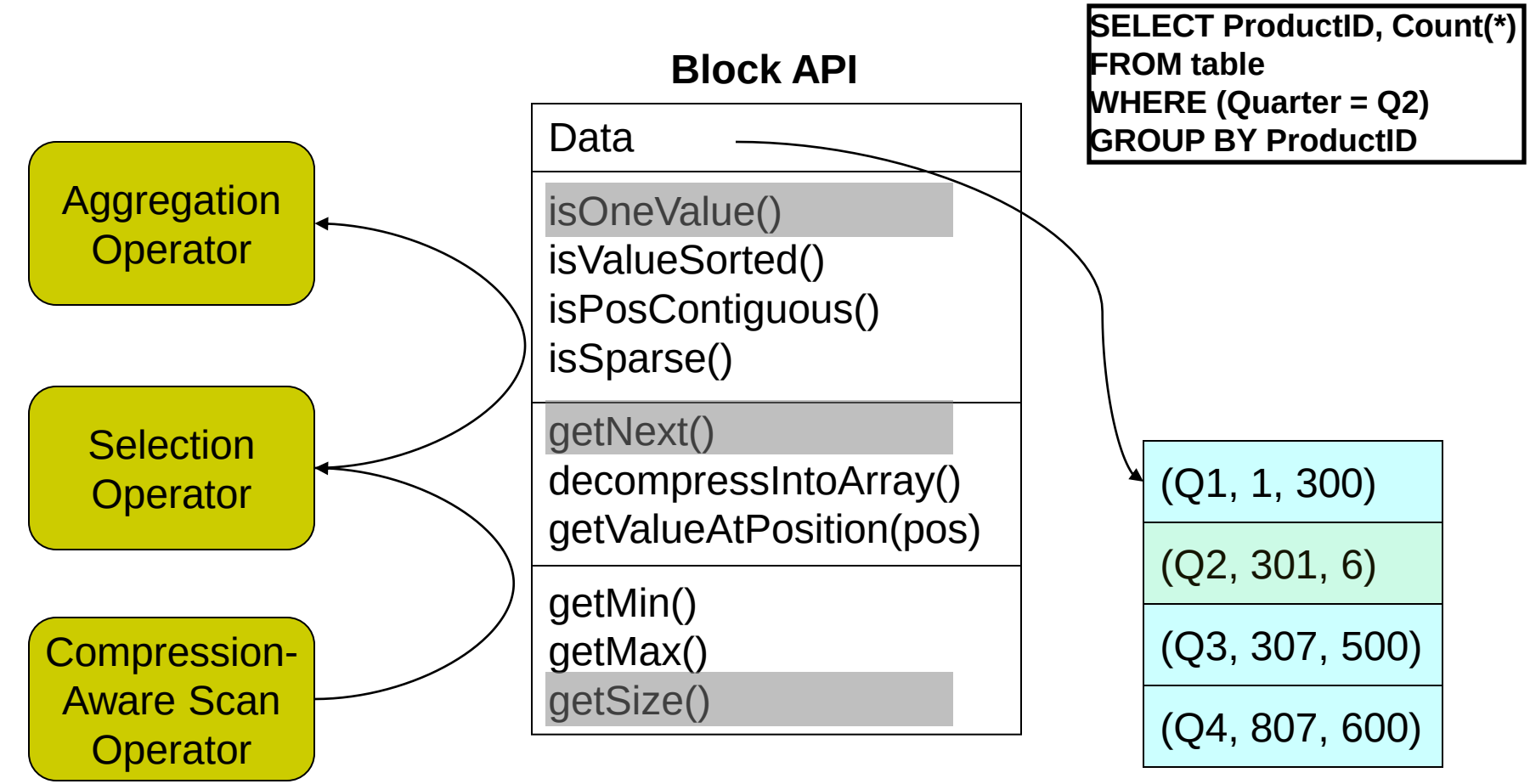
Operating Directly on Compressed Data

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

Operating Directly on Compressed Data



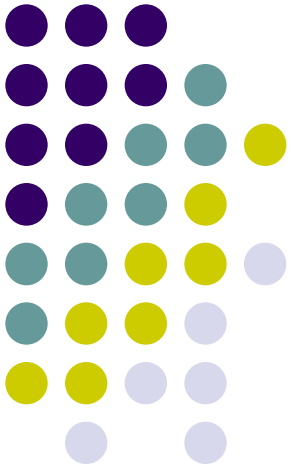
Operating Directly on Compressed Data



Column-Oriented Database Systems

Tutorial

Tuple Materialization and Column-Oriented Join Algorithms



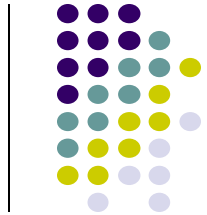
"Materialization Strategies in a Column-Oriented DBMS" Abadi, Myers, DeWitt, and Madden. ICDE 2007.

"Self-organizing tuple reconstruction in column-stores", Idreos, Manegold, Kersten, SIGMOD 2009

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Madden, and Hachem. SIGMOD 2008.

"Query Processing Techniques for Solid State Drives" Tsirogiannis, Harizopoulos Shah, Wiener, and Graefe. SIGMOD 2009.

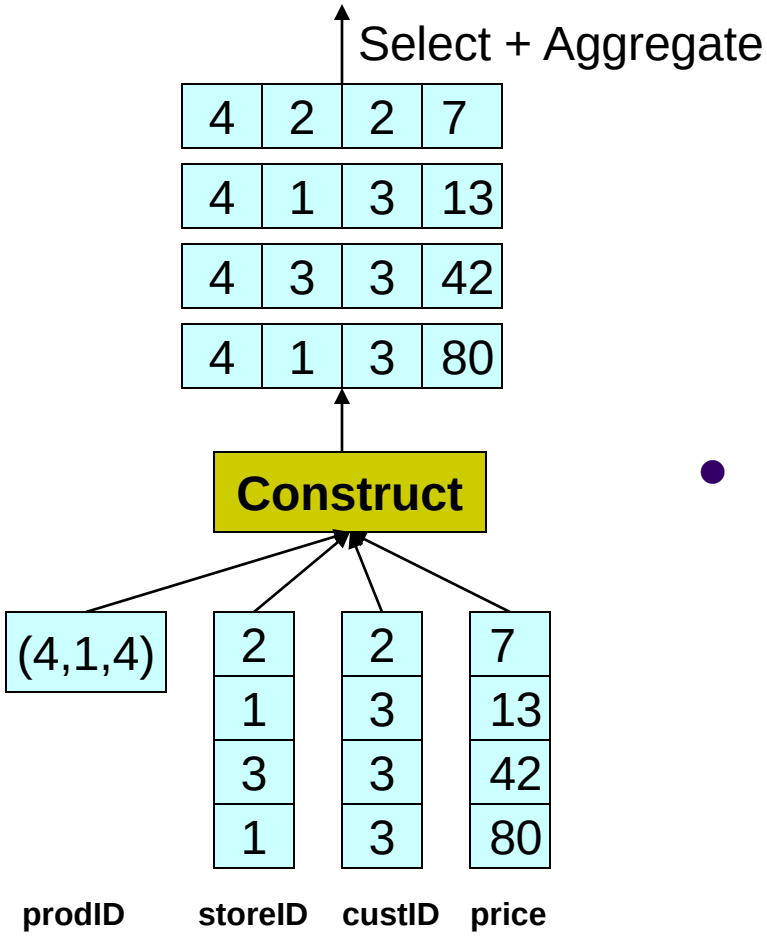
"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB 2004



When should columns be projected?

- Where should column projection operators be placed in a query plan?
  - 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
    - Most column-stores construct tuples and column projection time
      - Many database interfaces expect output in regular tuples (rows)
      - Rest of discussion will focus on this case

When should tuples be constructed?

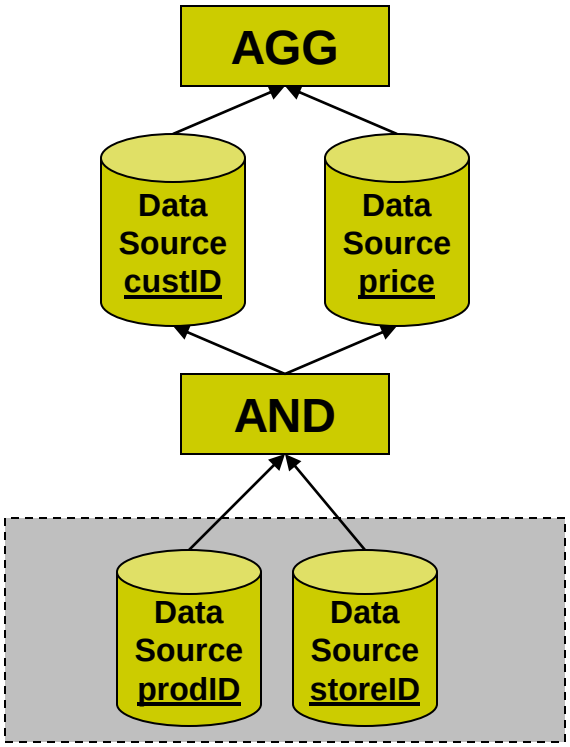


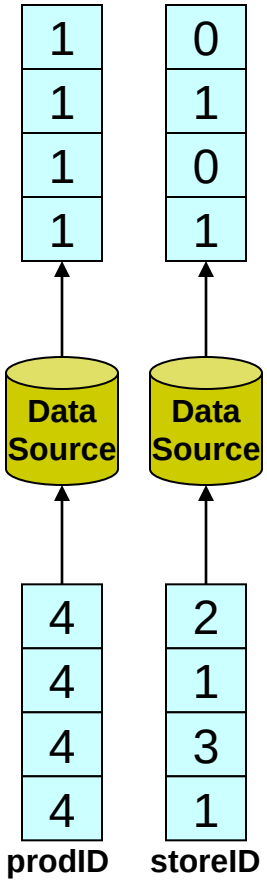
QUERY:

```
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



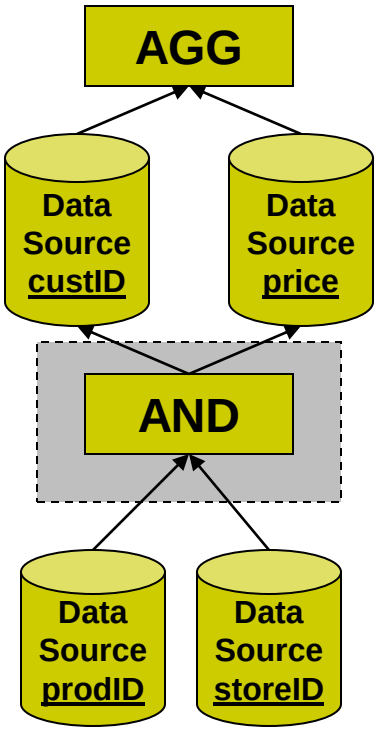


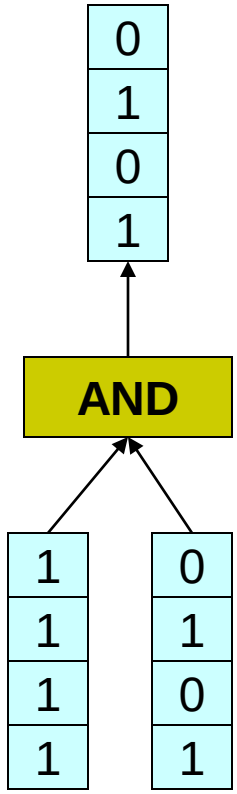
QUERY:

```
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



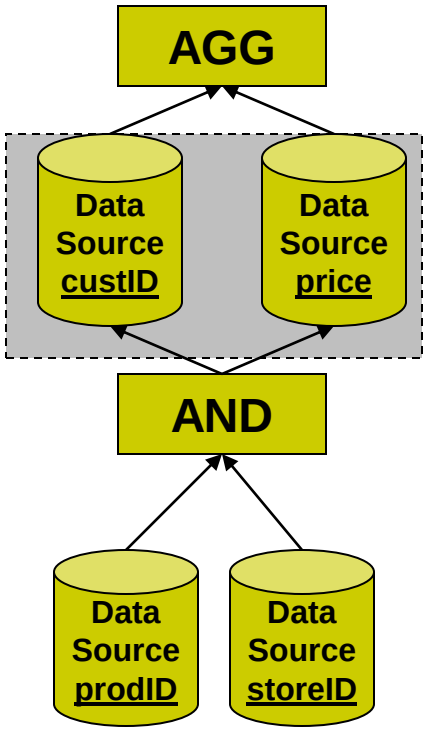


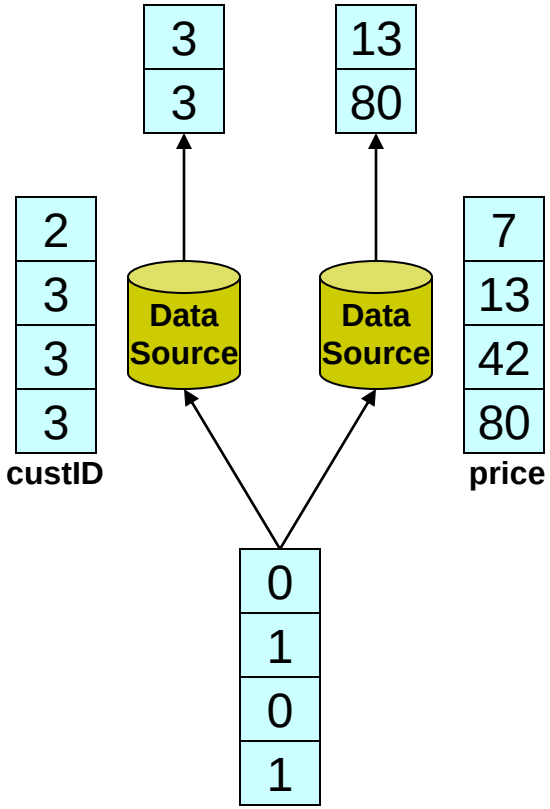
QUERY:

```
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



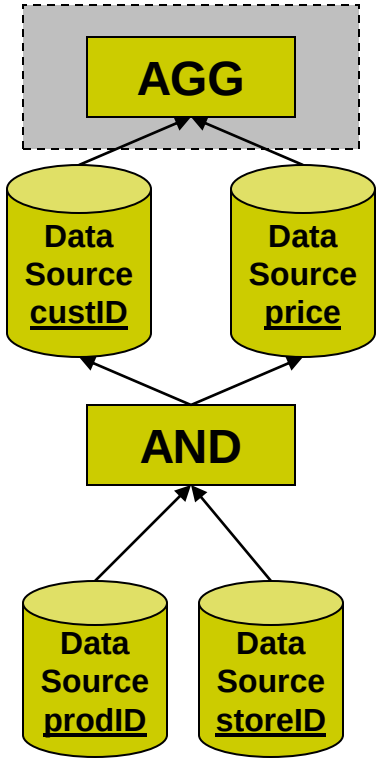


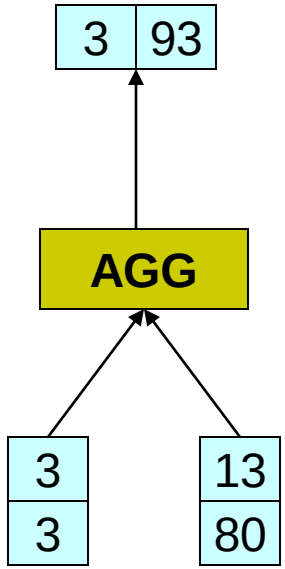
QUERY:

```
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





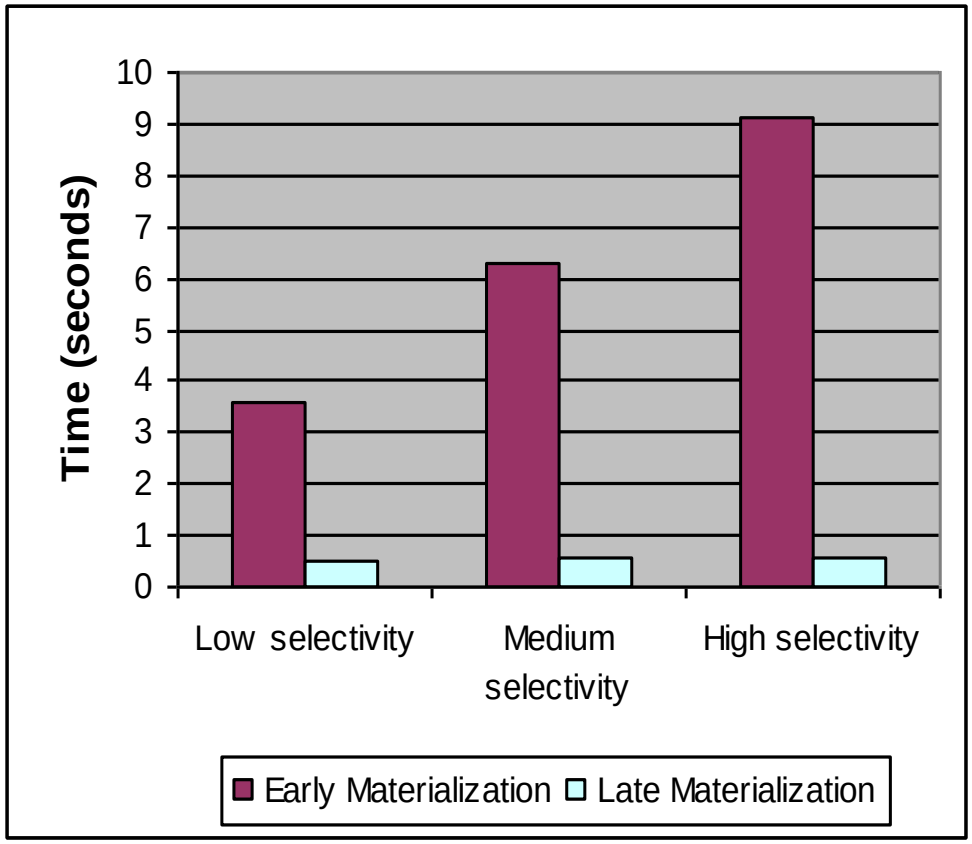
QUERY:

```
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

"Materialization Strategies in a Column-Oriented DBMS"
Abadi, Myers, DeWitt, and Madden. ICDE 2007.

For plans without joins, late materialization is a win



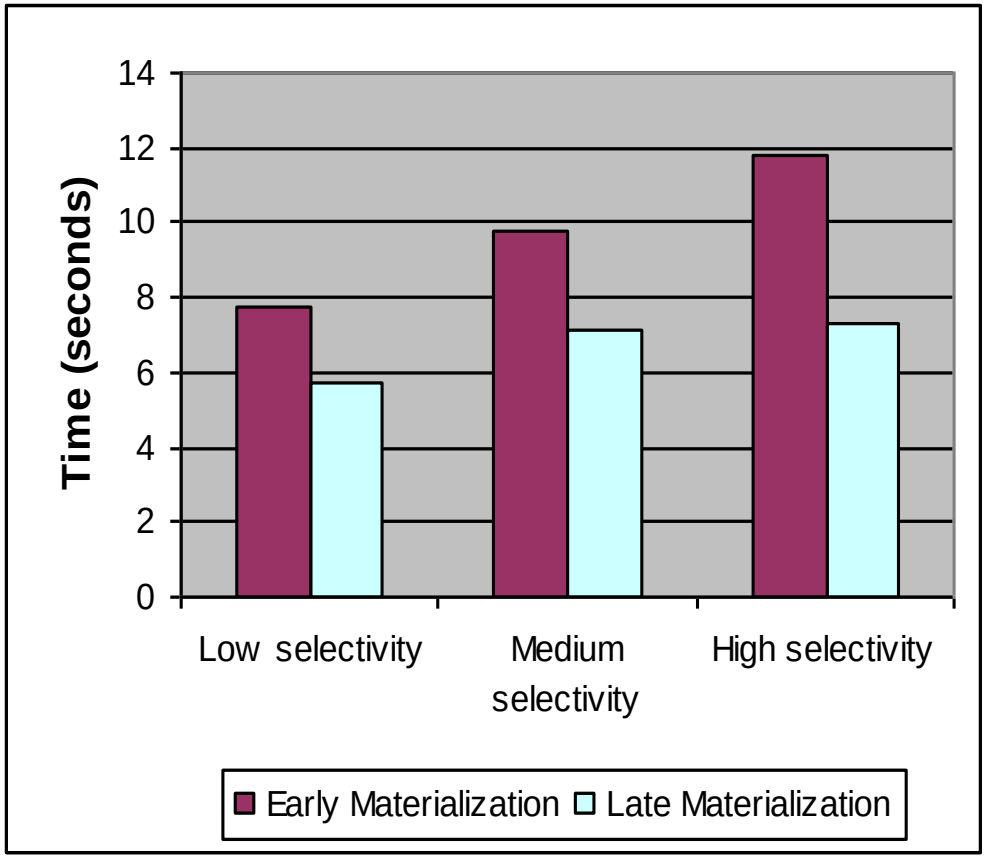
QUERY:

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

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

"Materialization Strategies in a Column-Oriented DBMS"
Abadi, Myers, DeWitt, and Madden. ICDE 2007.

Even on uncompressed data, late materialization is still a win



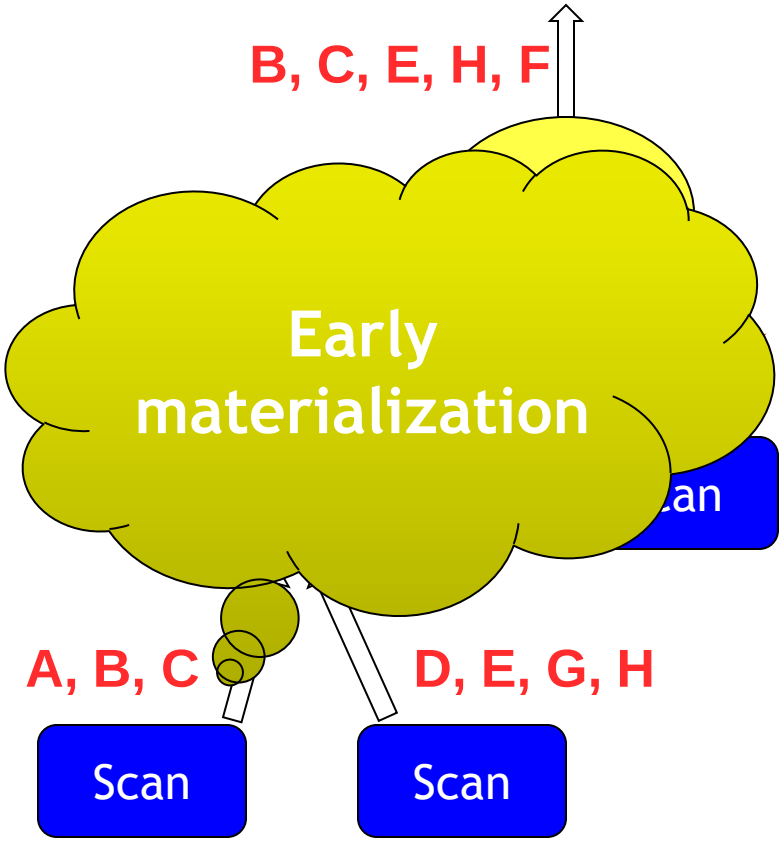
QUERY:

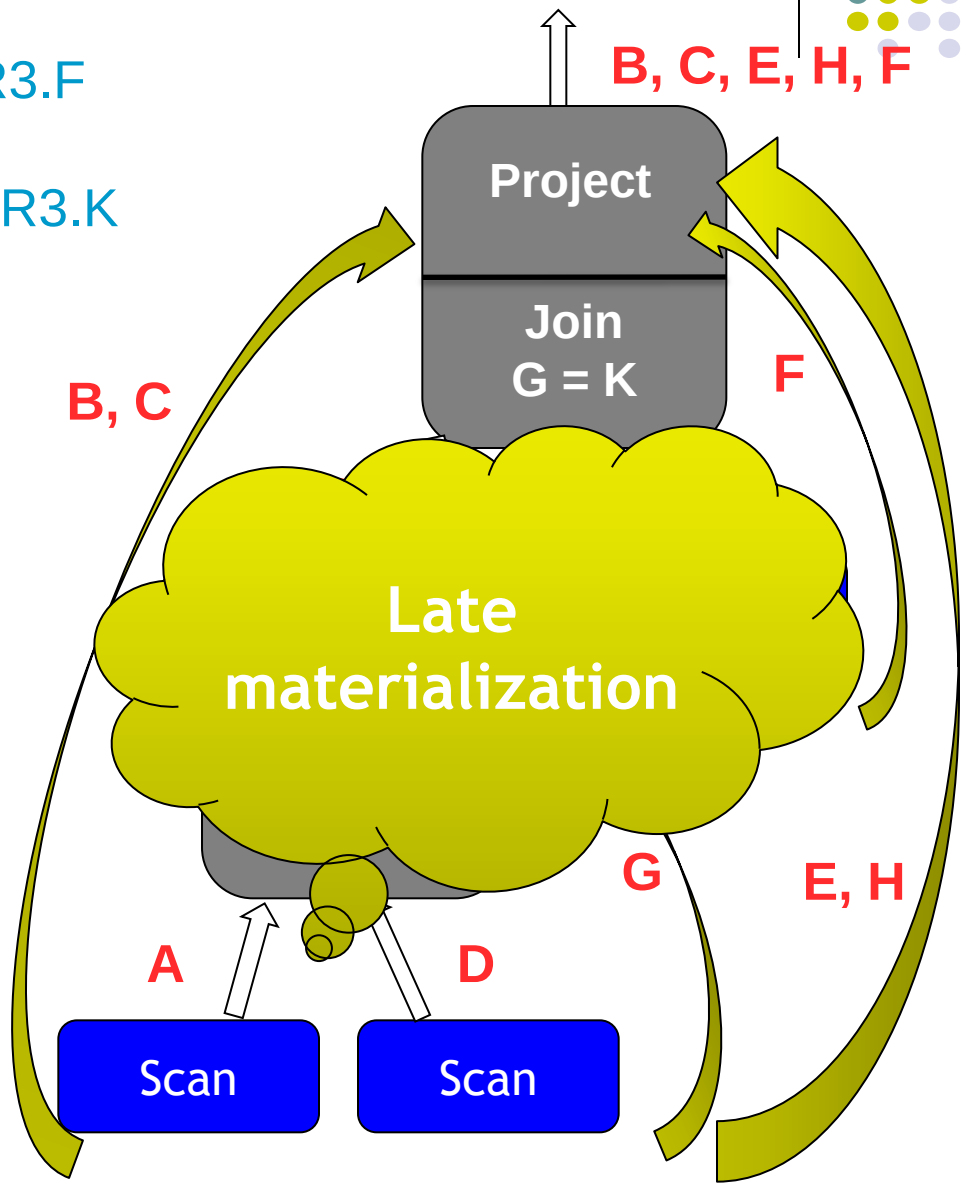
```
SELECT C1, SUM(C2)
FROM table
WHERE (C1 < CONST) AND
      (C2 < CONST)
GROUP BY C1
```

- Materializing late still works best

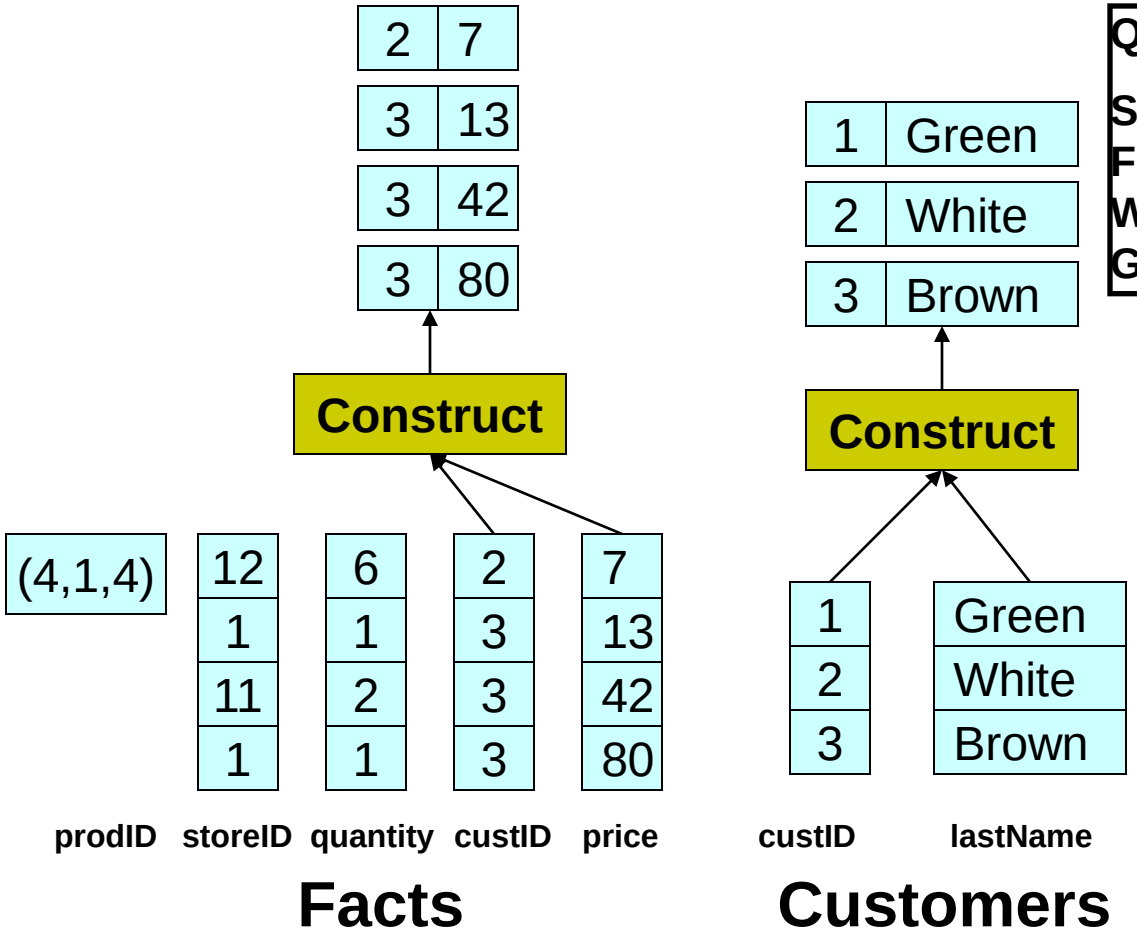
What about for plans with joins?

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





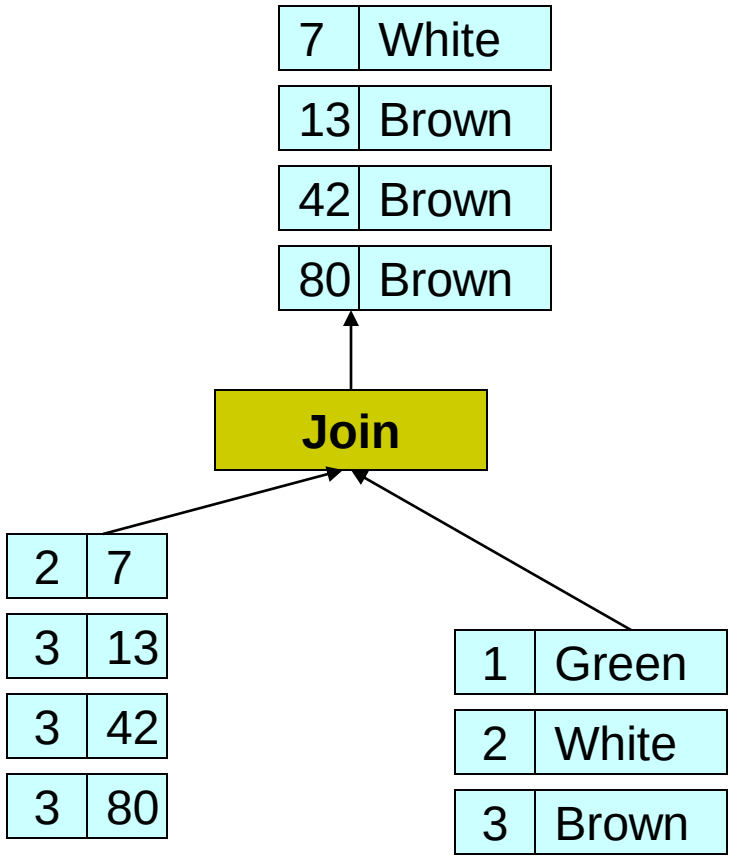
Early Materialization Example



QUERY:

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

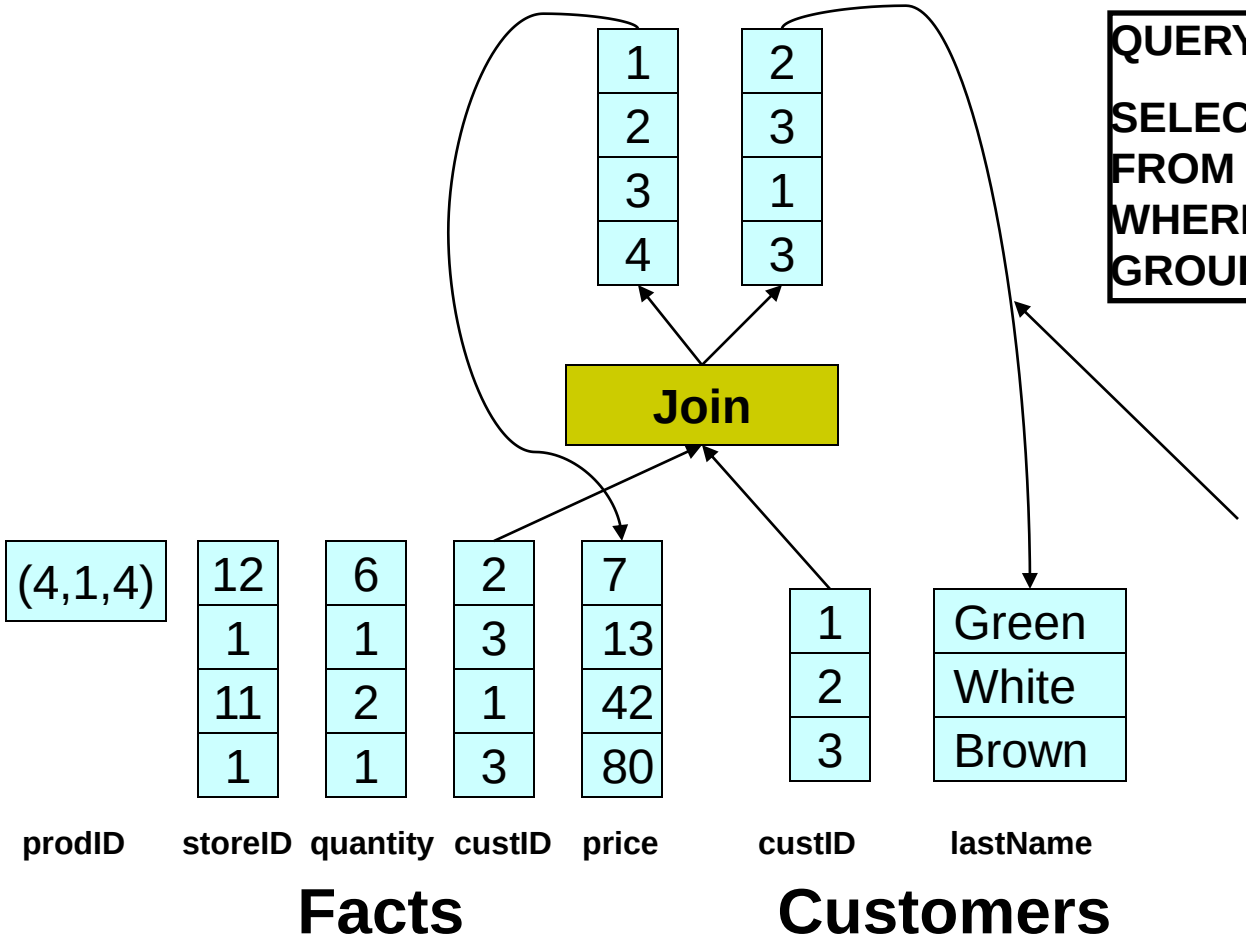
Early Materialization Example



QUERY:

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

Late Materialization Example

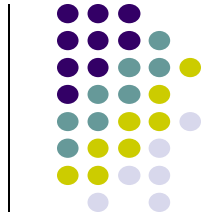


QUERY:

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

Late materialized join causes out of order probing of projected columns from the inner relation

Late Materialized Join Performance



- Naïve LM join about 2X slower than EM join on typical queries (due to random I/O)
  - This number is very dependent on
    - Amount of memory available
    - Number of projected attributes
    - Join cardinality
- But we can do better
  - Invisible Join
  - Jive/Flash Join
  - Radix cluster/decluster join

Invisible Join

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Madden, and Hachem. SIGMOD 2008.

- 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;
```

Invisible Join

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Madden, and Hachem. SIGMOD 2008.

Apply "region = 'Asia'" On Customer Table

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

Hash Table (or bit-map) Containing Keys 1, 2 and 3

Apply "region = 'Asia'" On Supplier Table

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

Hash Table (or bit-map) Containing Keys 1, 3

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

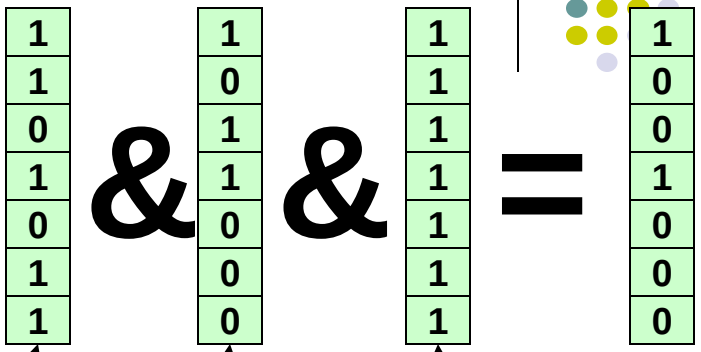
dateid	year	…
01011997	1997	…
01021997	1997	…
01031997	1997	…

Hash Table Containing Keys 01011997, 01021997, and 01031997

Original Fact Table

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

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi et. al. SIGMOD 2008.



Hash Table Containing Keys 1, 2 and 3

custkey
3
3
4
1
4
1
3

→

1
1
0
1
0
1
1

Hash Table Containing Keys 1 and 3

suppkey
1
2
3
1
2
2
2

→

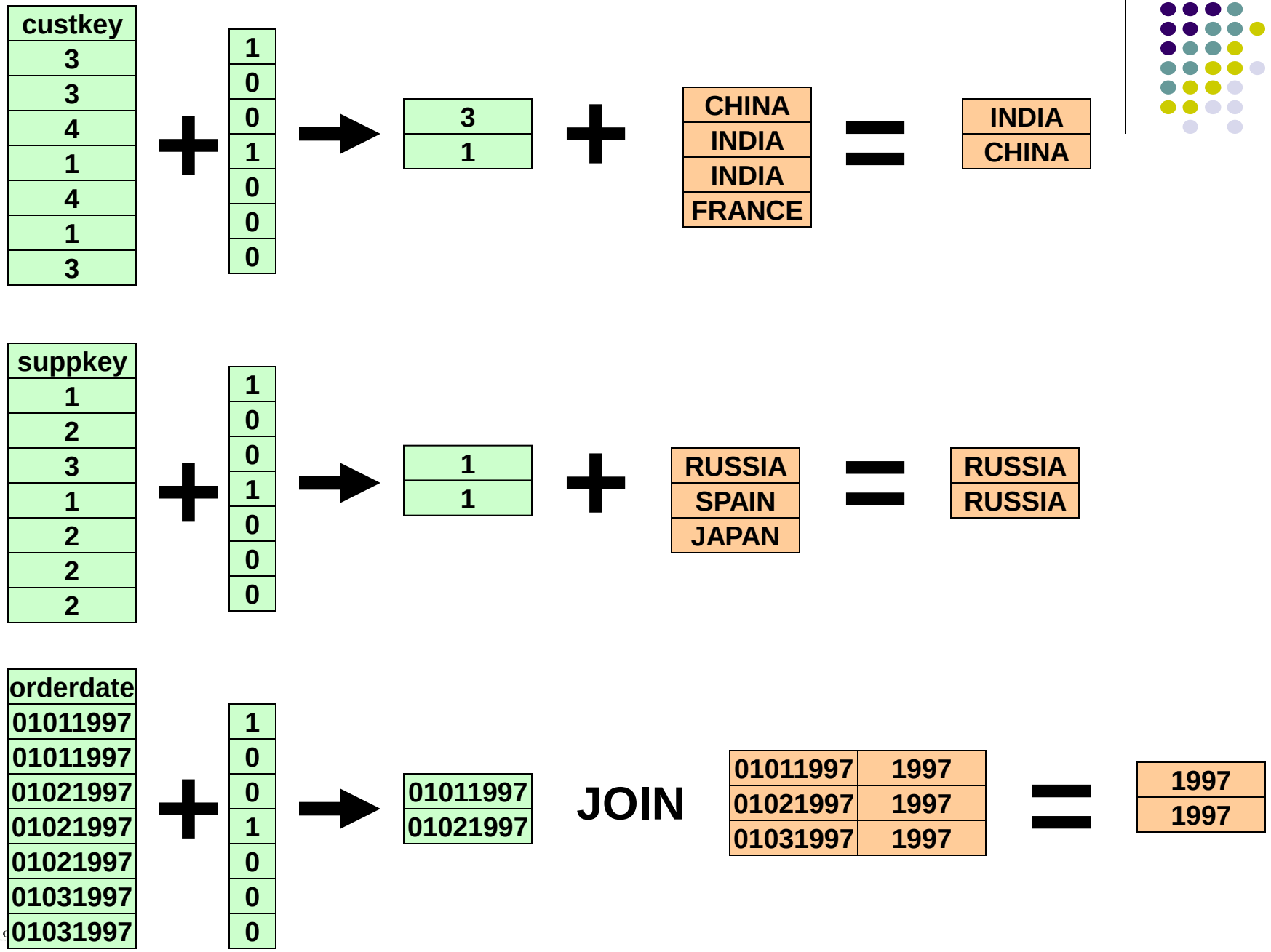
1
0
1
1
0
0
0

Hash Table Containing Keys 01011997, 01021997, and 01031997

orderdate
01011997
01011997
01021997
01021997
01021997
01031997
01031997

→

1
1
1
1
1
1
1


custkey
3
3
4
1
4
1
3
1
0
0
1
0
0
0
3
1
CHINA
INDIA
INDIA
FRANCE
INDIA
CHINA
suppkey
1
2
3
1
2
2
2
1
0
0
1
0
0
0
1
1
RUSSIA
SPAIN
JAPAN
RUSSIA
RUSSIA
orderdate
01011997
01011997
01021997
01021997
01021997
01031997
01031997
1
0
0
1
0
0
0
01011997
01021997
JOIN
01011997 1997
01021997 1997
01031997 1997
1997
1997

Invisible Join

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Madden, and Hachem. SIGMOD 2008.

Apply "region = 'Asia'" On Customer Table

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

~~Hash Table (or bit-map) Containing Keys 1, 2 and 3~~

Range [1-3]

(between-predicate rewriting)

Apply "region = 'Asia'" On Supplier Table

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

Hash Table (or bit-map) Containing Keys 1, 3

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

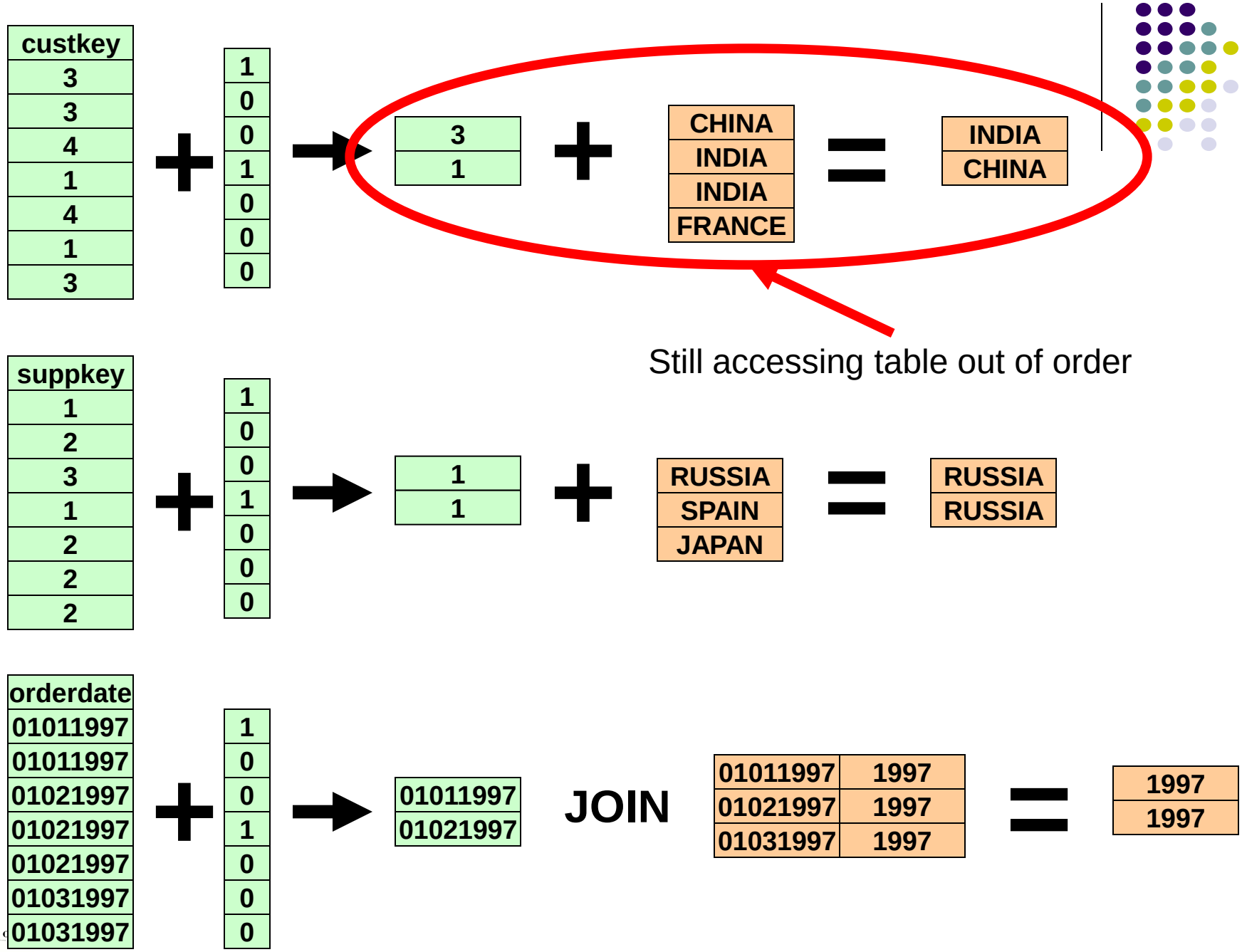
dateid	year	…
01011997	1997	…
01021997	1997	…
01031997	1997	…

Hash Table Containing Keys 01011997, 01021997, and 01031997

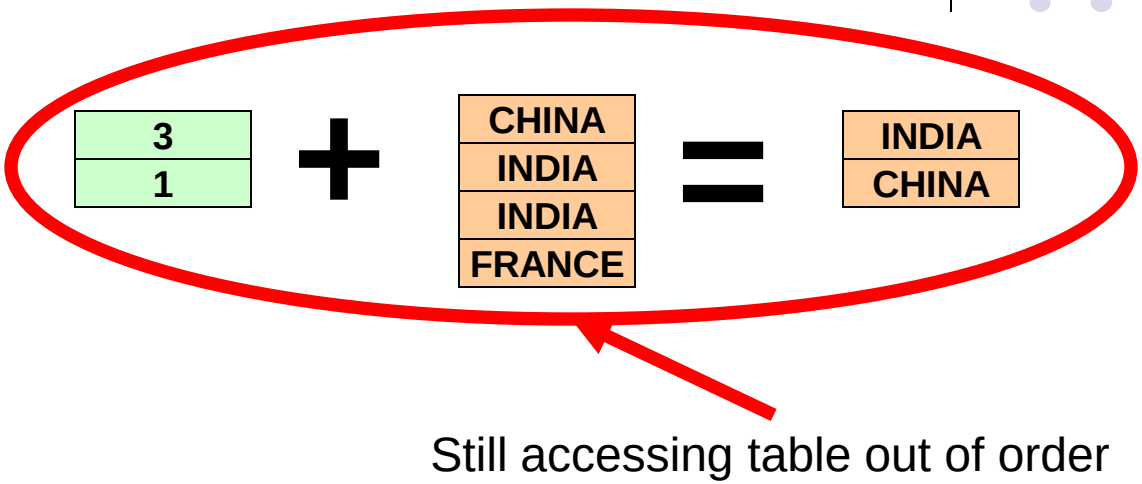
Invisible Join

Bottom Line

- Many data warehouses model data using star/snowflake schemes
- Joins of one (fact) table with many dimension tables is common
- Invisible join takes advantage of this by making sure that the table that can be accessed in position order is the fact table for each join
- Position lists from the fact table are then intersected (in position order)
- This reduces the amount of data that must be accessed out of order from the dimension tables
- “Between-predicate rewriting” trick not relevant for this discussion


custkey
3
3
4
1
4
1
3
1
0
0
1
0
0
0
3
1
CHINA
INDIA
INDIA
FRANCE
INDIA
CHINA
Still accessing table out of order
suppkey
1
2
3
1
2
2
2
1
0
0
1
0
0
0
1
1
RUSSIA
SPAIN
JAPAN
RUSSIA
RUSSIA
orderdate
01011997
01011997
01021997
01021997
01021997
01031997
01031997
1
0
0
1
0
0
0
01011997
01021997
JOIN
01011997 1997
01021997 1997
01031997 1997
1997
1997

Jive/Flash Join

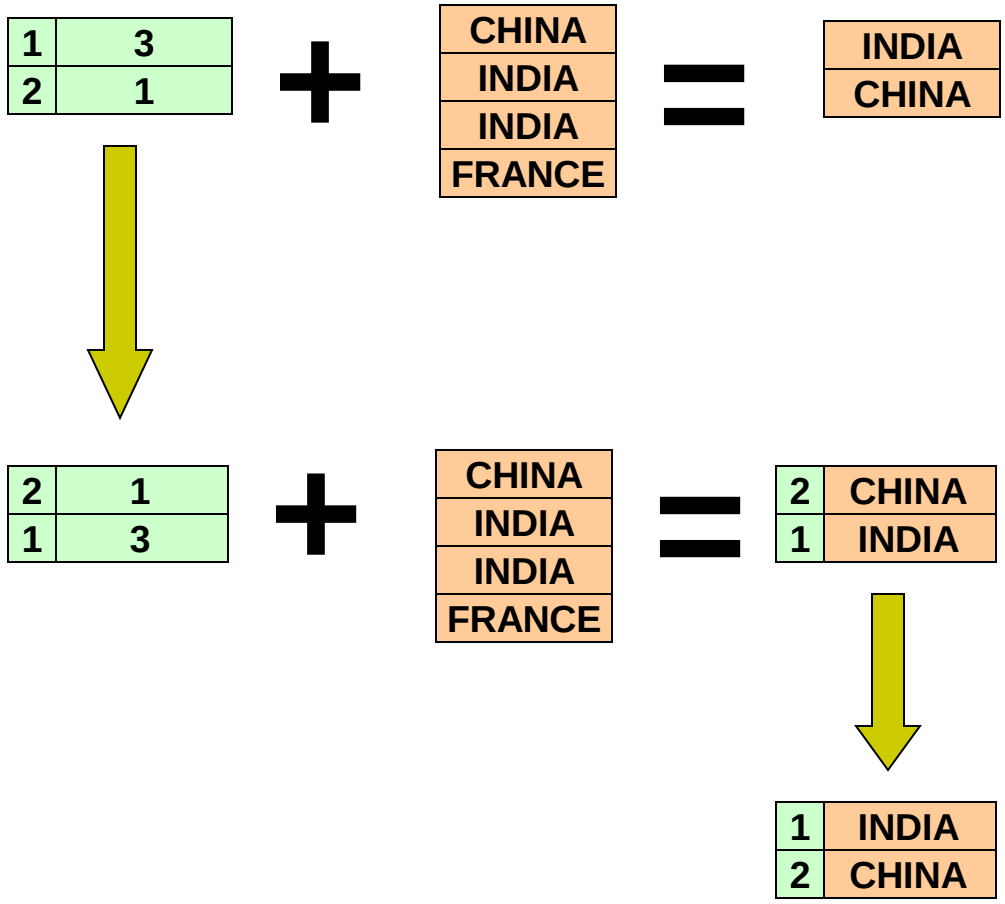


"Fast Joins using Join Indices". Li and Ross, VLDBJ 8:1-24, 1999.

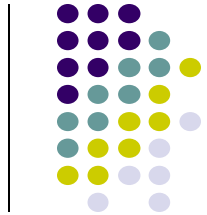
"Query Processing Techniques for Solid State Drives". Tsirogiannis, Harizopoulos et. al. SIGMOD 2009.

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



Jive/Flash Join



Bottom Line

- Instead of probing projected columns from inner table out of order:
  - Sort join index
  - Probe projected columns in order
  - Sort result using an added column
- LM vs EM tradeoffs:
  - LM has the extra sorts (EM accesses all columns in order)
  - LM only has to fit join columns into memory (EM needs join columns and all projected columns)
    - Results in big memory and CPU savings (see part 3 for why there is CPU savings)
  - LM only has to materialize relevant columns
  - In many cases LM advantages outweigh disadvantages
- LM would be a clear winner if not for those pesky sorts … can we do better?

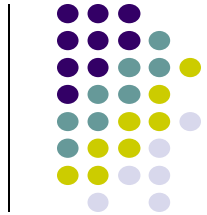
Radix Cluster/Decluster

- The full sort from the Jive join is actually overkill
  - We just want to access the storage blocks in order (we don't mind random access within a block)
  - So do a radix sort and stop early
  - By stopping early, data within each block is accessed out of order, but in the order specified in the original join index
    - Use this pseudo-order to accelerate the post-probe sort as well

•"Database Architecture Optimized for the New Bottleneck: Memory Access" VLDB'99
•"Generic Database Cost Models for Hierarchical Memory Systems", VLDB'02 (all Manegold, Boncz, Kersten)

"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

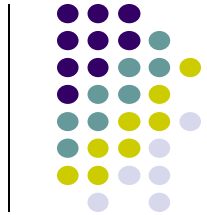
Radix Cluster/Decluster



Bottom line

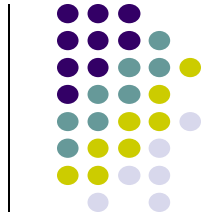
- Both sorts from the Jive join can be significantly reduced in overhead
- Only been tested when there is sufficient memory for the entire join index to be stored three times
  - Technique is likely applicable to larger join indexes, but utility will go down a little
- Only works if random access within a storage block
  - Don’t want to use radix cluster/decluster if you have variable-width column values or compression schemes that can only be decompressed starting from the beginning of the block

LM vs EM joins



- Invisible, Jive, Flash, Cluster, Decluster techniques contain a bag of tricks to improve LM joins
- Research papers show that LM joins become 2X faster than EM joins (instead of 2X slower) for a wide array of query types

Tuple Construction Heuristics

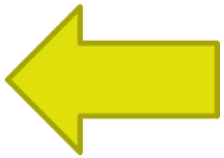


- For queries with selective predicates, aggregations, or compressed data, use late materialization
- For joins:
  - Research papers:
    - Always use late materialization
  - Commercial systems:
    - Inner table to a join often materialized before join (reduces system complexity):
    - Some systems will use LM only if columns from inner table can fit entirely in memory

Outline

- Part 1: Basic concepts
  - Introduction to key features
  - From DSM to column-stores and performance tradeoffs
  - Column-store architecture overview
  - Will rows and columns ever converge?
- Part 2: Column-oriented execution

- Part 3: MonetDB/X100 (“VectorWise”) and CPU efficiency



Outline

- Computational Efficiency of DB on modern hardware
  - how column-stores can help here
  - MonetDB & VectorWise in more depth
- CPU efficient column compression
  - vectorized decompression
- Conclusions
  - future work

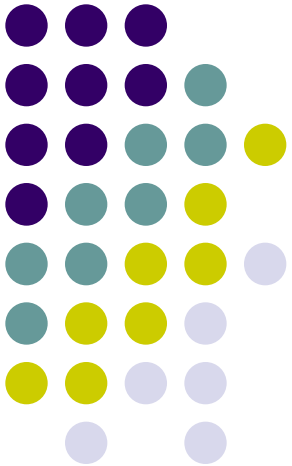
Column-Oriented Database Systems

Tutorial

40 years of hardware evolution

vs.

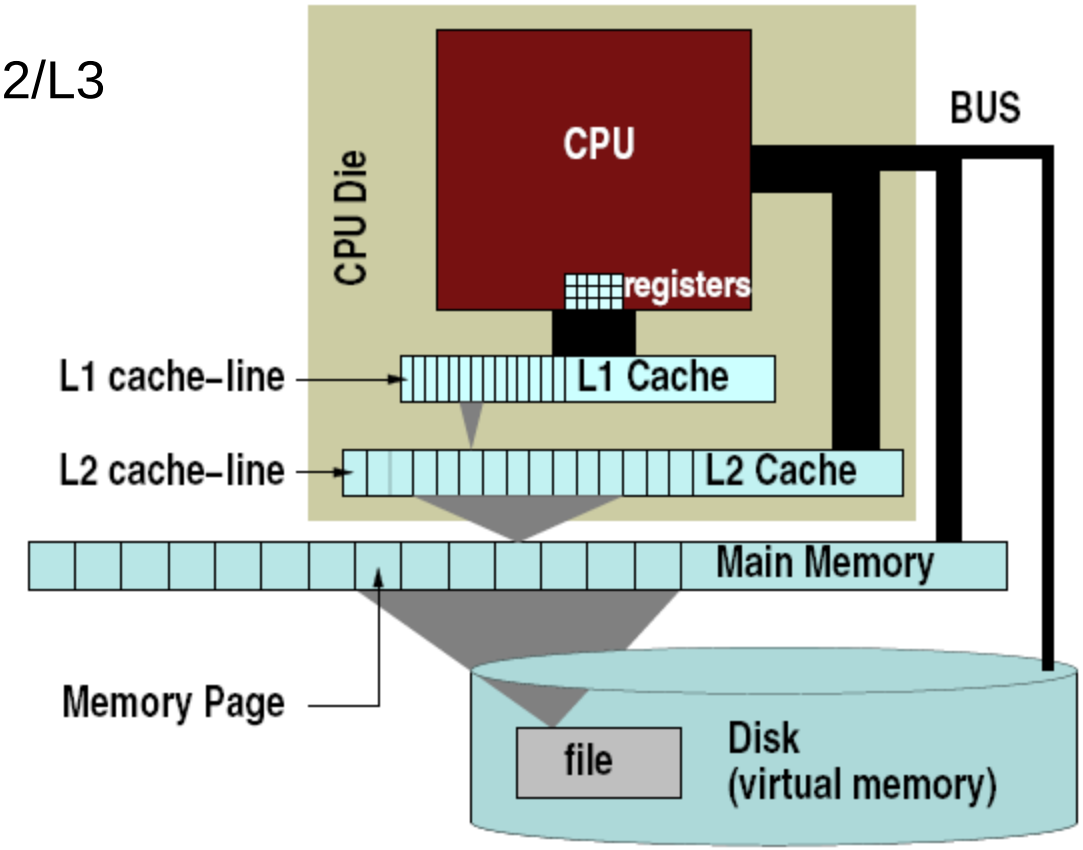
DBMS computational efficiency



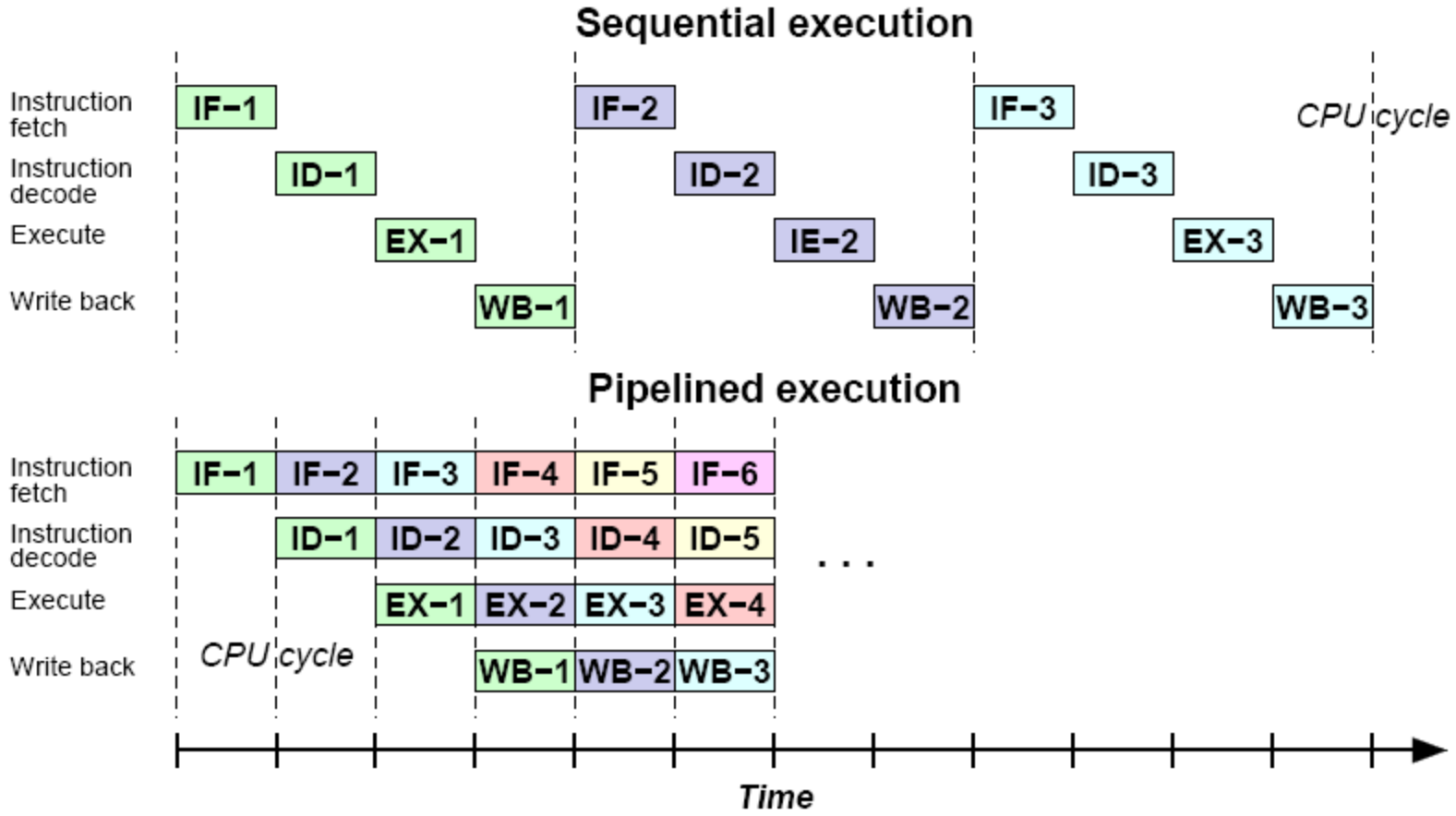
CPU Architecture

Elements:

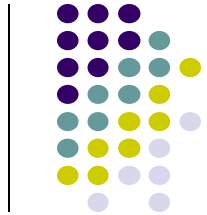
- Storage
  - CPU caches L1/L2/L3
- Registers
- Execution Unit(s)
  - Pipelined
  - SIMD



Super-Scalar Execution (pipelining)

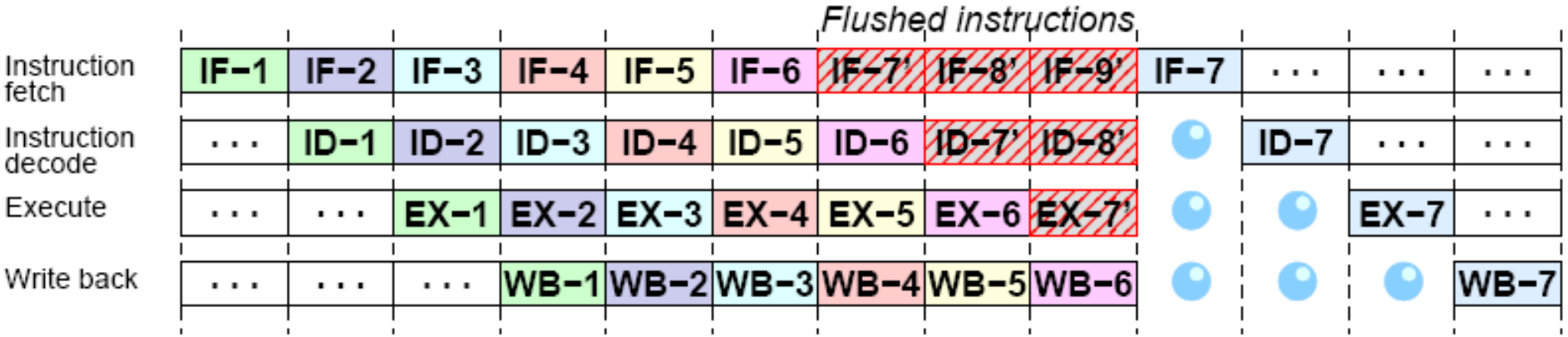


Hazards



- Data hazards
  - Dependencies between instructions
  - L1 data cache misses
- Control Hazards
  - Branch mispredictions
  - Computed branches (late binding)
  - L1 instruction cache misses

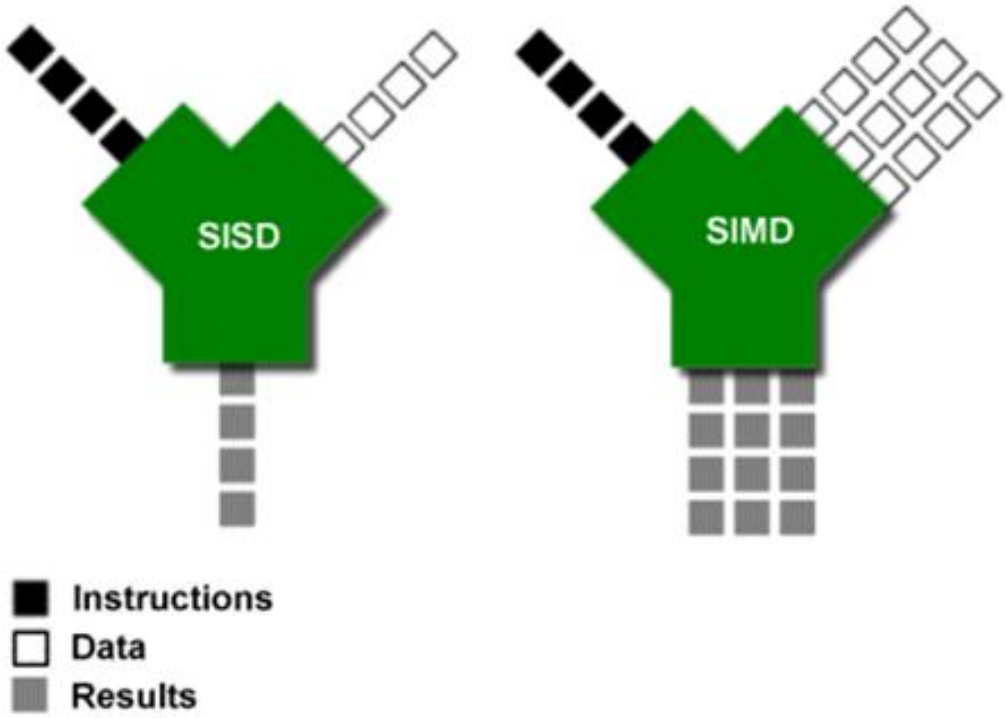
Result: bubbles in the pipeline



Out-of-order execution addresses data hazards

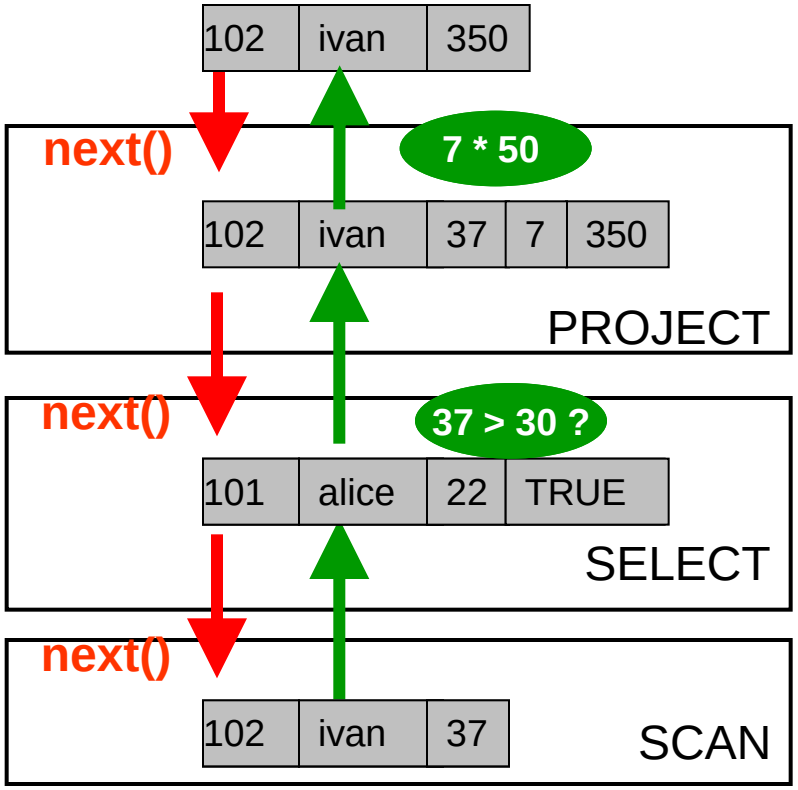
- control hazards typically more expensive

SIMD



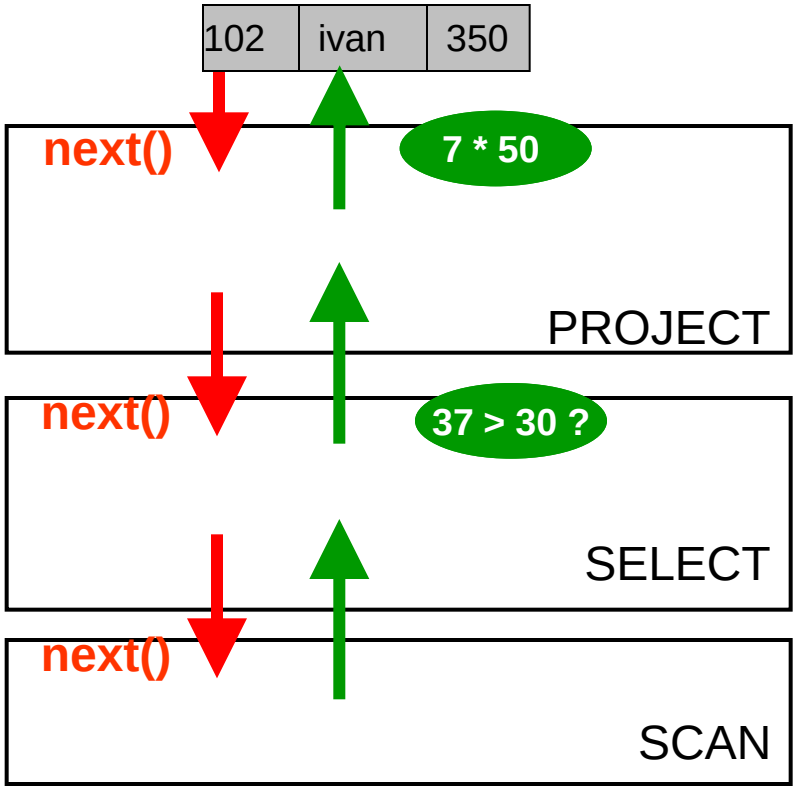
- Single Instruction Multiple Data
  - Same operation applied on a vector of values
  - MMX: 64 bits, SSE: 128bits, AVX: 256bits
  - SSE, e.g. multiply 8 short integers

A Look at the Query Pipeline




```
SELECT   id, name
         (age-30)*50 AS bonus
FROM     employee
WHERE    age > 30
```

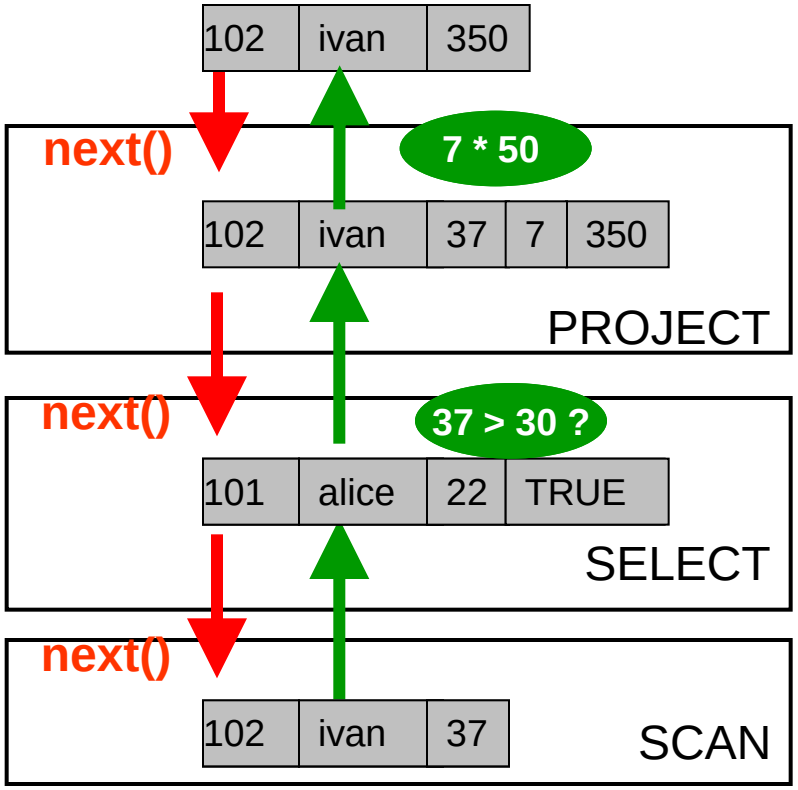
A Look at the Query Pipeline



Operators

Iterator interface
-open()
-next(): tuple
-close()

A Look at the Query Pipeline



Primitives

Provide computational functionality

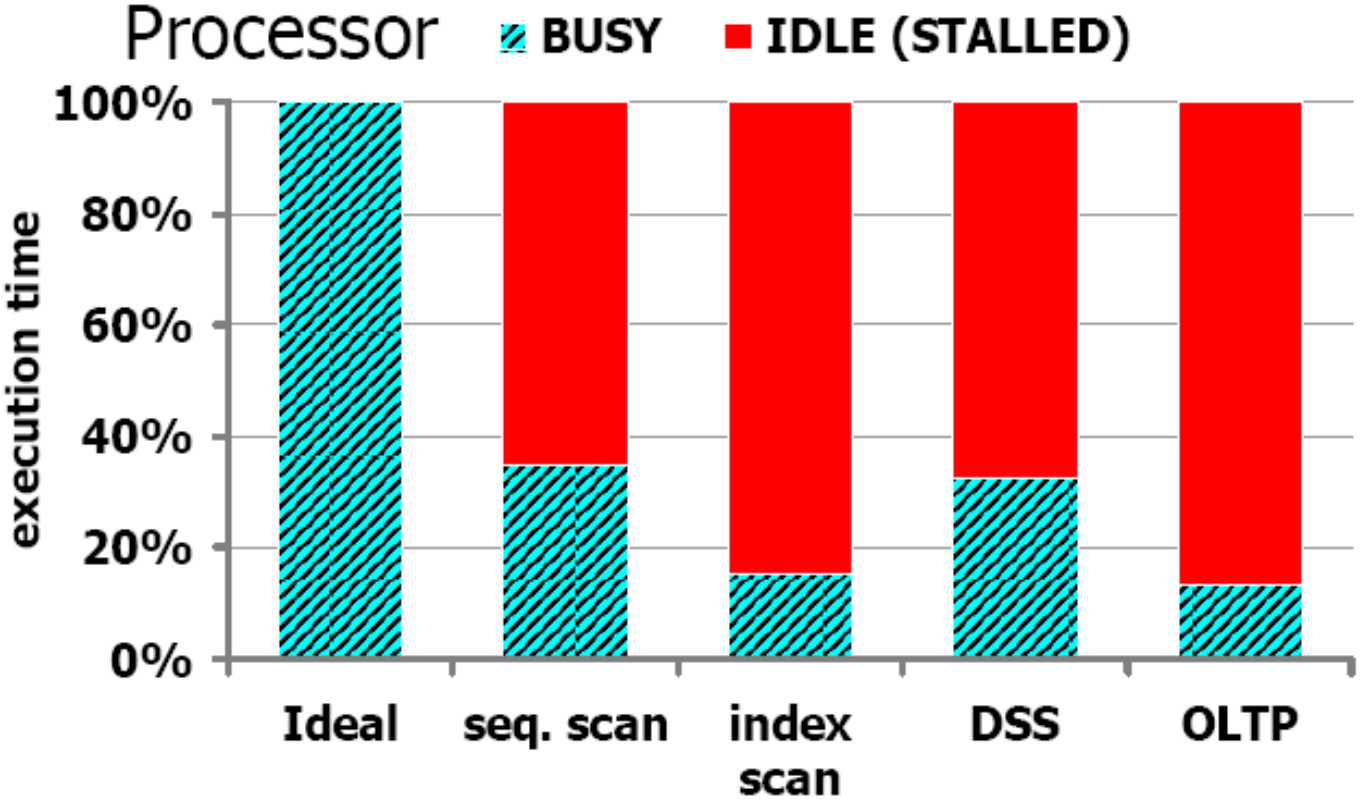
All arithmetic allowed in expressions,
e.g. Multiplication

7 * 50

`mult(int,int)` ➔ `int`

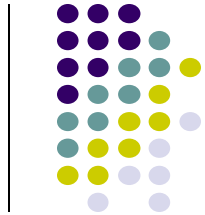
Database Architecture causes Hazards

- DB workload execution on a modern computer



"DBMSs On A Modern Processor: Where Does Time Go? "
Ailamaki, DeWitt, Hill, Wood, VLDB'99

DBMS Computational Efficiency

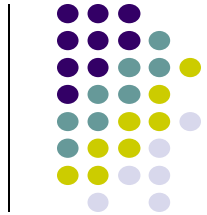


TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all
- Results:
  - C program: ?
  - MySQL: 26.2s
  - DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

DBMS Computational Efficiency



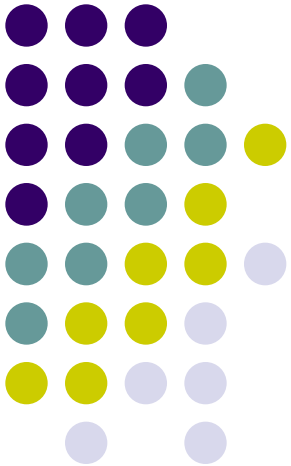
TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all
- Results:
  - C program: **0.2s**
  - MySQL: 26.2s
  - DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

Column-Oriented Database Systems

Tutorial



MonetDB a column-store

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

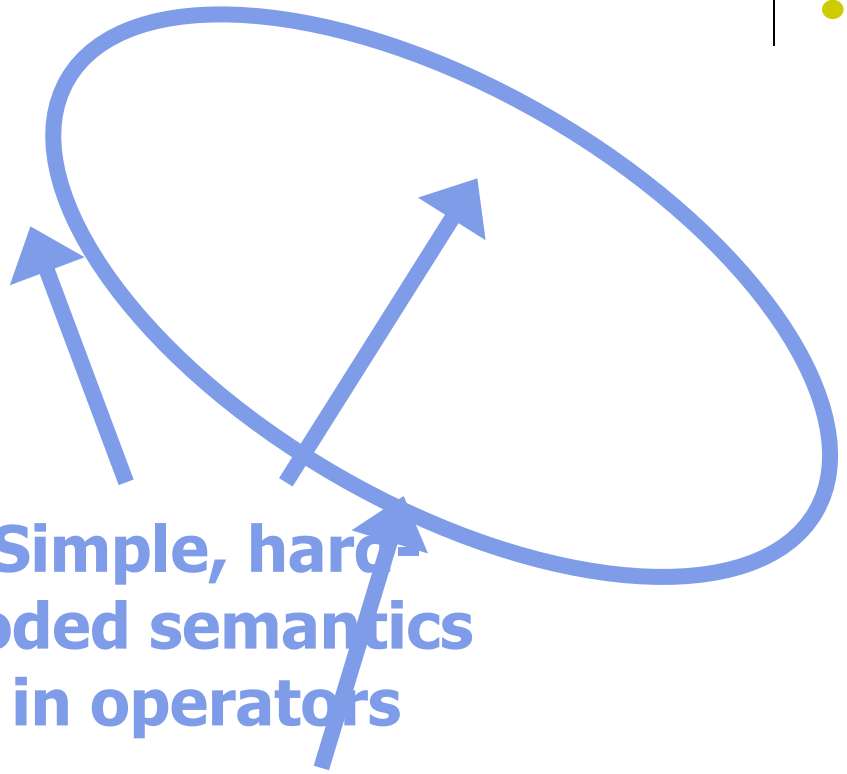
RISC Database Algebra

CPU ☺? Give it "nice" code !

- few dependencies (control,data)
- CPU gets out-of-order execution
- compiler can e.g. generate SIMD

One loop for an entire column
- no per-tuple interpretation
- arrays: no record navigation
- better instruction cache locality

```
{
    for(i=0; i<n; i++)
        res[i] = col[i] - val;
}
```



MonetDB a column-store

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

How?

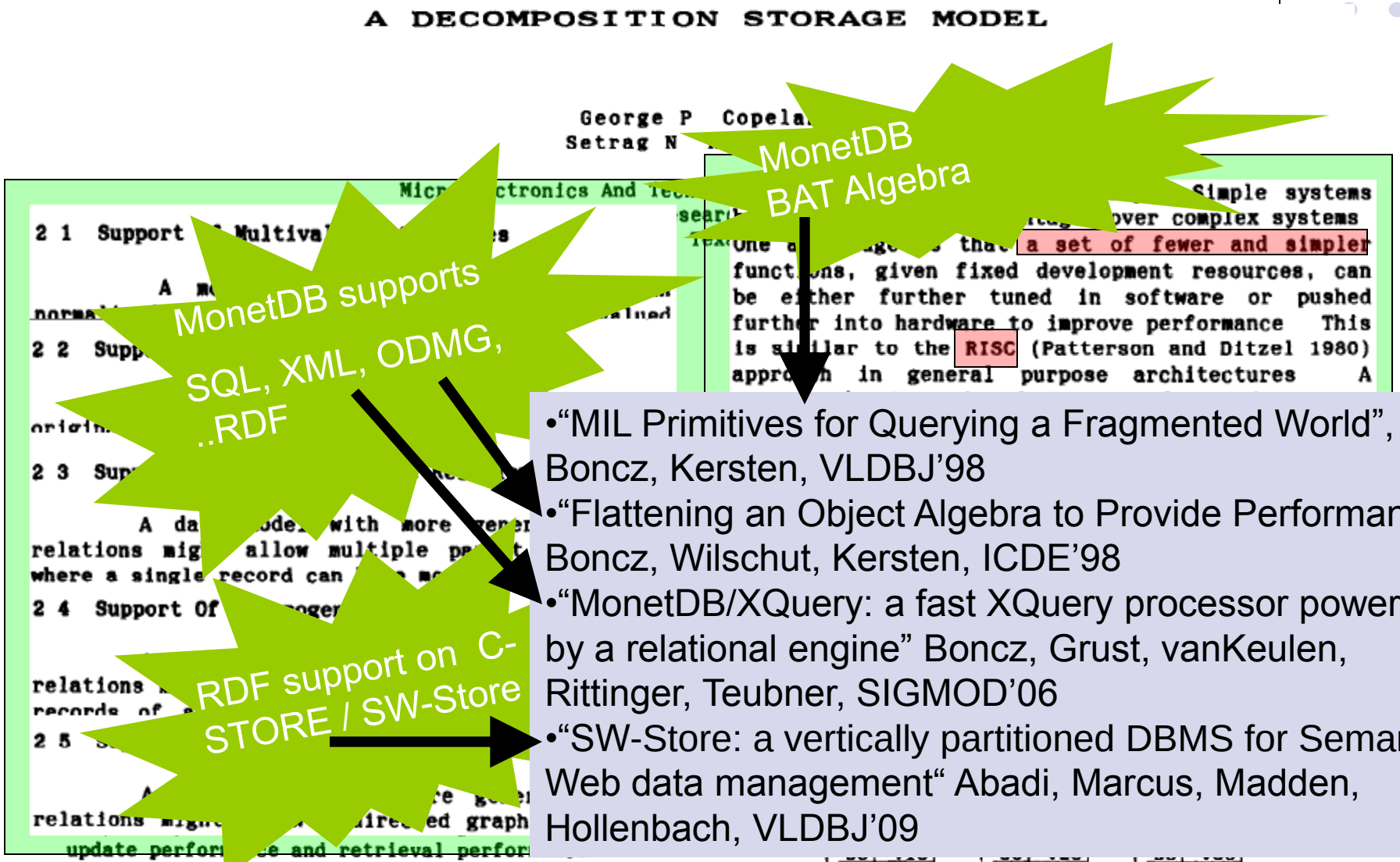
- RISC query algebra: hard-coded semantics
  - Decompose complex expressions in multiple operations
- Operators only handle **simple arrays**
  - No code that handles slotted buffered record layout
- Relational algebra becomes **array manipulation language**
  - Often SIMD for free
  - Plus: use of *cache-conscious* algorithms for Sort/Aggr/Join

MonetDB a Faustian pact

- You want efficiency
  - Simple hard-coded operators
- I take scalability
  - Result materialization

C program:	0.2s
MonetDB:	3.7s
MySQL:	26.2s
DBMS "X":	28.1s

SIGMOD 1985


A DECOMPOSITION STORAGE MODEL
George P Copela
Setrag N
MonetDB
BAT Algebra
MonetDB supports
SQL, XML, ODMG,
..RDF
RDF support on C-STORE / SW-Store
•"MIL Primitives for Querying a Fragmented World", Boncz, Kersten, VLDBJ'98
•"Flattening an Object Algebra to Provide Performance" Boncz, Wilschut, Kersten, ICDE'98
•"MonetDB/XQuery: a fast XQuery processor powered by a relational engine" Boncz, Grust, vanKeulen, Rittinger, Teubner, SIGMOD'06
•"SW-Store: a vertically partitioned DBMS for Semantic Web data management" Abadi, Marcus, Madden, Hollenbach, VLDBJ'09

optimizer frontend backend

as a research platform

- Cache-Conscious Joins
  - Cost Models, Radix-cluster Radix-decluster

•"Database Architecture Optimized for the New Bottleneck: Memory Access" VLDB'99
•"Generic Database Cost Models for Hierarchical Memory Systems", VLDB'02 (all Manegold, Boncz, Kersten)
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

- MonetDB/XQuery:
  - structural joins exploiting positional column access

"MonetDB/XQuery: a fast XQuery processor powered by a relational engine" Boncz, Grust, vanKeulen, Rittinger, Teubner, SIGMOD'06

- Cracking:
  - on-the-fly automatic indexing without workload knowledge

"Database Cracking", CIDR'07
"Updating a cracked database ", SIGMOD'07
"Self-organizing tuple reconstruction in column-stores", SIGMOD'09 (all Idreos, Manegold, Kersten)

- Recycling:
  - using materialized intermediates

"An architecture for recycling intermediates in a column-store", Ivanova, Kersten, Nes, Goncalves, SIGMOD'09

- Run-time Query Optimization:
  - correlation-aware run-time optimization without cost model

"ROX: run-time optimization of XQueries", Abdelkader, Boncz, Manegold, vanKeulen, SIGMOD'09

Cache-Conscious Joins
Radix-Cluster

- Radix-Partitioned Hash Join
  - create partitions << CPU cache
  - small partitions ➔ many partitions
  - many partitions ➔ multiple passes needed

•"Database Architecture Optimized for the New Bottleneck: Memory Access" VLDB'99
•"Generic Database Cost Models for Hierarchical Memory Systems", VLDB'02 (all Manegold, Boncz, Kersten)

Cache-Conscious Joins
Radix-Cluster

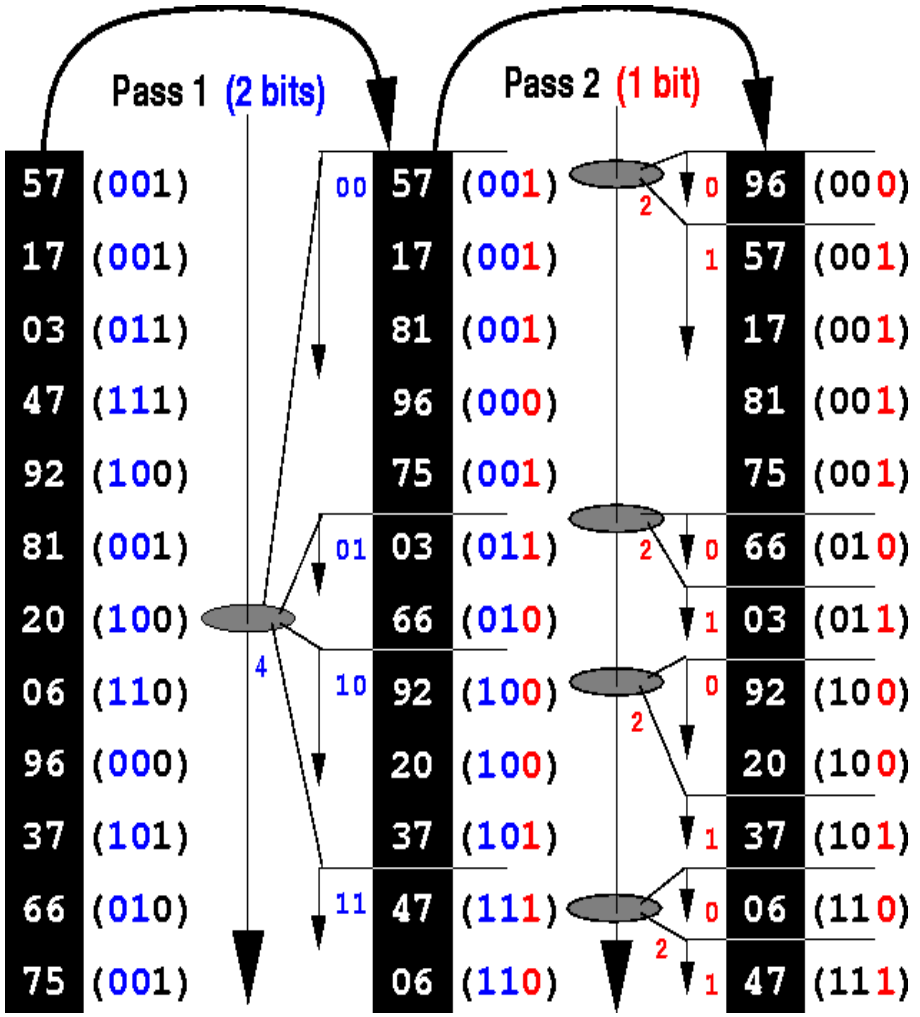
- Radix-Partitioned Hash Join
  - create partitions << CPU cache
  - small partitions ➔ many partitions
  - many partitions ➔ multiple passes needed

- Radix-Cluster
  - Radix-Sort with early stopping
    - Each pass looks at Bi higher-most radix bits
    - Splitting each input cluster into 2^{Bi} output clusters
  - leaves relation partially ordered

•"Database Architecture Optimized for the New Bottleneck: Memory Access" VLDB'99
•"Generic Database Cost Models for Hierarchical Memory Systems", VLDB'02 (all Manegold, Boncz, Kersten)

Cache-Conscious Joins

Radix-Cluster

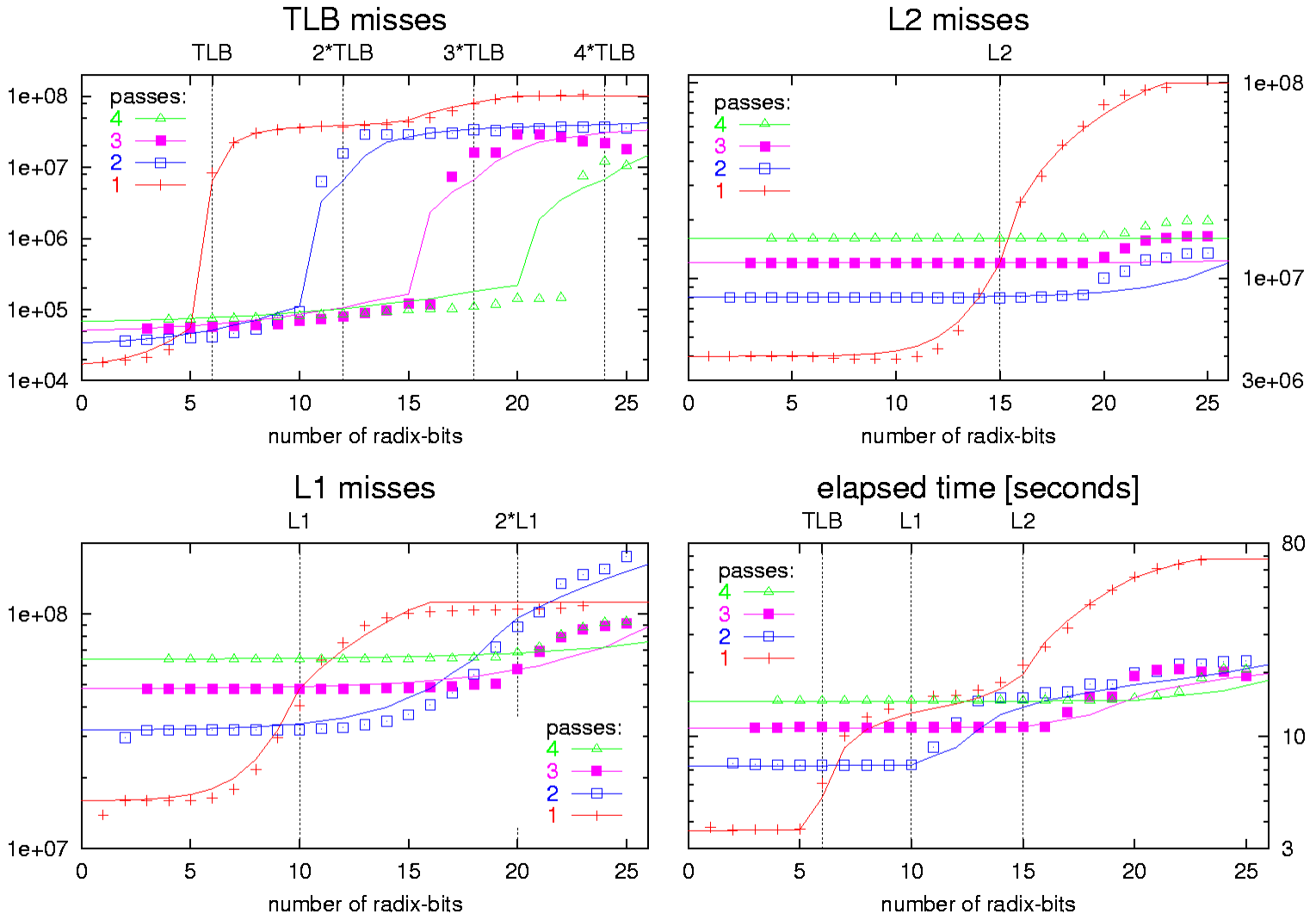


- Multiple clustering passes
- Limit number of clusters per pass
- Avoid cache/TLB trashing
- Trade memory cost for CPU cost
- Any data type (hashing)

•"Database Architecture Optimized for the New Bottleneck: Memory Access" VLDB'99
•"Generic Database Cost Models for Hierarchical Memory Systems", VLDB'02 (all Manegold, Boncz, Kersten)

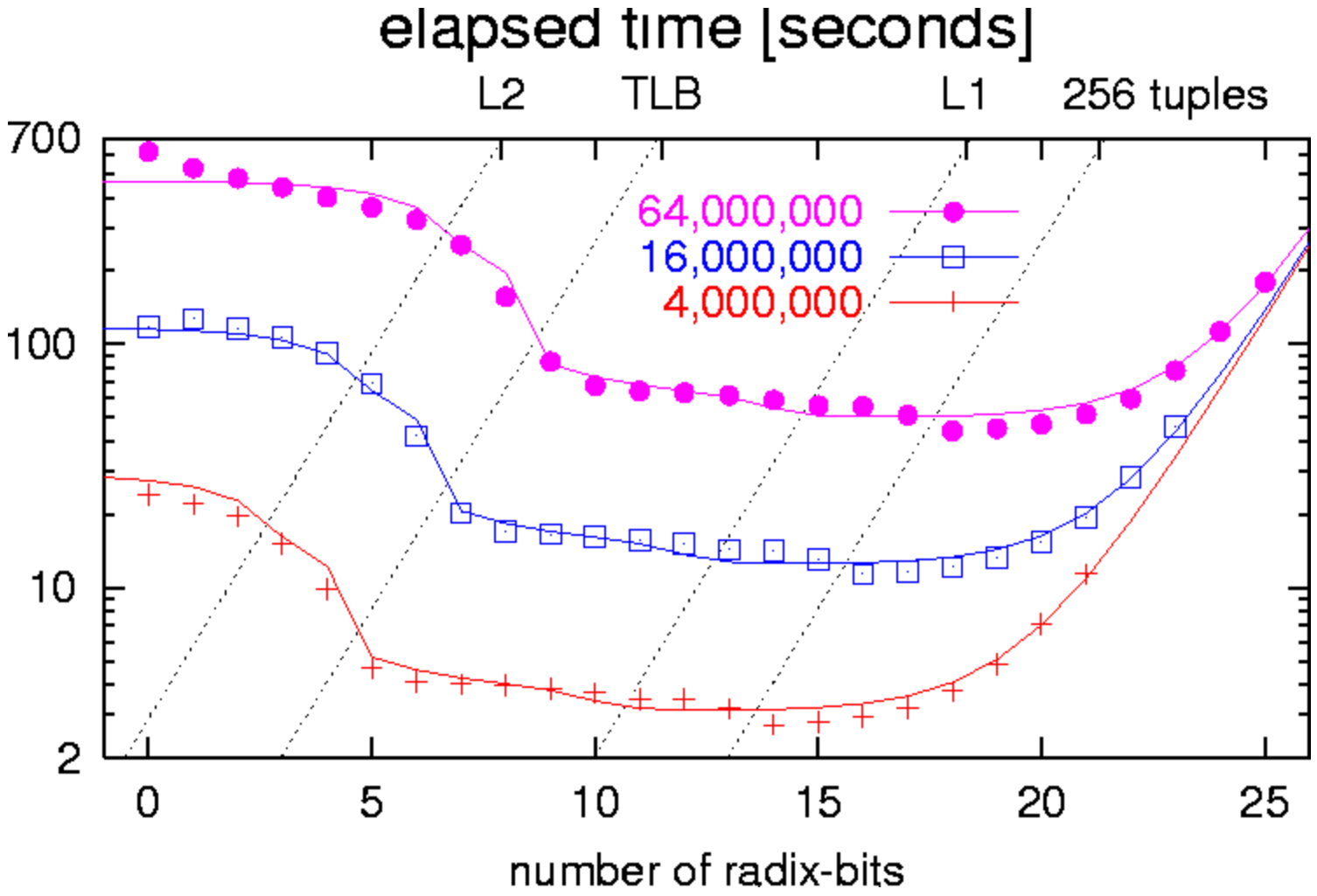
Cache-Conscious Joins

Accurate Cache Miss Cost Modeling



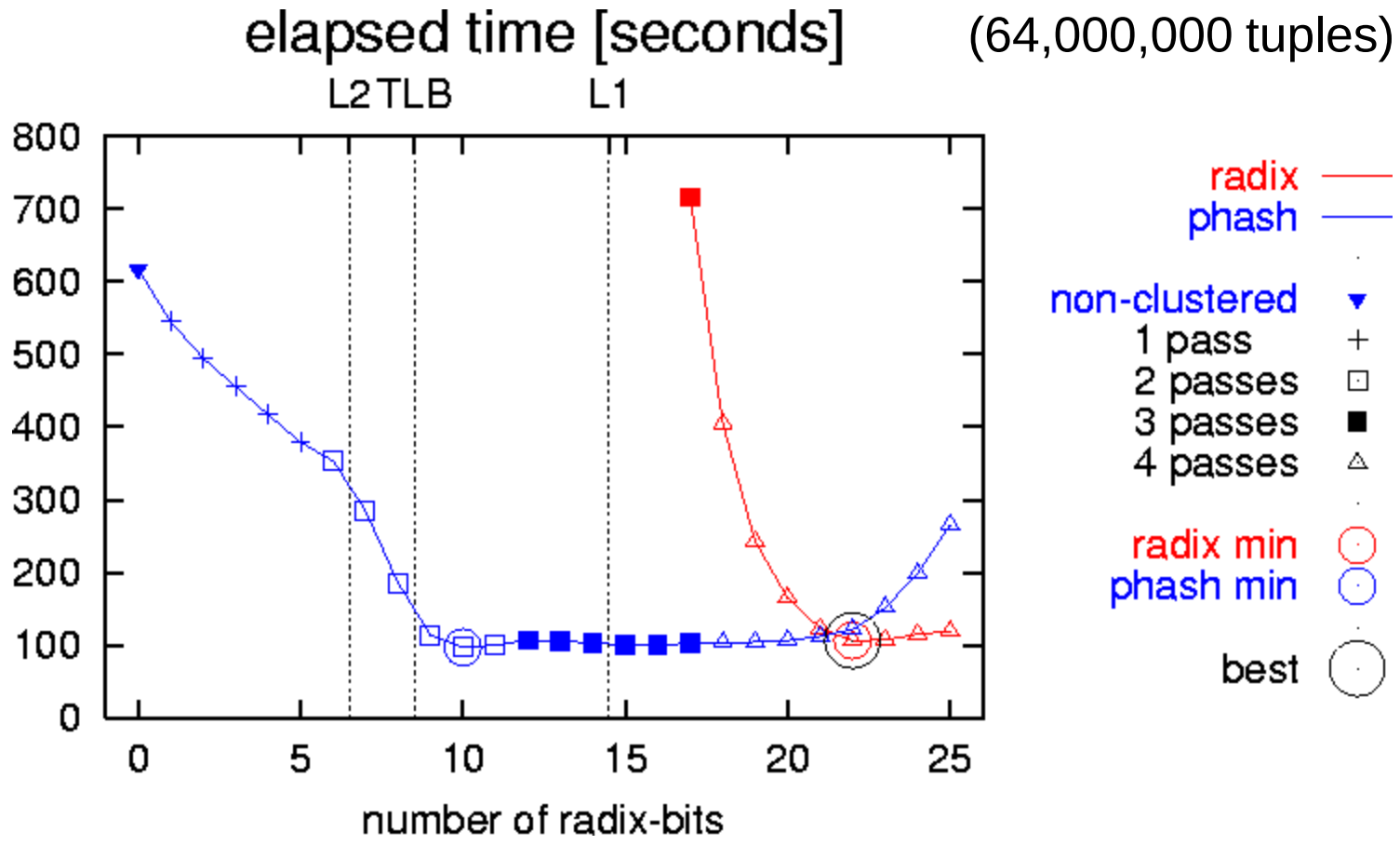
Cache-Conscious Joins

Partitioned Hash Join

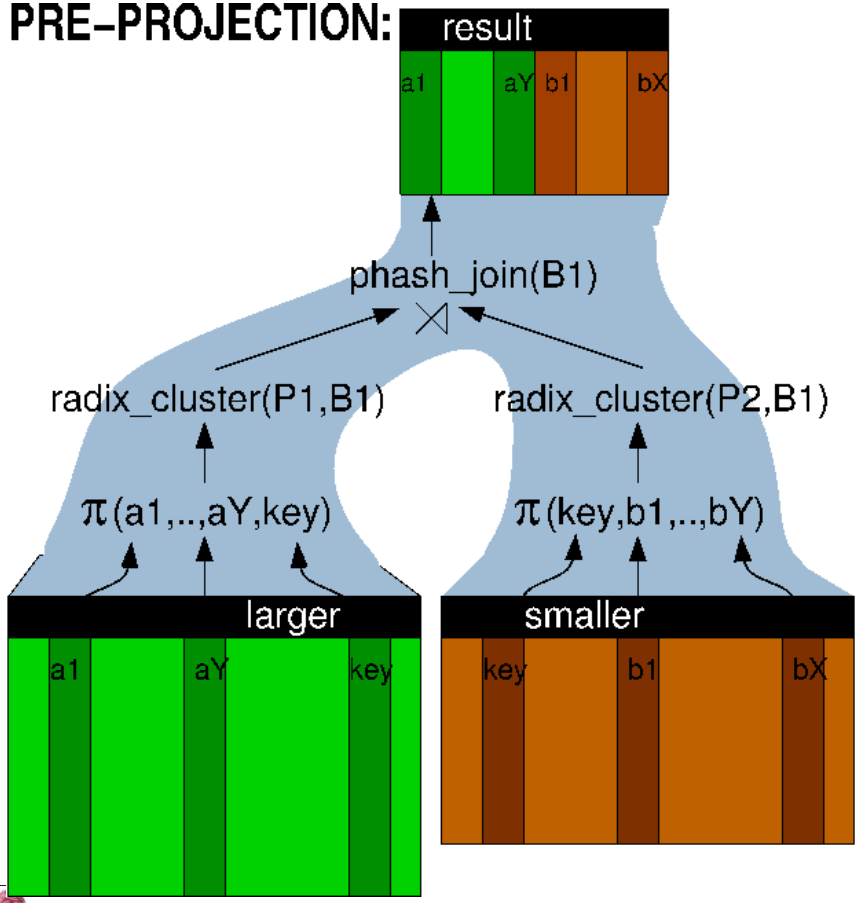


Cache-Conscious Joins

Radix-Clustered Hash-Join: overall perf

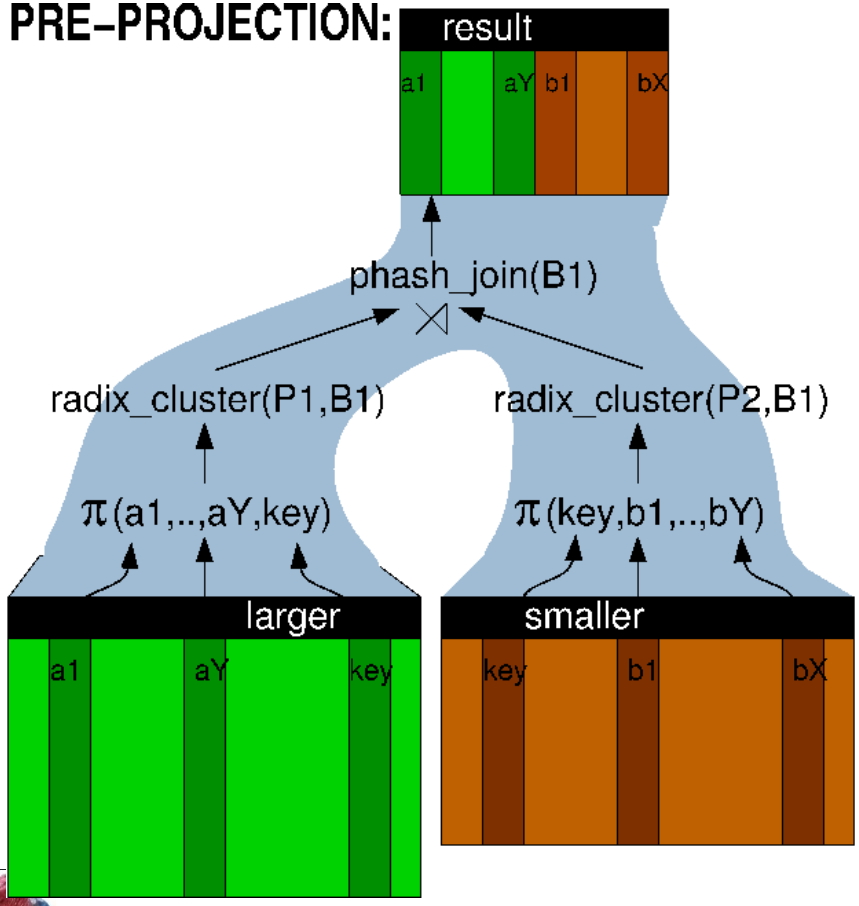


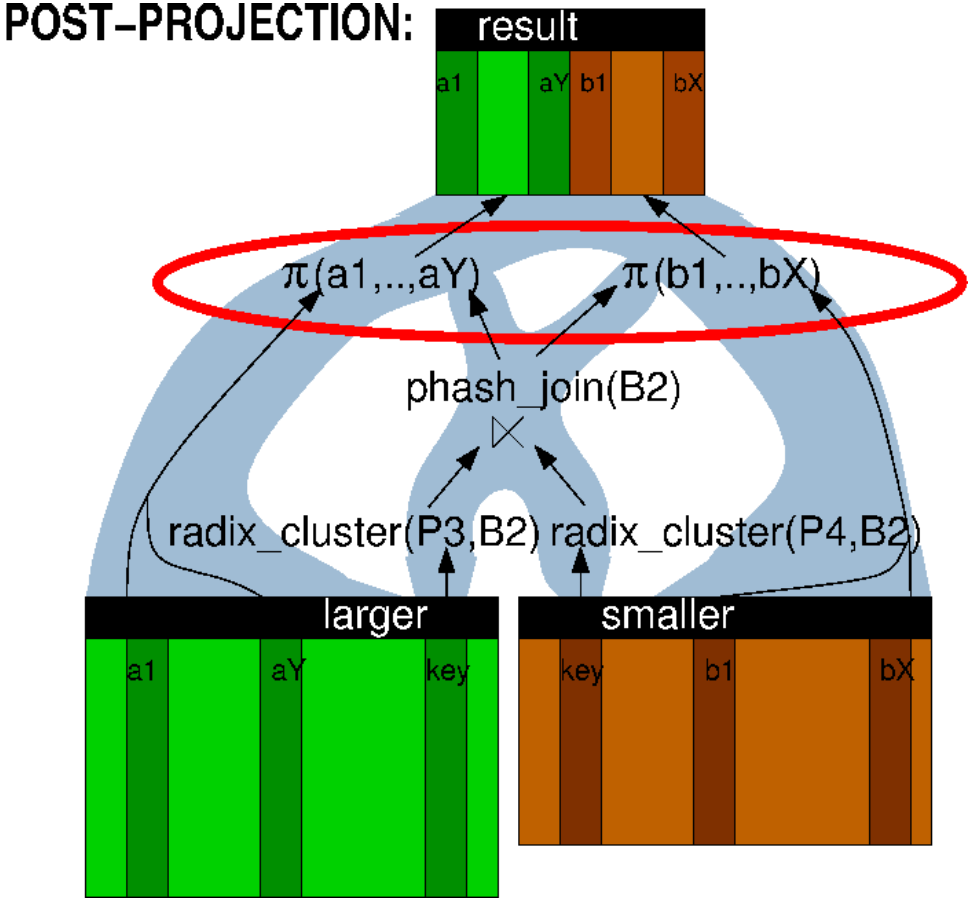
Cache-Conscious Joins
Pre-Projection vs. Post-Projection



Cache-Conscious Joins
Pre-Projection vs. Post-Projection

- Radix-Decluster =
  cache-conscious post-projection





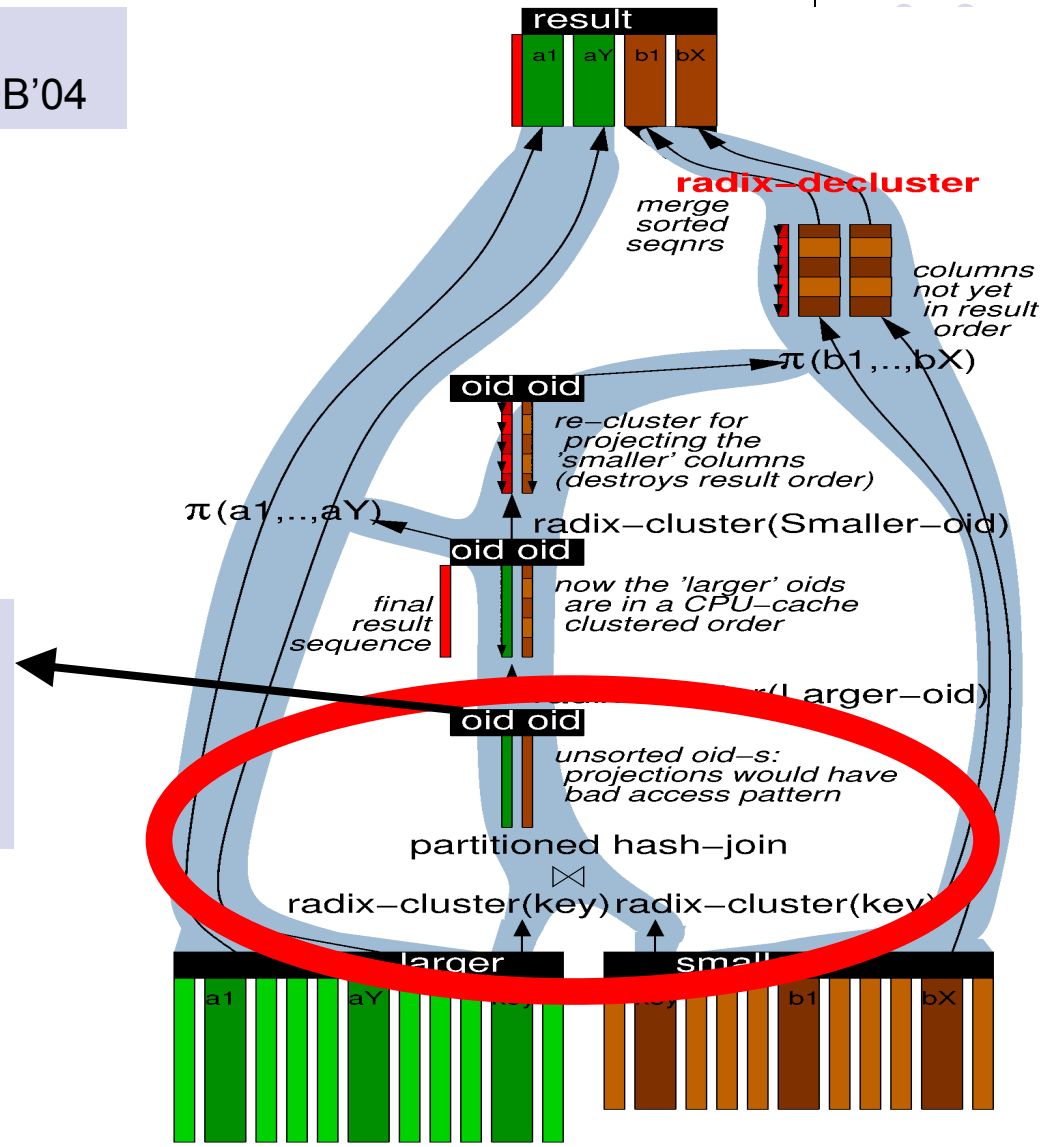
Cache-Conscious Joins
Post-Projection

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

- Partitioned Hash-Join

Join Index!

"Join Indices"
Valduriez, TODS'87

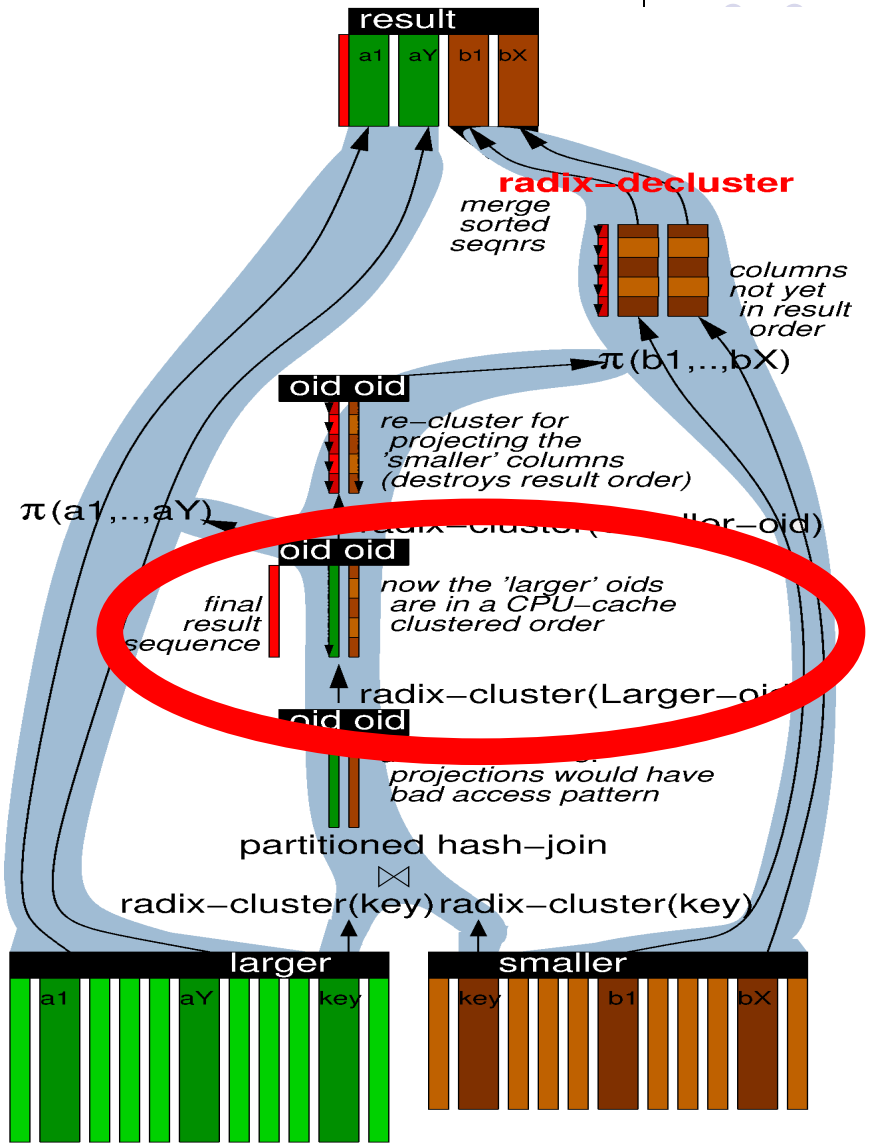


Cache-Conscious Joins

Post-Projection

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

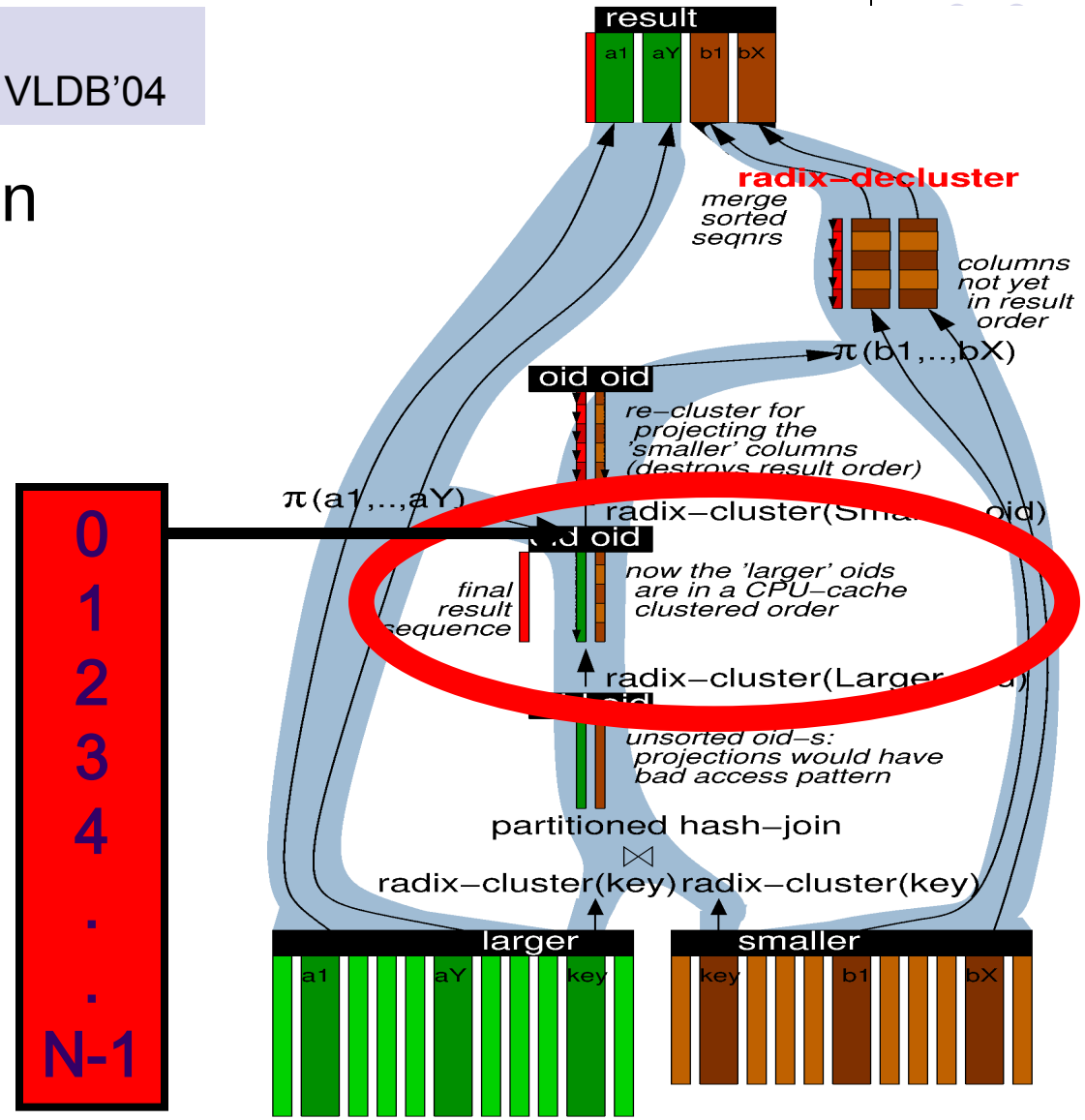
- Partitioned Hash-Join
- Cluster on Left



Cache-Conscious Joins
Post-Projection

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

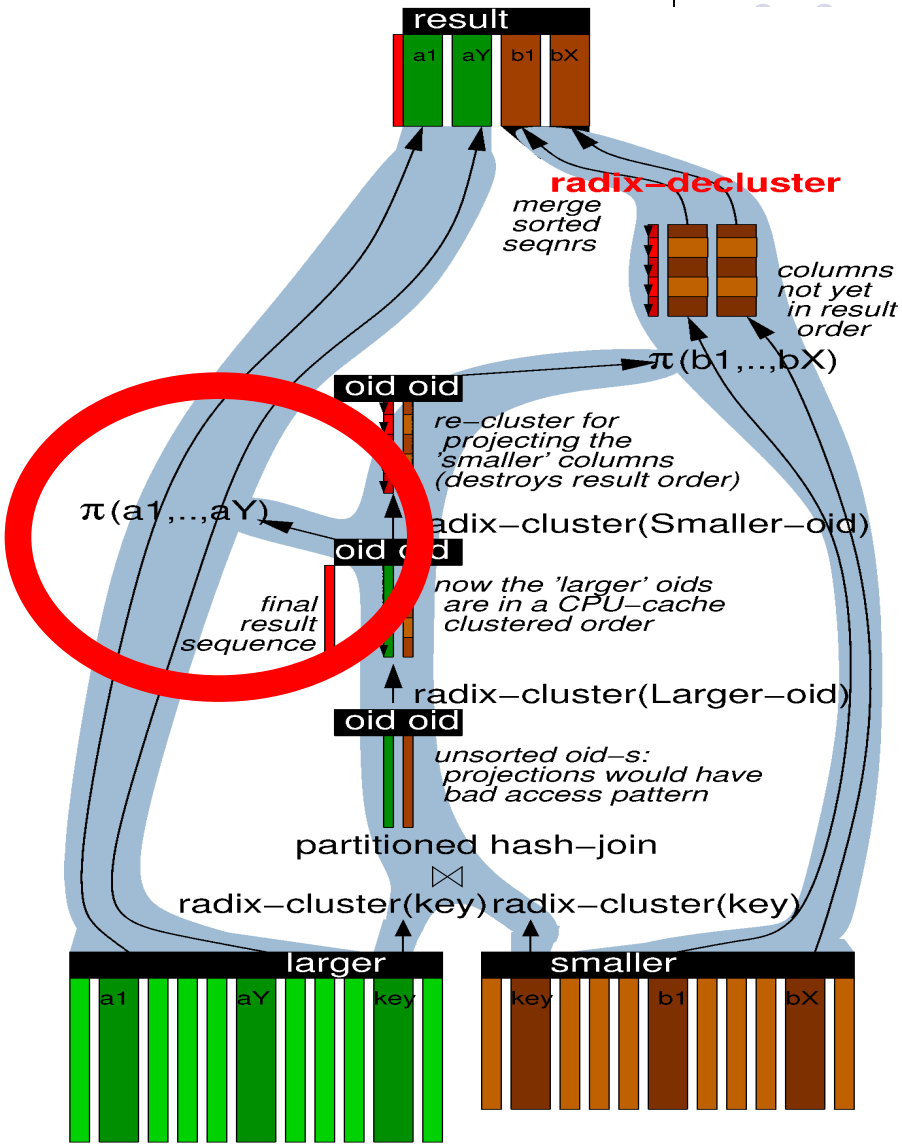
- Partitioned Hash-Join
- Cluster on Left



Cache-Conscious Joins
Post-Projection

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

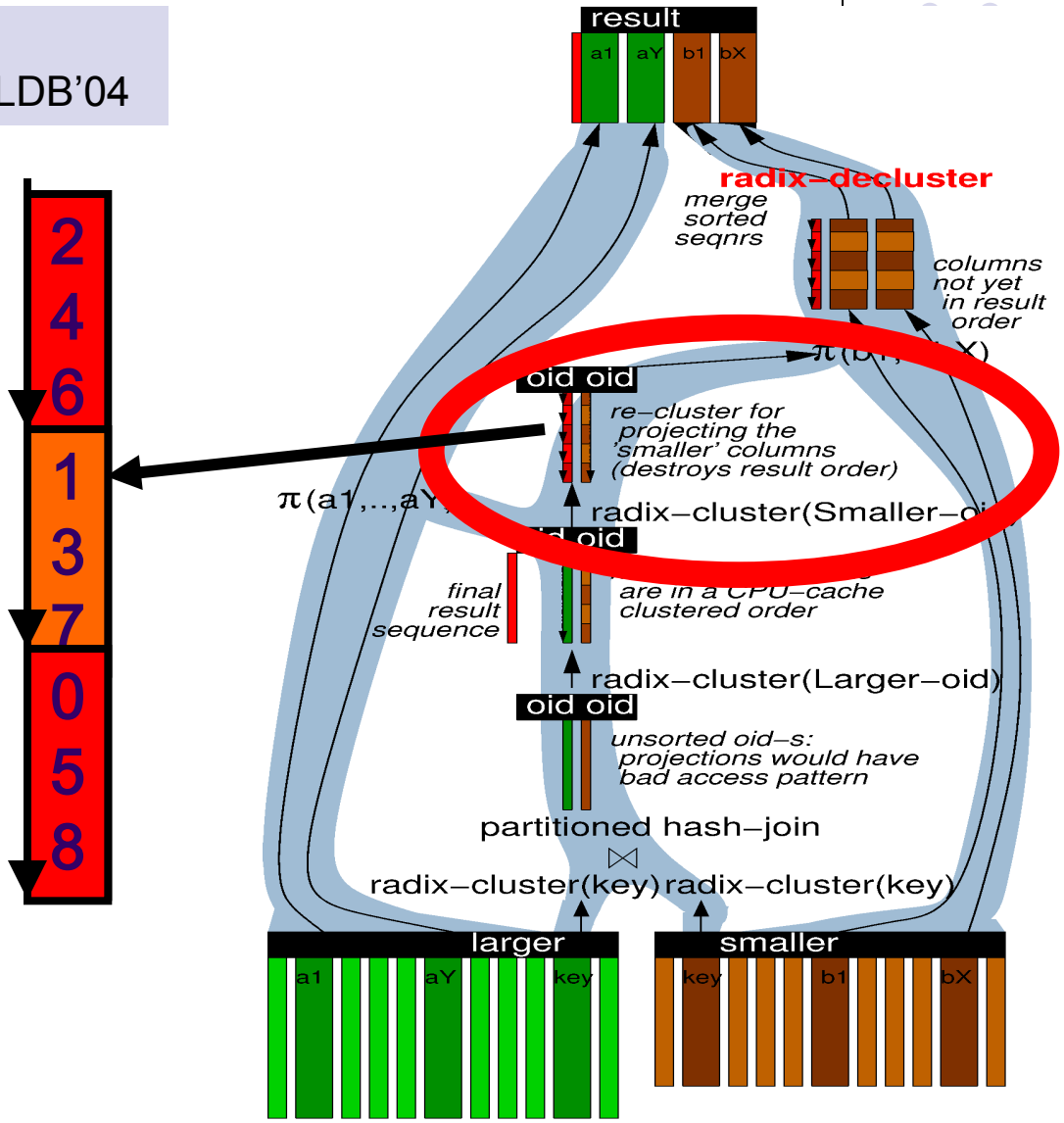
- Partitioned Hash-Join
- Cluster on Left
- Project Left



Cache-Conscious Joins
Post-Projection

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

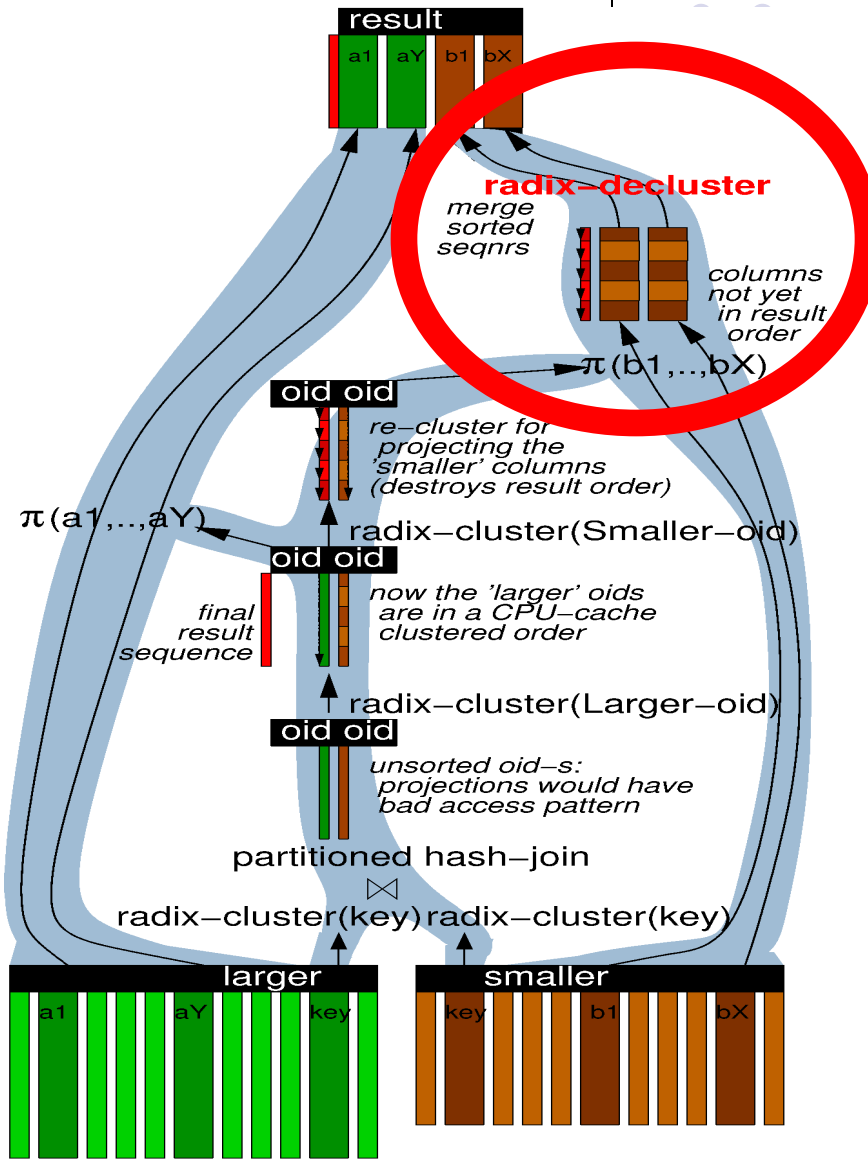
- Partitioned Hash-Join
- Cluster on Left
- Project Left
- Cluster on Right



Cache-Conscious Joins
Post-Projection

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

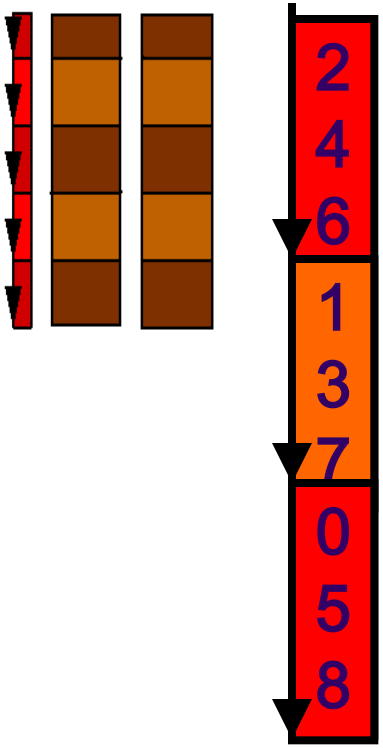
- Partitioned Hash-Join
- Cluster on Left
- Project Left
- Cluster on Right
- Project Right
  - ➔ Radix-Decluster



Cache-Conscious Joins
Radix-Decluster

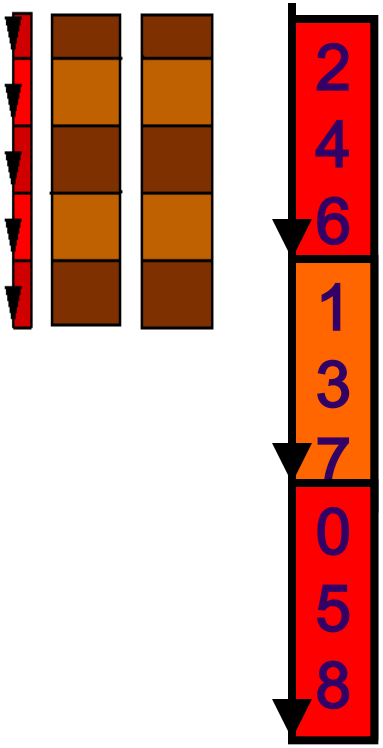
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

- Red column forms a dense domain
  - {0,1,2,…,N-1}
- All subsequences are ordered
  - e.g.
- Task: merge subsequences into dense sequence



Cache-Conscious Joins
Radix-Decluster

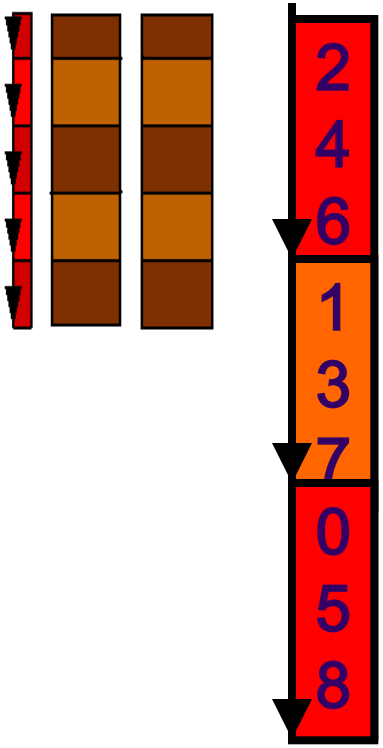
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



- Red column forms a dense domain
  - {0,1,2,…,N-1}
- All subsequences are ordered
  - e.g.
- Task: merge subsequences into dense sequence
  - Approach 1: merge $H=2^B$ lists
    - ☹ cost O(N log(H))
    - many (H) cursors needed ➔cache thrashing

Cache-Conscious Joins
Radix-Decluster

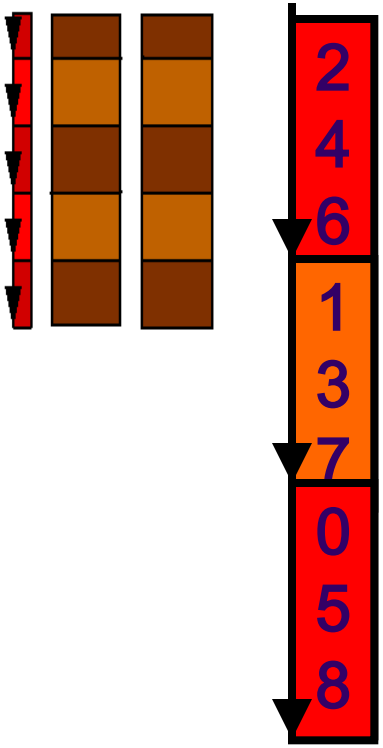
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



- Red column forms a dense domain
  - {0,1,2,…,N-1}
- All subsequences are ordered
  - e.g.
- Task: merge subsequences into dense sequence
  - Approach 1: merge $H=2^B$ lists
    - ☹ cost O(N log(H)),
    - many (H) cursors needed ➔cache thrashing
  - Approach 2: insert by position
    - ☹ many (H) sparse passes over the result ➔no cache reuse

Cache-Conscious Joins
Radix-Decluster

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



- Red column forms a dense domain
  - {0,1,2,…,N-1}
- All subsequences are ordered
  - e.g.
- Task: merge subsequences into dense sequence
  - Approach 1: merge $H=2^B$ lists
    - ☹ cost O(N log(H))
    - many (H) cursors needed ➔cache thrashing
  - Approach 2: insert by position
    - ☹ many (H) sparse passes over the result ➔ no cache reuse
  - Radix-Decluster: insert by position with sliding window

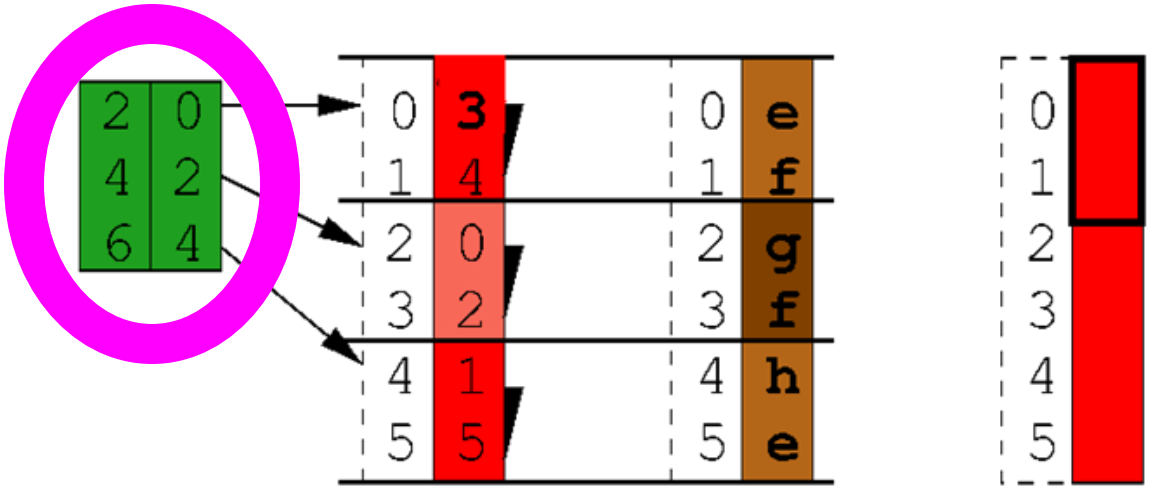
Cache-Conscious Joins
Radix-Decluster In Action

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

Cache-Conscious Joins
Radix-Decluster In Action

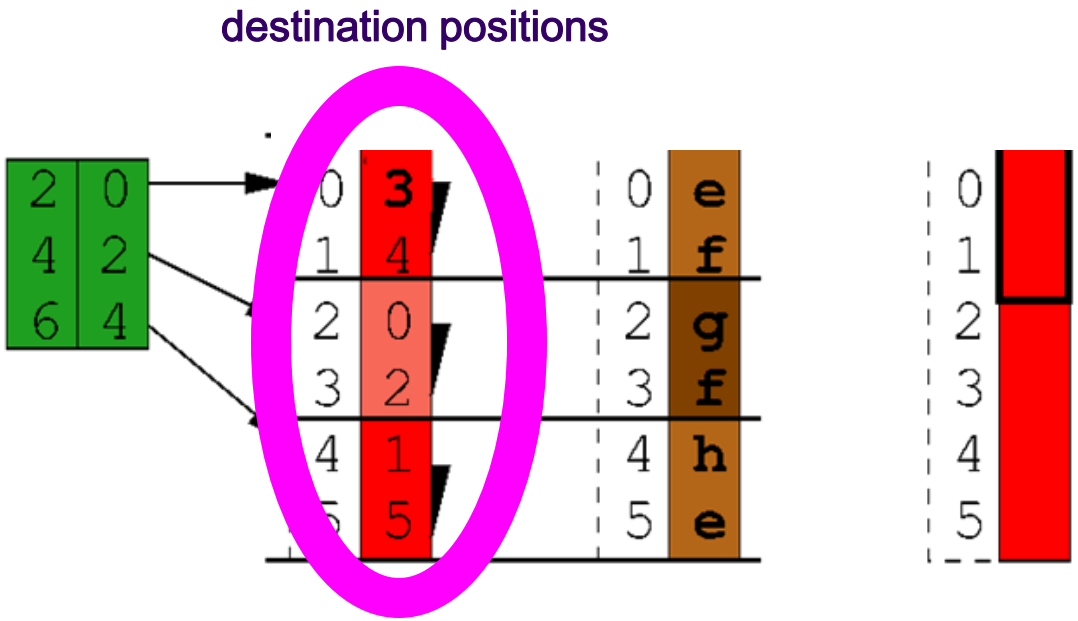
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

3 clusters



Cache-Conscious Joins
Radix-Decluster In Action

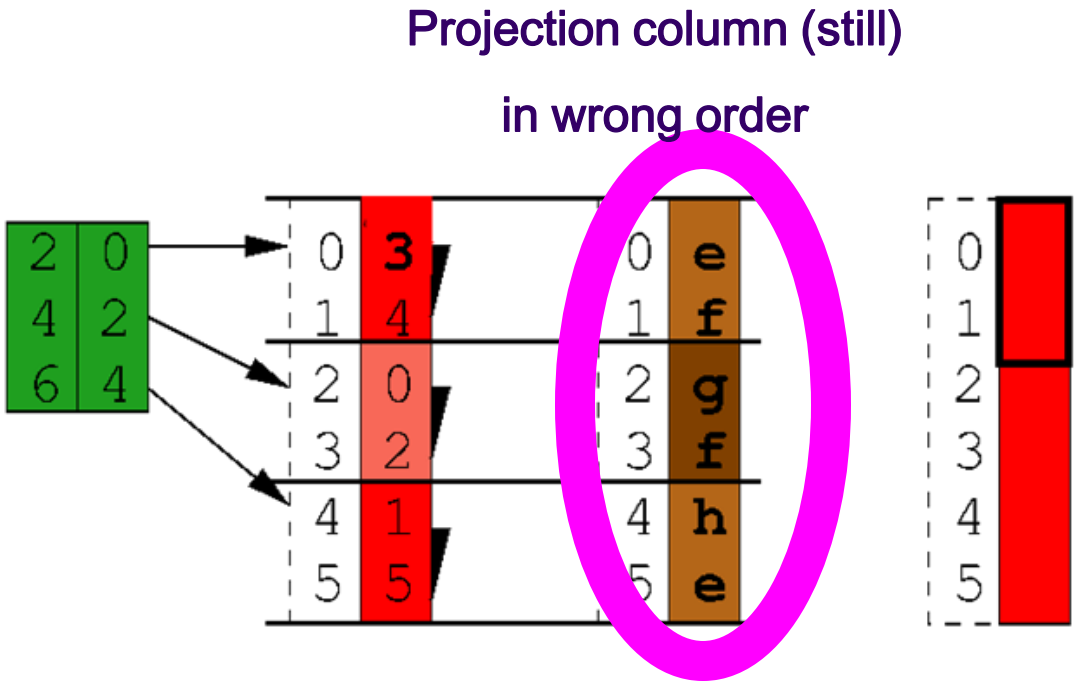
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



Cache-Conscious Joins

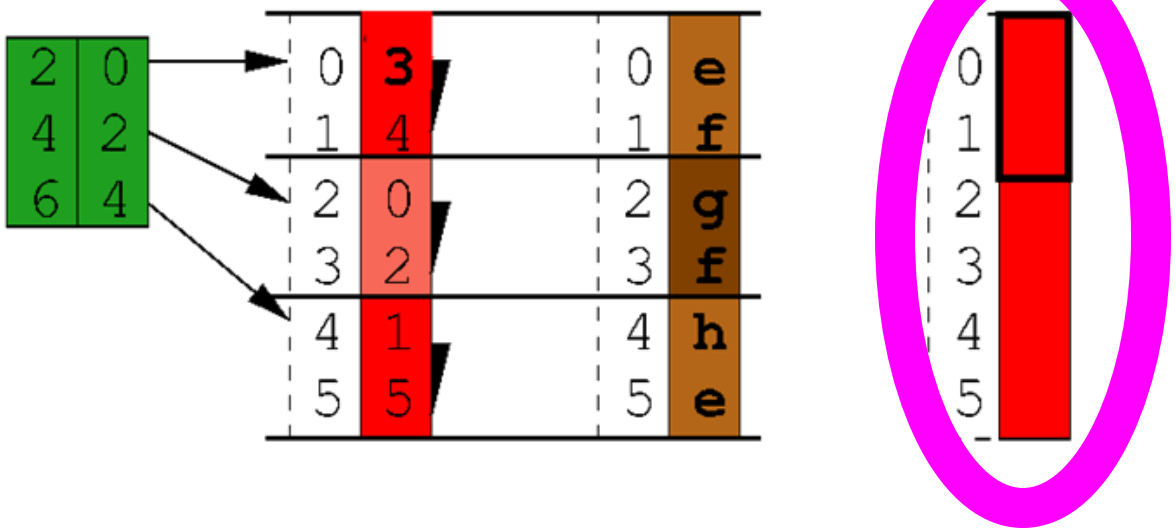
Radix-Decluster In Action

•“Cache-Conscious Radix-Decluster Projections”, Manegold, Boncz, Nes, VLDB’04



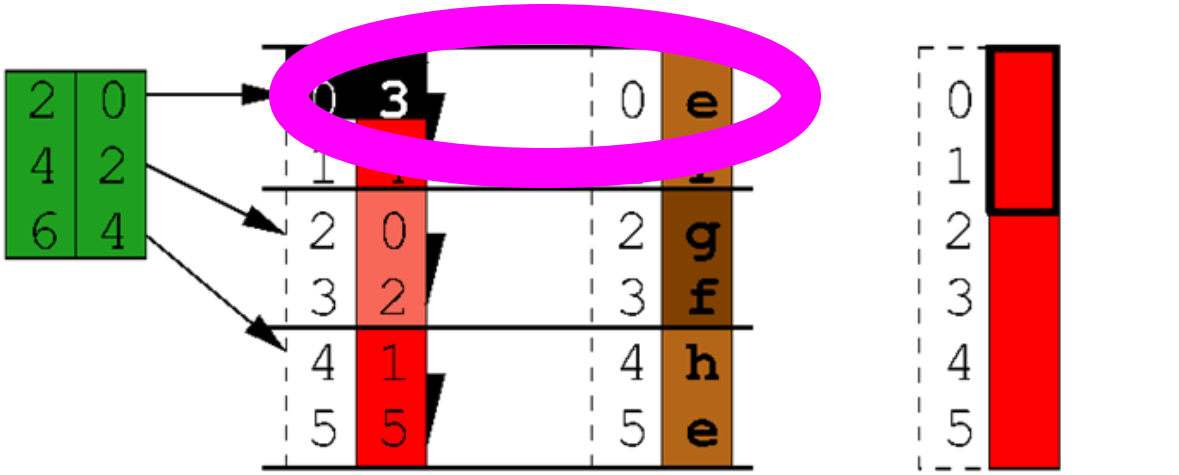
Cache-Conscious Joins
Radix-Decluster In Action

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



Cache-Conscious Joins
Radix-Decluster In Action

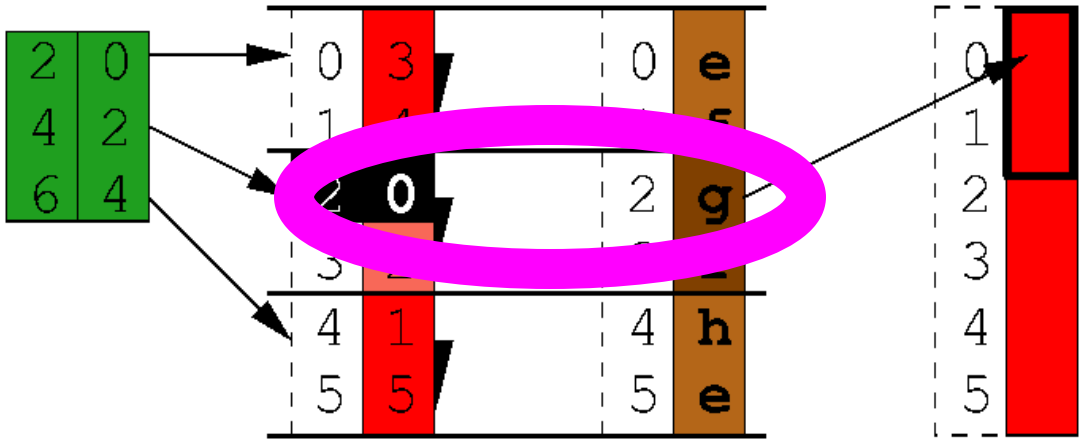
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



next cluster (3 is outside window)

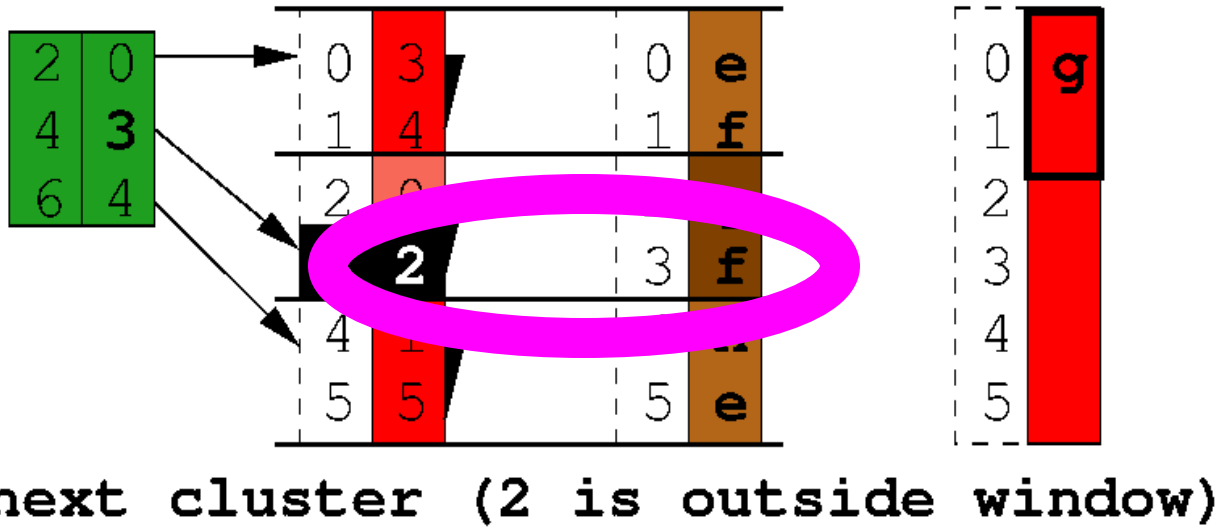
Cache-Conscious Joins
Radix-Decluster In Action

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



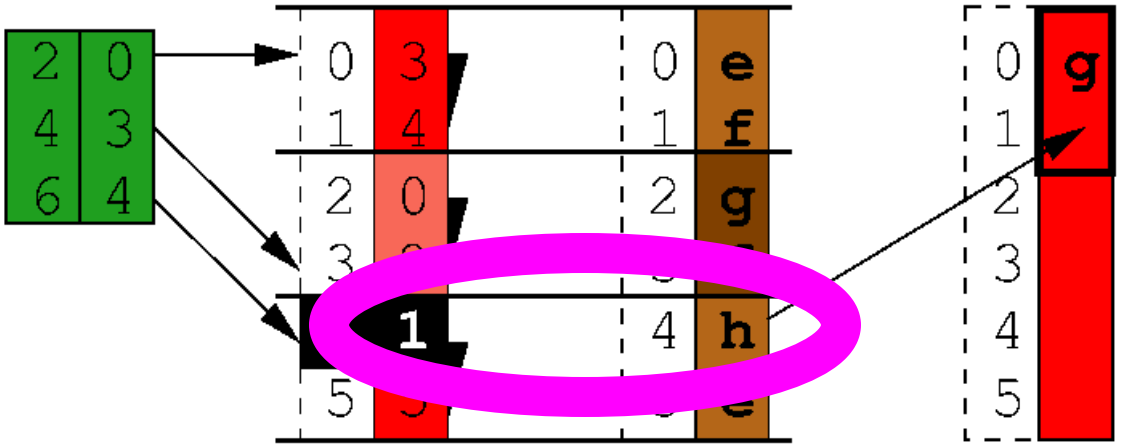
Cache-Conscious Joins
Radix-Decluster In Action

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



Cache-Conscious Joins
Radix-Decluster In Action

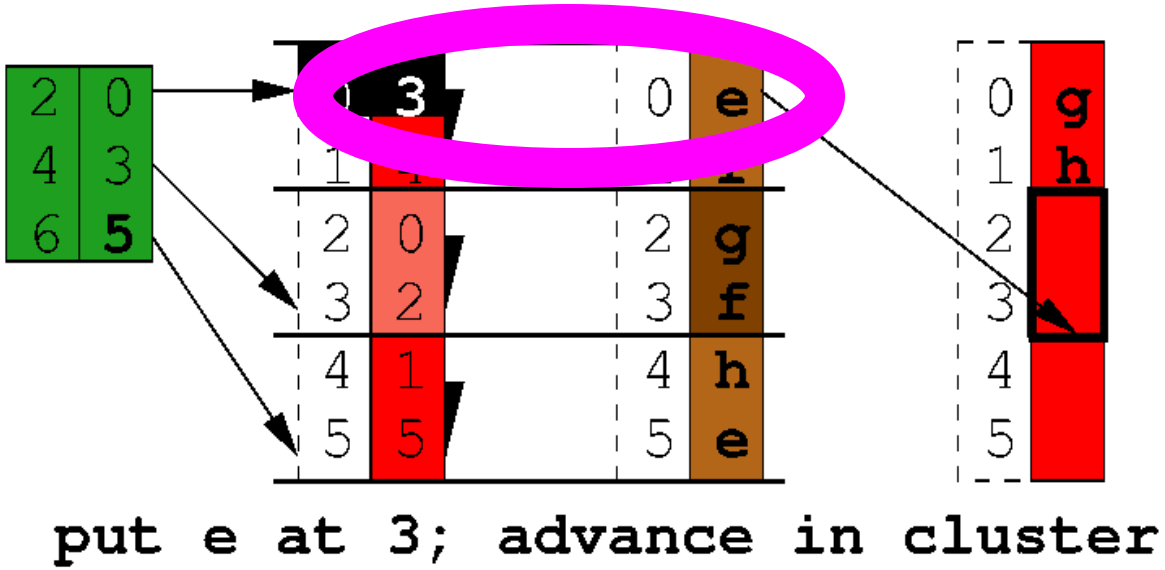
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



put h at 1; advance window; reset

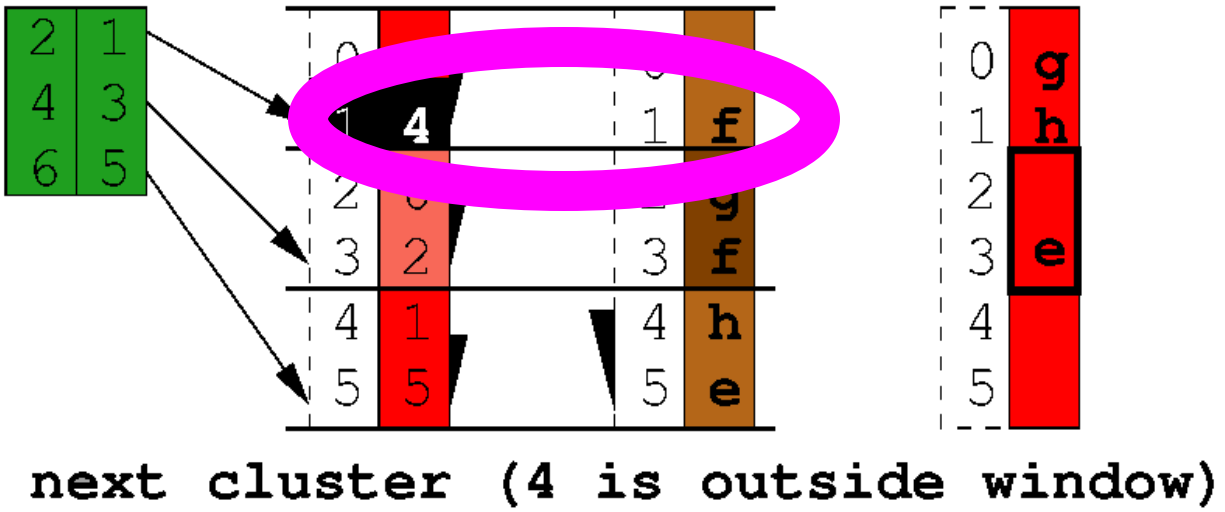
Cache-Conscious Joins
Radix-Decluster In Action

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



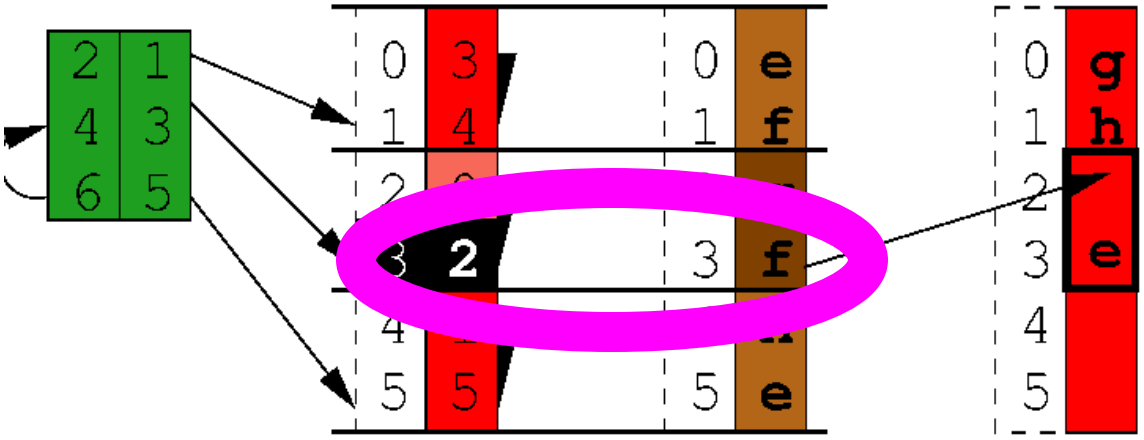
Cache-Conscious Joins
Radix-Decluster In Action

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



Cache-Conscious Joins
Radix-Decluster In Action

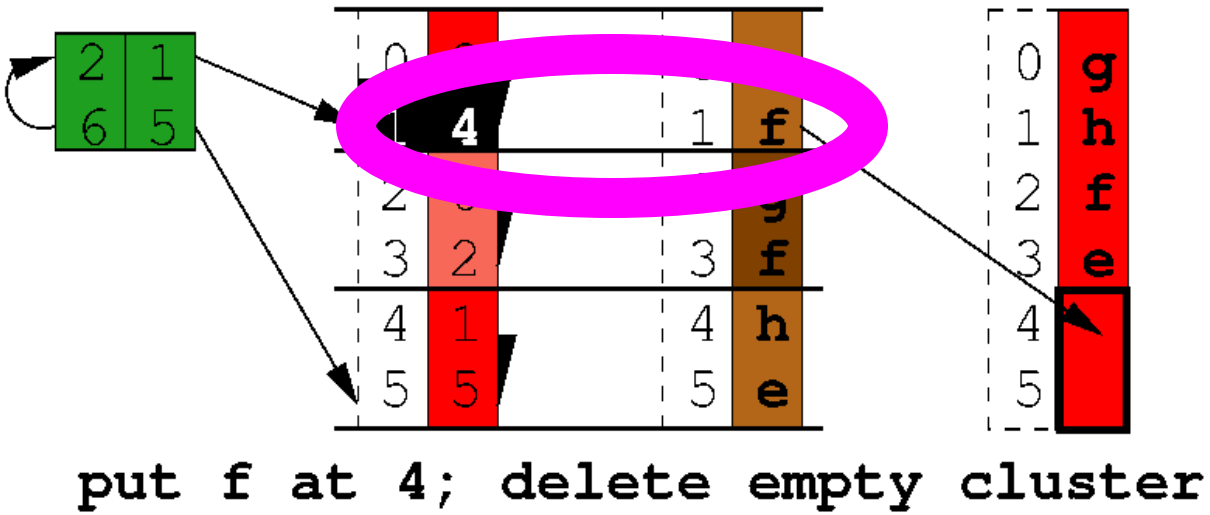
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



put f at 2; delete empty cluster; advance window; reset

Cache-Conscious Joins
Radix-Decluster In Action

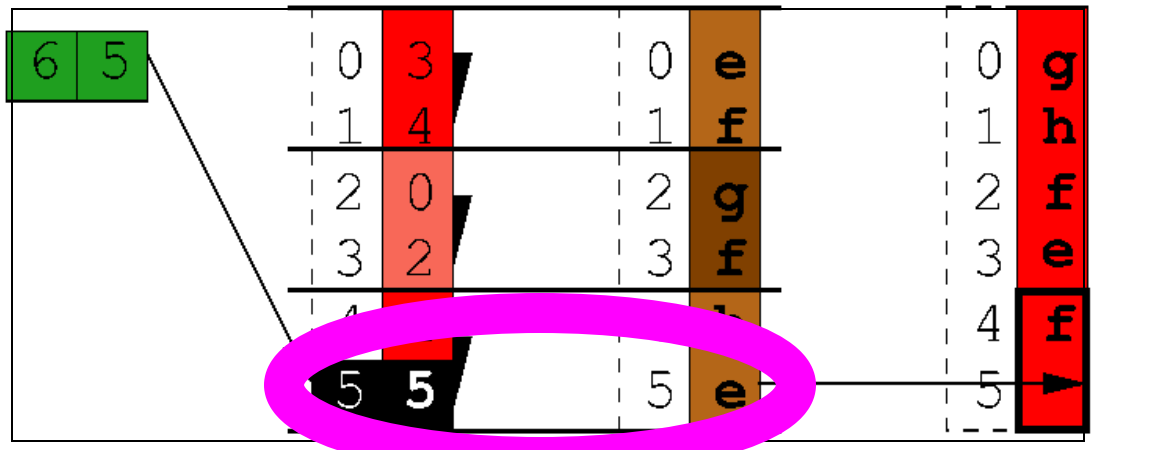
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



put f at 4; delete empty cluster

Cache-Conscious Joins
Radix-Decluster In Action

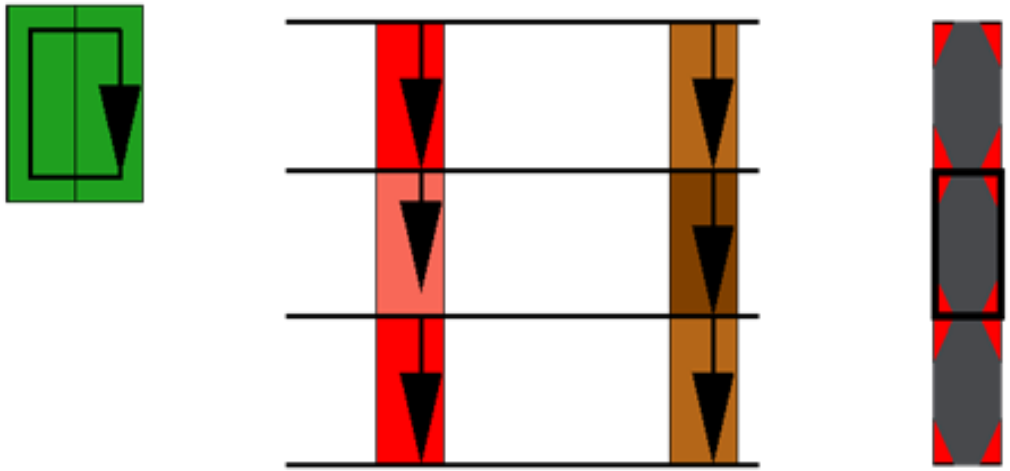
•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04



Cache-Conscious Joins
Radix-Decluster Memory Access Pattern

•"Cache-Conscious Radix-Decluster Projections", Manegold, Boncz, Nes, VLDB'04

- Random access only to sliding window(<< cache size)



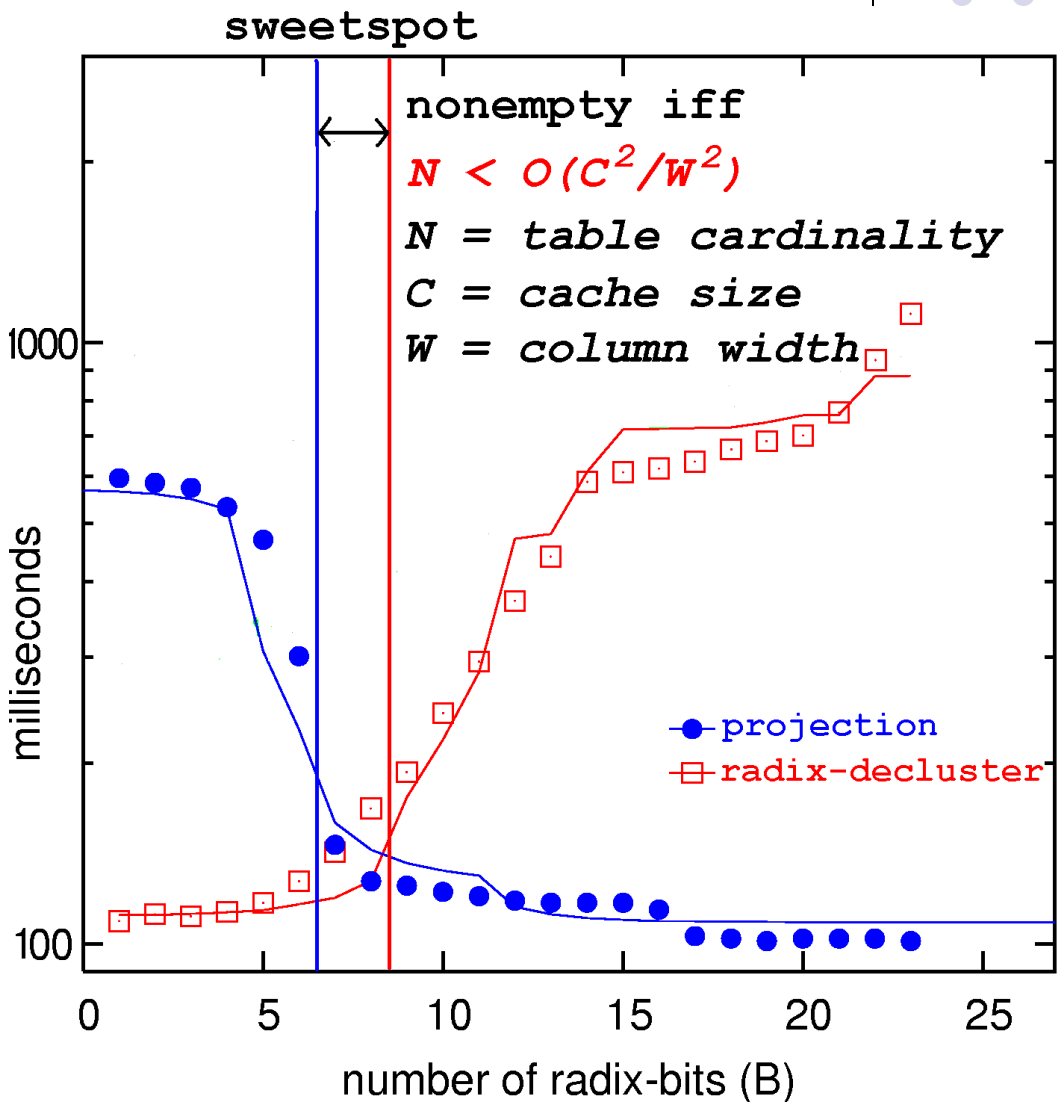
- Only compulsary misses ➔ cache-conscious

Cache-Conscious Joins
Radix-Decluster Performance Tradeoff

- Radix-Decluster prefers small W
  (i.e. vertical fragmentation)

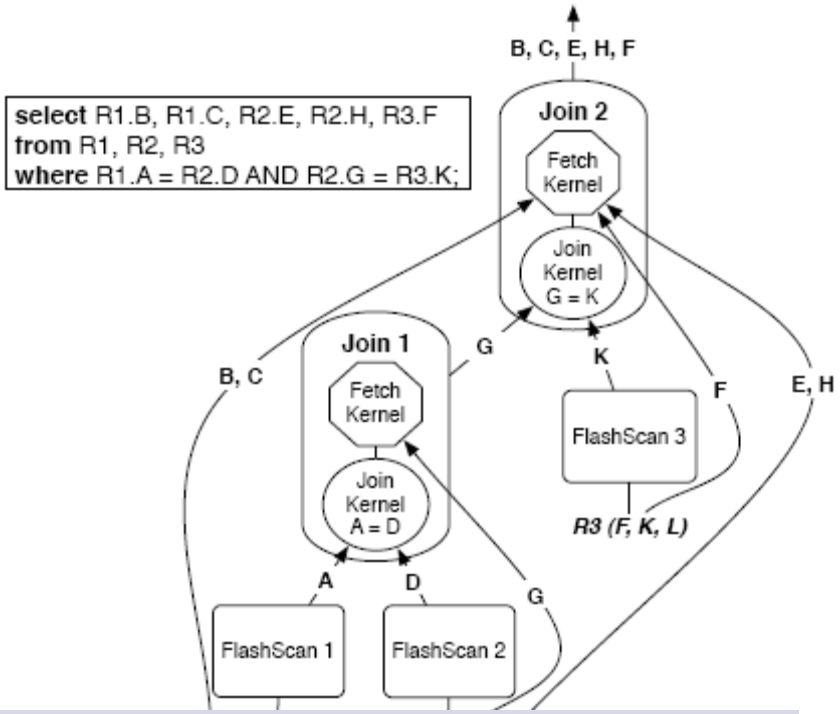
Read Also:
- Jive Join
- Slam Join

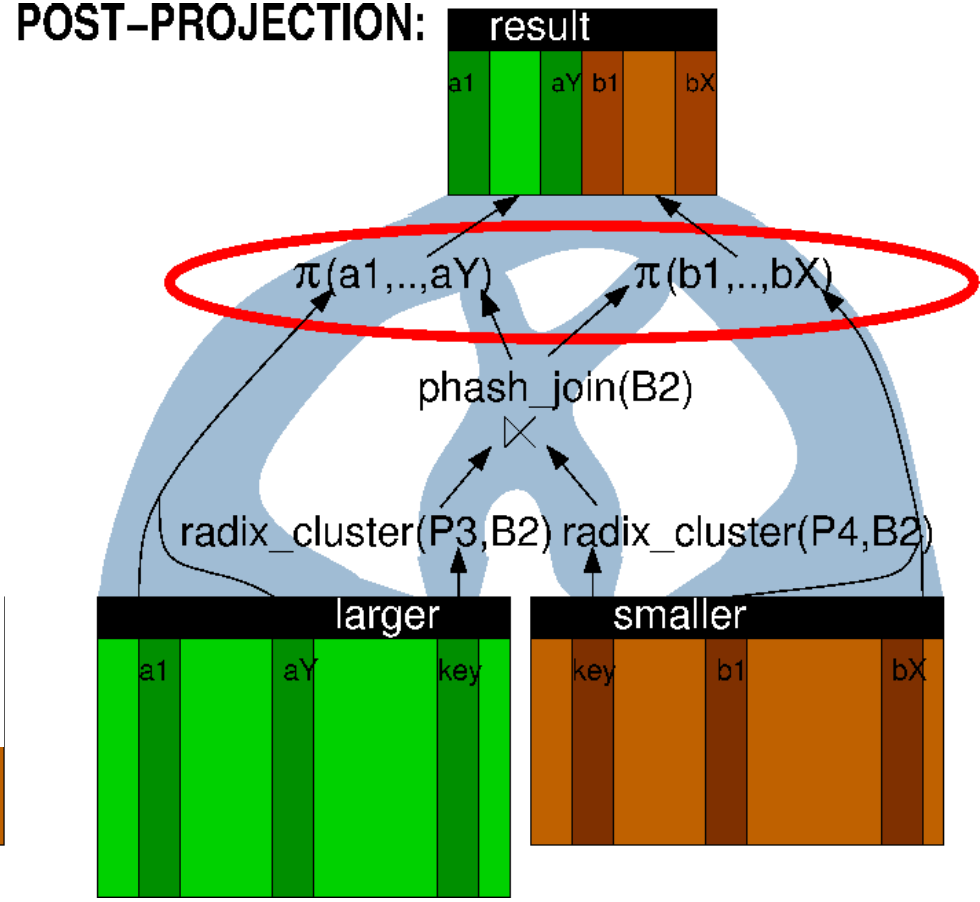
"Fast Joins Using Join Indices"
Ross, Lei, VLDBJ'98



Cache-Conscious Joins
Pre-Projection vs. Post-Projection

4.1 FlashJoin Overview

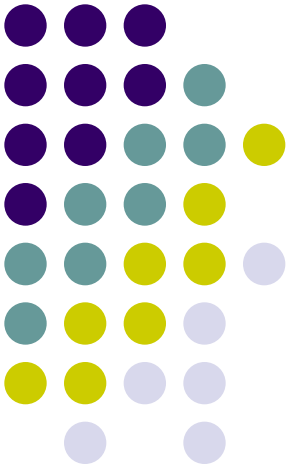




"Query Processing Techniques for Solid State Drives" Tsirogiannis, Harizopoulos,Shah, Wiener, Graefe, SIGMOD'09

Column-Oriented Database Systems

VLDB 2009 Tutorial



vectorwise

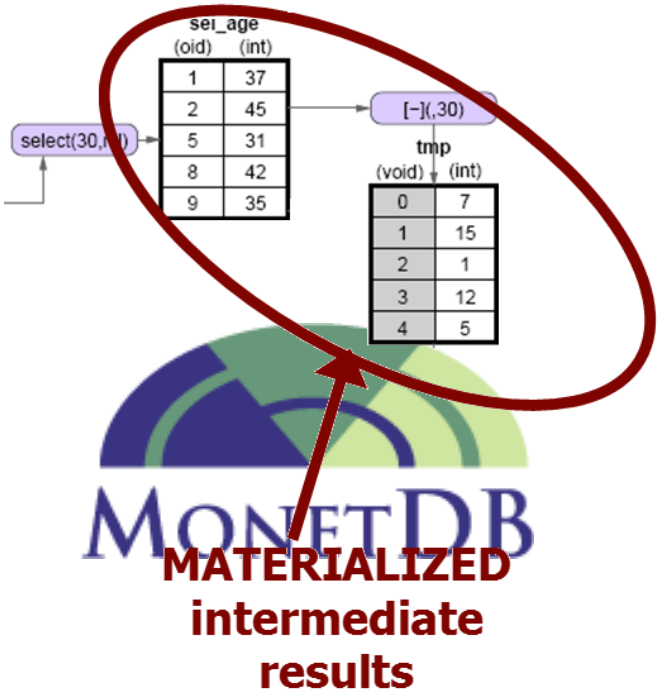
"MonetDB/X100"

vectorized query processing

MonetDB spin-off: MonetDB/X100

vectorwise

Materialization vs Pipelining



"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

vectorwise

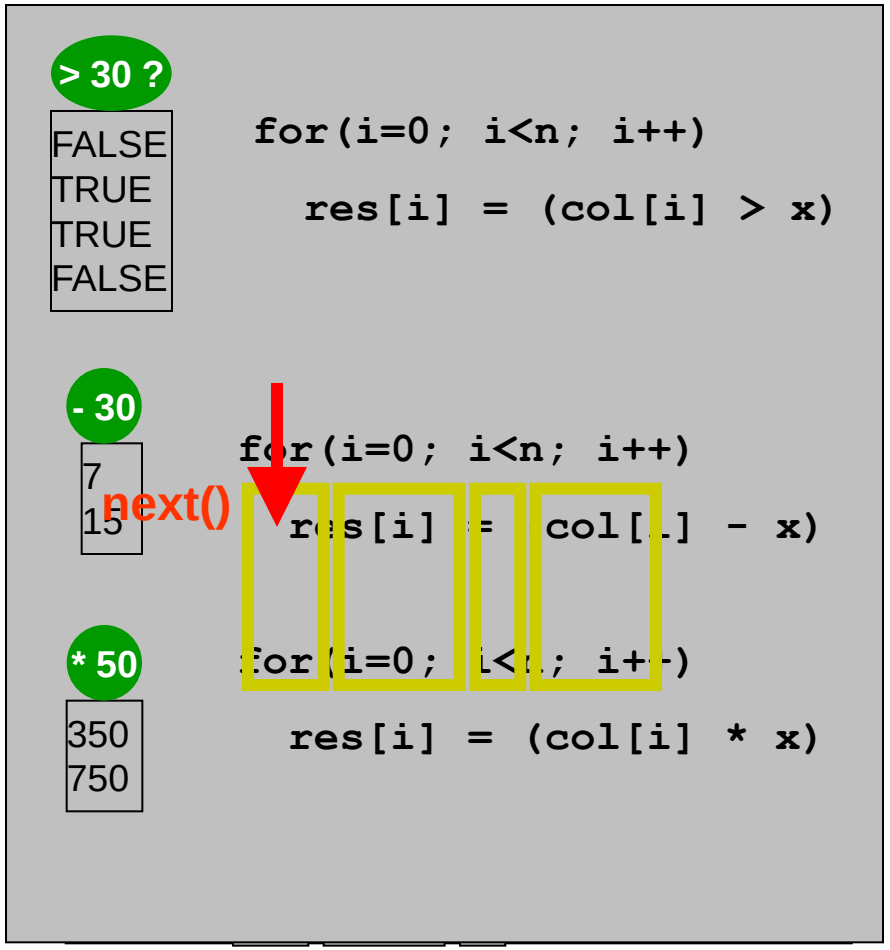
"Vectorized In Cache Processing"

vector = array of ~100

processed in a tight loop

CPU cache Resident

- automatic SIMD



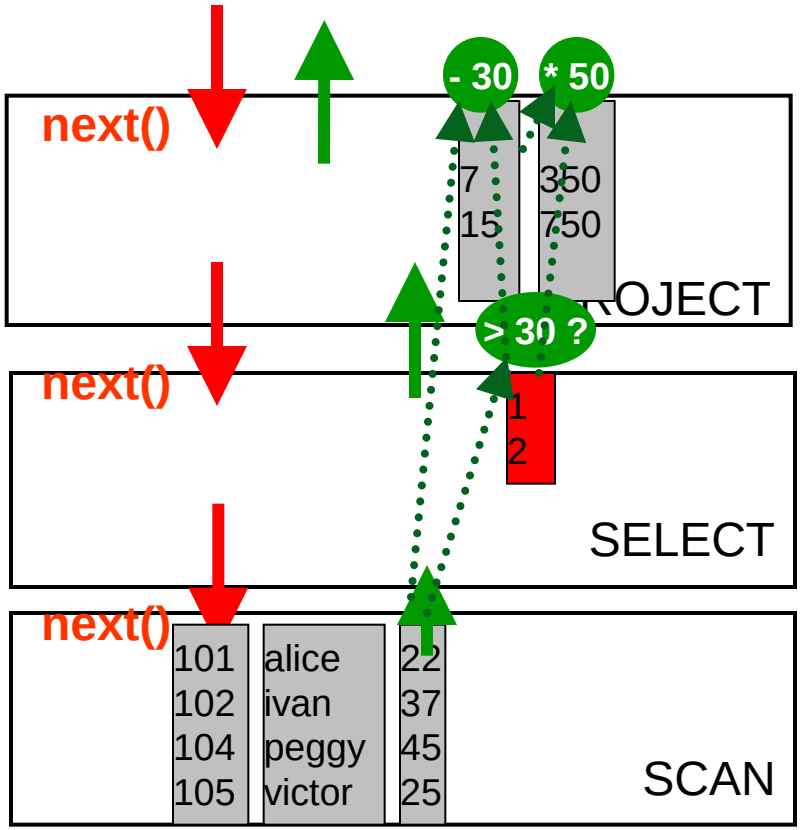
"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

vectorwise

```
map_mul_flt_val_flt_col(
    float *res,
    int*   sel,
    float  val,
    float *col, int n)

{
    if (many selected and narrow)
      for(int i=0; i<n; i++) // SIMD!!
            res[i] = val * col[i];
    else
        for(int i=0; i<n; i++)
            res[i] = val * col[sel[i]];
}
```

"push selections up" to get SIMD



"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

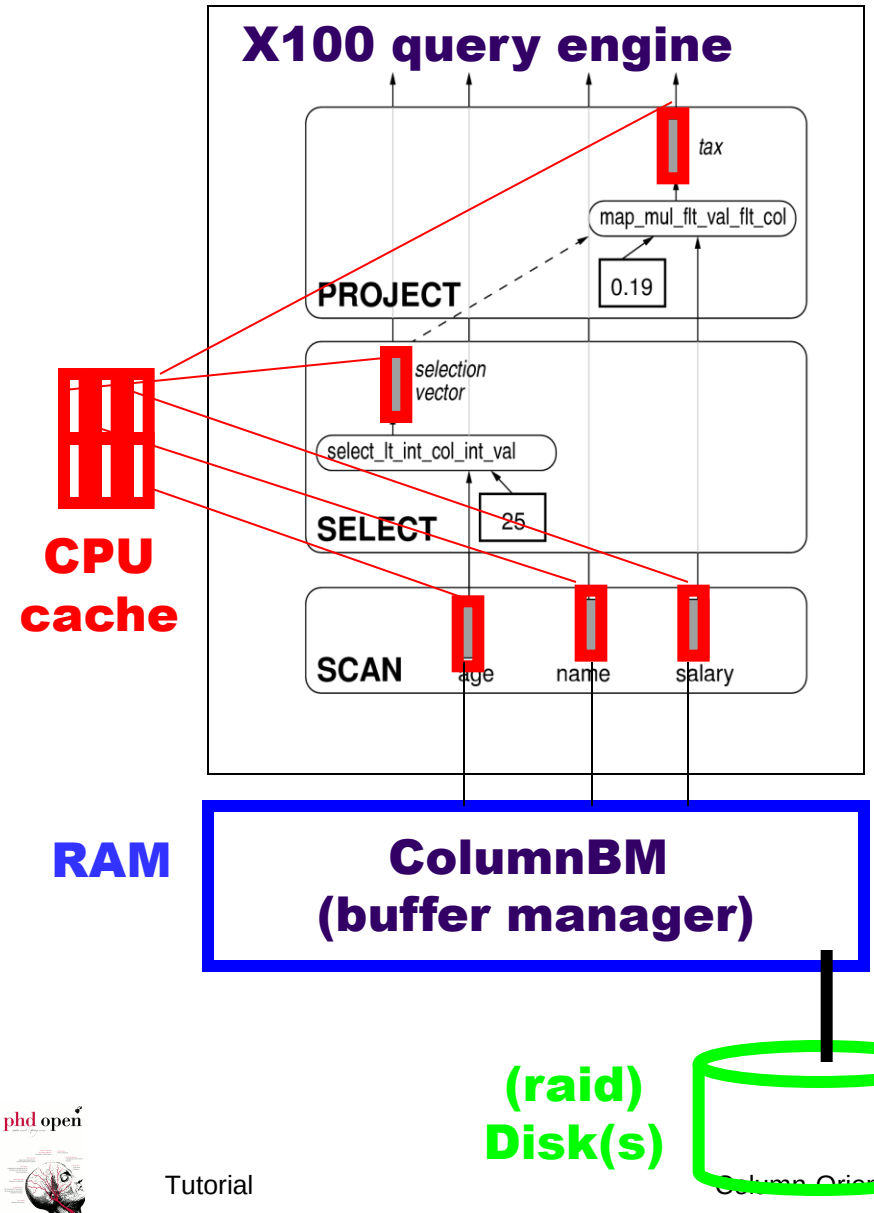
MonetDB/X100

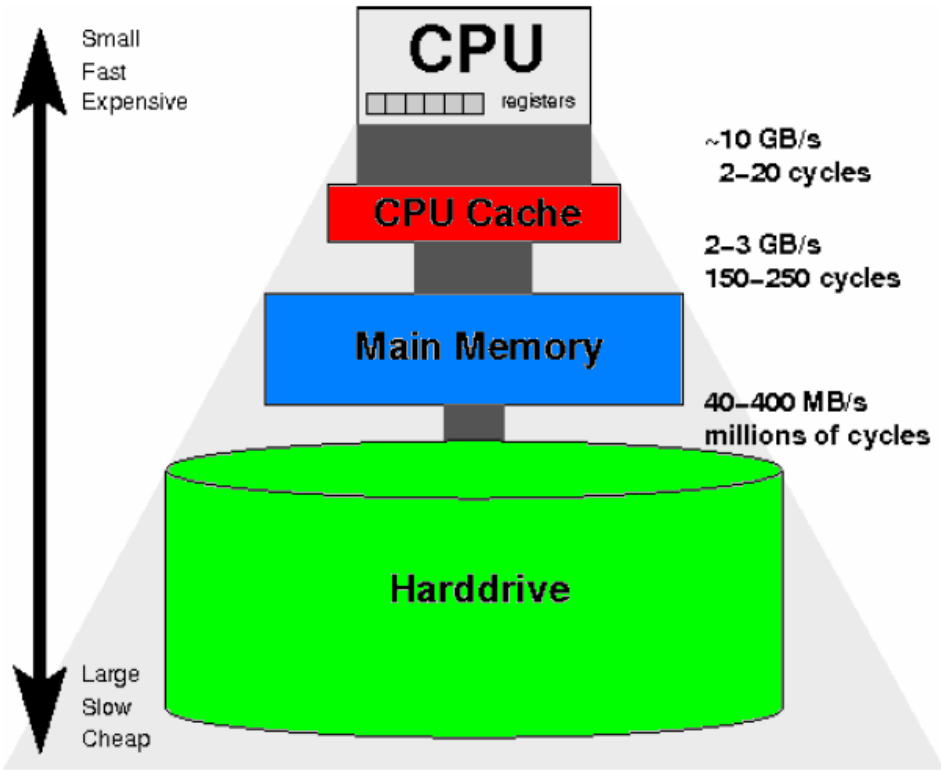
- Both efficiency
  - Vectorized primitives
- and scalability..
  - Pipelined query evaluation

C program:	0.2s
MonetDB/X100:	0.6s
MonetDB:	3.7s
MySQL:	26.2s
DBMS "X":	28.1s

"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

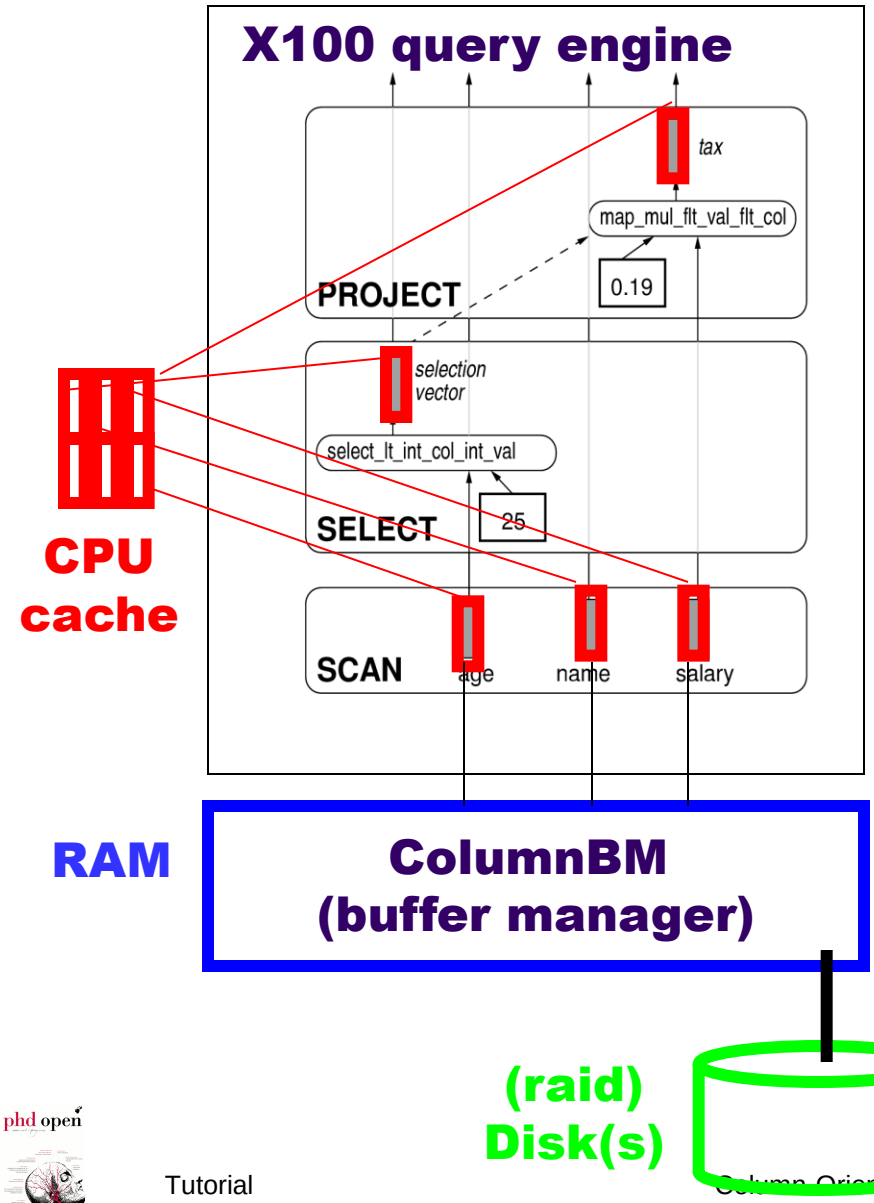
Memory Hierarchy





"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

Memory Hierarchy



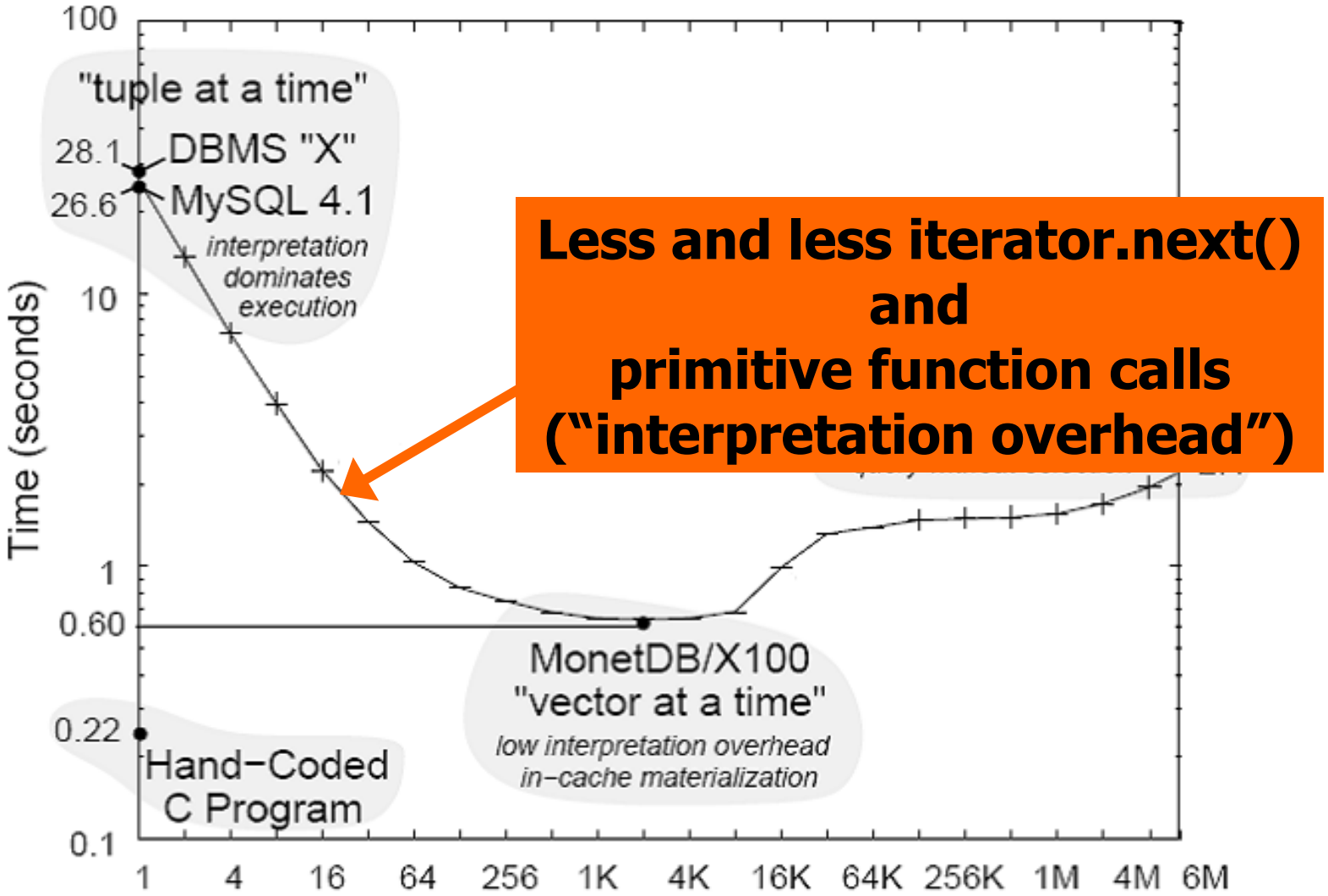
Vectors are only the in-cache representation

RAM & disk representation might actually be different

(vectorwise uses both PAX & DSM)

"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

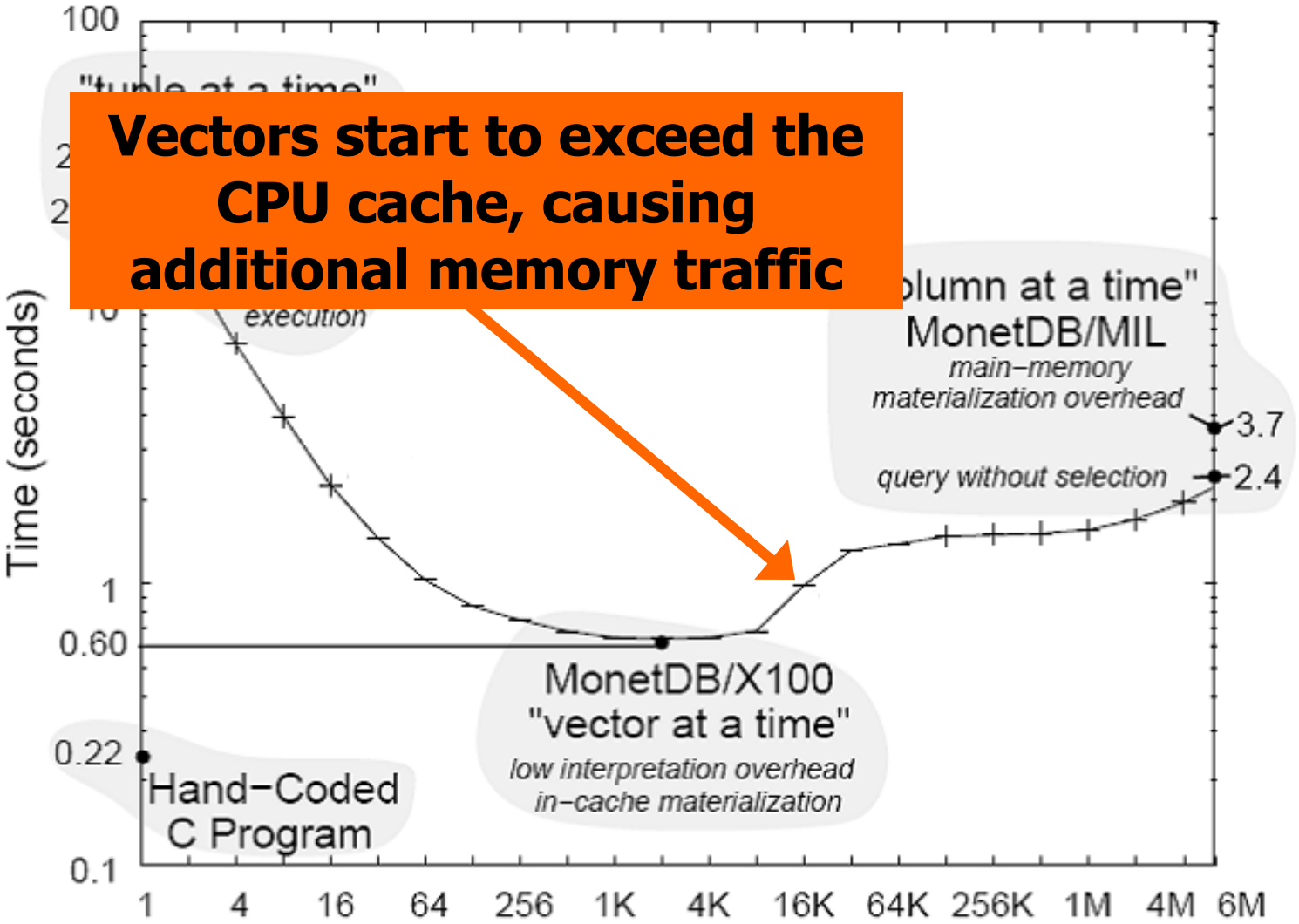
Varying the Vector size



Less and less iterator.next() and primitive function calls ("interpretation overhead")

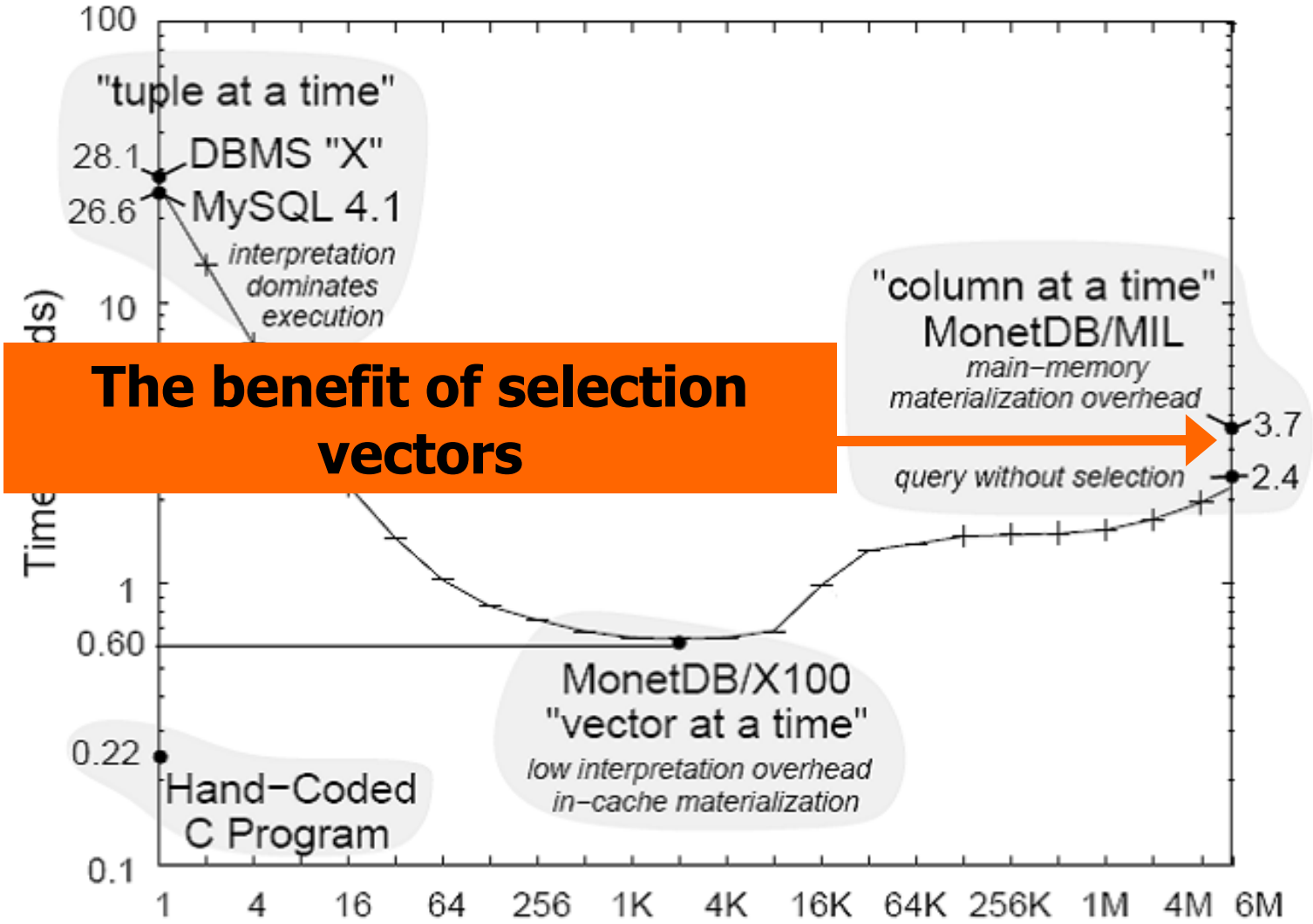
"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

Varying the Vector size



"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

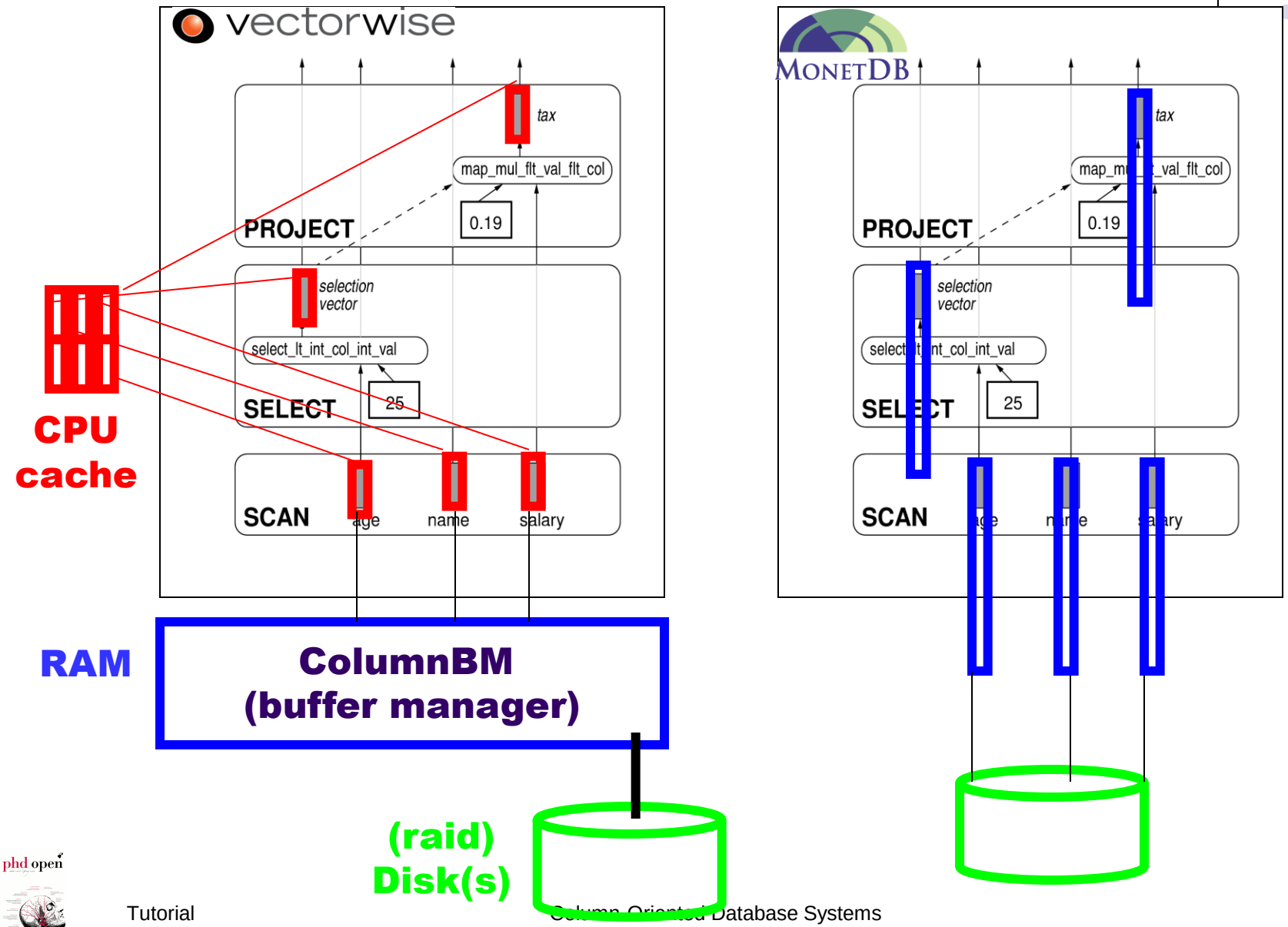
Varying the Vector size



The benefit of selection vectors

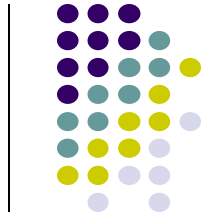
"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

MonetDB/MIL materializes columns



"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

Benefits of Vectorized Processing



"Buffering Database Operations for Enhanced Instruction Cache Performance" Zhou, Ross, SIGMOD'04

"Block oriented processing of relational database operations in modern computer architectures" Padmanabhan, Malkemus, Agarwal, ICDE'01

- **100x less Function Calls**
  - iterator.next(), primitives
- **No Instruction Cache Misses**
  - High locality in the primitives
- **Less Data Cache Misses**
  - Cache-conscious data placement
- **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

"Balancing Vectorized Query Execution with Bandwidth Optimized Storage " Zukowski, CWI 2008

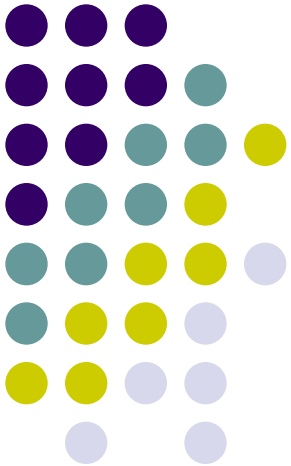
Vectorizing Relational Operators

- Project
- Select
  - Exploit selectivities, test buffer overflow
- Aggregation
  - Ordered, Hashed
- Sort
  - Radix-sort nicely vectorizes
- Join
  - Merge-join + Hash-join

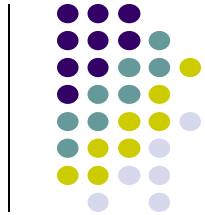
Column-Oriented Database Systems

Tutorial

Conclusion

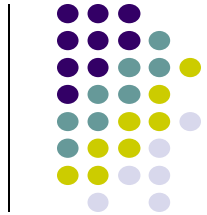


Summary (1/2)



- Columns and Row-Stores: different?
  - No fundamental differences
  - Can current row-stores simulate column-stores now?
    - not efficiently: row-stores need change
  - On disk layout vs execution layout
    - actually independent issues, on-the-fly conversion pays off
    - column favors sequential access, row random
  - Mixed Layout schemes
    - Fractured mirrors
    - PAX, Clotho
    - Data morphing

Summary (2/2)



- Crucial Columnar Techniques
  - Storage
    - Lean headers, sparse indices, fast positional access
  - Compression
    - Operating on compressed data
    - Lightweight, vectorized decompression
  - Late vs Early materialization
    - Non-join: LM always wins
    - Naïve/Invisible/Jive/Flash/Radix Join (LM often wins)
  - Execution
    - Vectorized in-cache execution
    - Exploiting SIMD