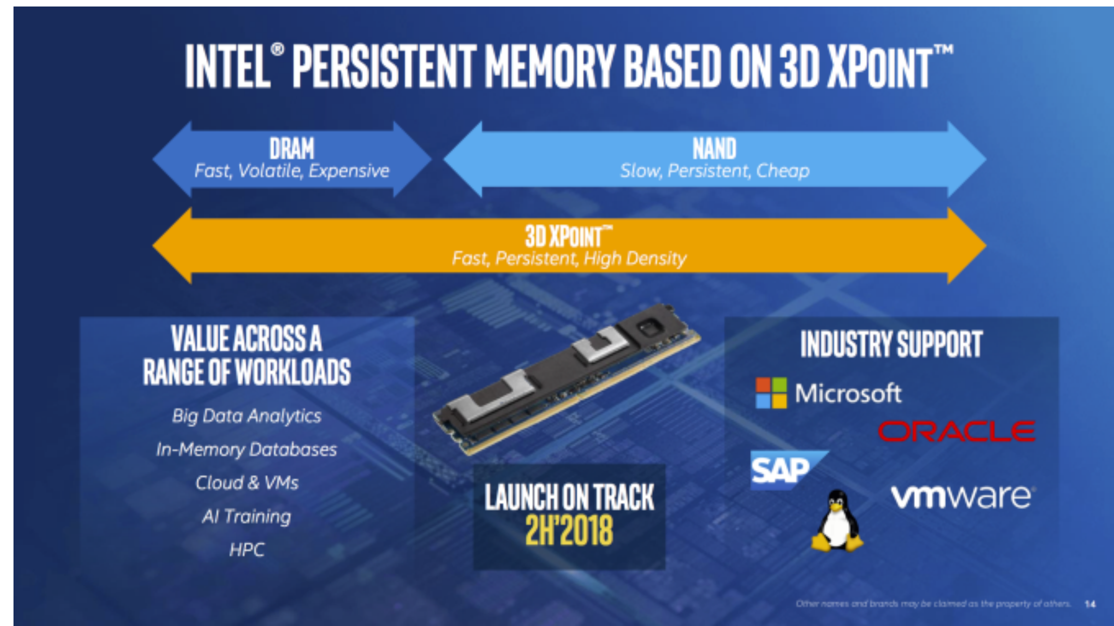
BDCC: Exploiting Fine-Grained Persistent Memories for OLAP

Peter Boncz

Centrum Wiskunde & Informatica

NVRAM

- System integration:
  - NVMe: block devices on the PCIe bus
  - NVDIMM: persistent RAM, byte-level access
- Low latency
  - Lower than Flash,
  - close to DRAM
  - Asymmetric (r<w)
- Fine-grained access
  - 512byte for NMVe
  - NVDIMM: cache-line



NVRAM: DB impact

- Back to the 5-minute rule:
  - Restoring old balance of latency and bandwidth?
- Many challenges in OLTP
  - index structures, (in-page) logging
  - ensure consistency, prevent leakage, control wear

➔ what about OLAP?

Should we re-think warehouse storage for low-latency persistent memories?

CWI Centrum Wiskunde & Informatica

VLDB Journal 2016
Volume 25, Issue 3
p. 291- 316

BDCC: Bitwise Dimensional Co-Clustering

Stephan Baumann · Peter Boncz · Kai-Uwe Sattler

Received: date / Accepted: date

Abstract Analytical workloads in data warehouses often include heavy joins where queries involve multiple fact tables in addition to the typical star-patterns, dimensional grouping and selections. In this paper we propose a new processing and storage framework called *Bitwise Dimensional Co-Clustering* (BDCC) that avoids replication and thus keeps updates fast, yet is able to accelerate all these foreign key joins, efficiently support grouping and pushes down most dimensional selections. The core idea of BDCC is to cluster each table on a mix of dimensions, each possibly derived from attributes imported over an incoming foreign key and this way creating foreign key connected tables with partially shared clusterings. These are later used to accelerate any join between two tables that have some dimension in common; and additionally permit to push down and propagate selections (reduce I/O) and accelerate aggregation and ordering operations. Besides the general framework, we describe an algorithm to derive such a physical co-clustering database automatically and describe query processing and query optimization techniques that can easily be fitted into existing relational engines. We present an experimental evaluation on the TPC-H benchmark in the Vectorwise system, showing that co-clustering can significantly enhance its already high performance and at the same time significantly reduce the memory consumption of the system.

S. Baumann
Technische Universität Ilmenau
E-mail: stephan.baumann@tu-ilmenau.de

P. Boncz
CWI
E-mail: p.boncz@cwi.nl

K.-U. Sattler
Technische Universität Ilmenau
E-mail: kus@tu-ilmenau.de

1 Introduction

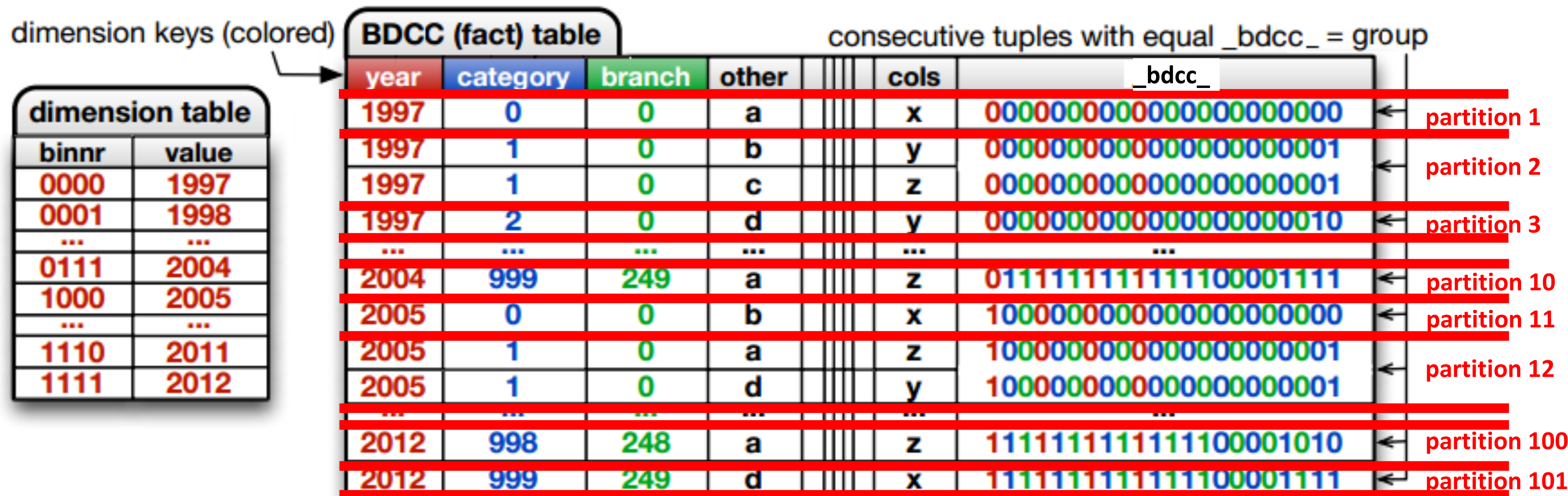
Data warehouses keep on growing, pushing the limits of machines and database technology, while analysts rely more on interactive systems. This requires robust query performance in terms of interactivity and quick response time for a broad set of queries but also in the need for shorter update cycles of the database. Also, analytical databases often go beyond the form of star and snow flake schemas and contain multiple large (fact) tables that are joined during the analysis. For example, the TPC-DS benchmark models 7 fact tables connected only through dimension tables, and a common use-case in warehousing is to analyze multiple snapshots of the same schema, joining fact tables with different versions of itself, in order to identify trends. This results in large joins dominating query execution and complicates meeting the above requirements.

In the area of physical data organization, data warehousing technology has come up with many approaches. Most important are indexing, clustering, partitioning and materialization. While all these techniques have their advantages, the also come with drawbacks: table partitioning works best only for rather coarse-grained schemes, materialization/replication requires additional storage overhead and increases update costs, and clustering typically accelerates only scans and selections.

In this work, we present a novel storage and processing framework that avoids these drawbacks. The basic idea of our Bitwise Dimensional Co-Clustering (short BDCC) approach is to *cluster* each table on *multiple* dimensions which are derived from foreign key relationships. In this way, we create foreign key connected tables (partially) sharing clustering while allowing *fine-grained granularities* of up to millions of groups. This gives us the opportunity to optimize query execution

TECHNISCHE UNIVERSITÄT ILMENAU

BDCC: how tables are stored



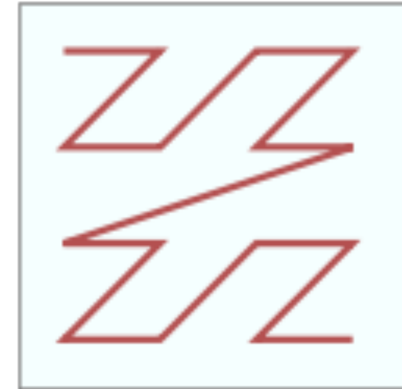
bdcc column ordering ➔ works in column stores

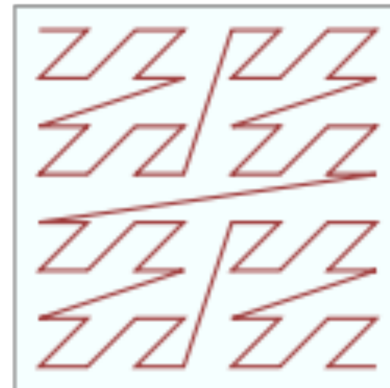
Bitwise Interleaving = Z-Ordering

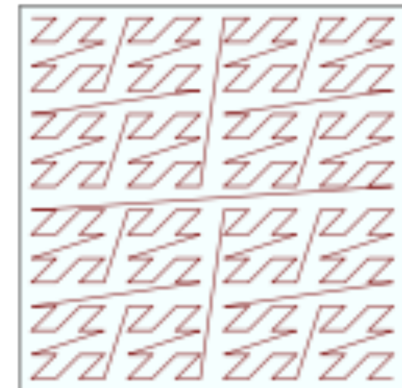
space filling curve

Computationally cheaper than eg Hilbert Curve

Almost as nice properties







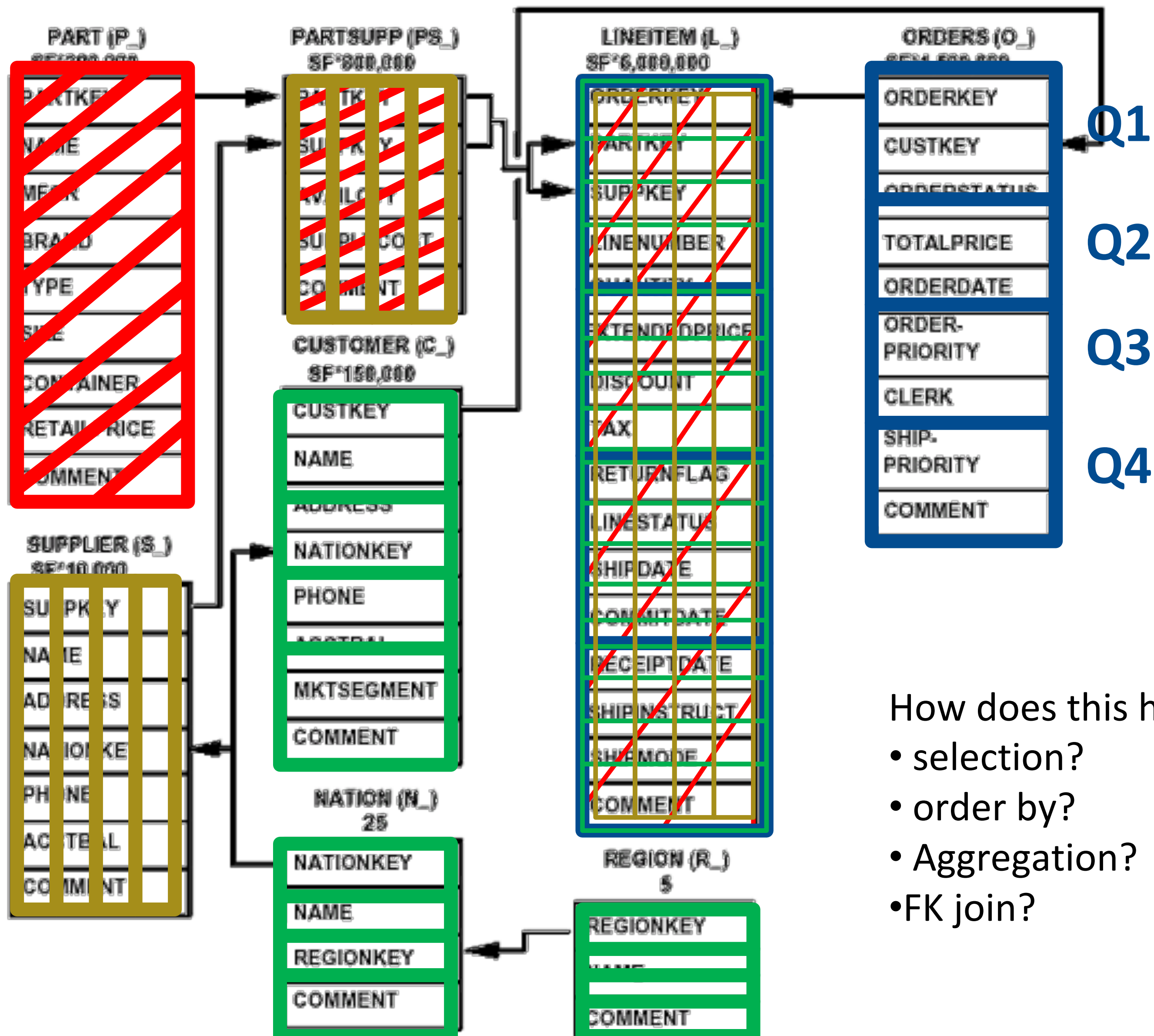
BDCC - Data Order

- **any bit interleaving** of dimensions possible
  - round-robin = Z-order
  - major-minor = classical MD index (eg DB2)
  - any bitmix in between
- our **automatic** algorithms use
  - **round robin bit interleaving**
  - clustering depth based on column densities, typically 32KB (SSD) and 512KB (HDD) blocks

BDCC - What is it?

- **Multi-dimensional** indexing
  - *table* indexing: not multi-media (audio,image) indexing here ☺
  - limited amount of dimensions (up to 5..7)
- **Multi-table** clustering (co-clustering)
  - indexing on dimensions from ***other*** tables..
  - ..reachable over foreign-key relationships
  - and exploiting common indexing dimensions among tables in operators
- Grouping into **MILLIONS** of very small groups
  - **scattered** access patterns ➔ **Flash** IO friendly!
  - clustering: because millions not possible with partitioning
- **Column-store** optimized

BDCC - The Idea



How does this help:

- selection?
- order by?
- Aggregation?
- FK join?

What BDCC gives you

Accelerates

- Most **Selections** -> selection push-down, correlations
- Most **Groupings**
- **All Foreign Key Joins** (no matter if dimensions are involved)
  - even removes joins, turning them into selections
- Certain **Order-by**

Mostly through **strong reduction of memory usage** while

- **No storage overhead**: every tuple stored once
- Bulk **update-friendly**
- Easy to **parallelize** query processing

Two Stages of the Project

- Bitwise Dimensional Co-Clustering (BDCC)
  - I/O level clustering and indexing
  - Query processing via PartitionSplit, PartitionRestart

  published in VLDBJ 2016

- Deep Dimensional Co-Clustering(DDC)
  - additional I/O block clustering
  - Query processing via DDC-Recluster()

  unpublished yet.. WIP

BDCC Structures

- BDCC **dimension**
  - mapping to consecutive integers
  - balancing through histograms and Hu-Tucker
- BDCC **table**
  - re-ordered primary copy
  - additional **_bdcc_** order attribute
- BDCC **count table**
  - summary table (**_bdcc_**, **_count_**)
  - cluster access index

BDCC Structures

LINEITEM

datekey	custkey	partkey		cols
D0	C2	P2		y
D2	C1	P1		z
D1	C0	P0		y
...	...	...		...
D3	C3	P2		z
D1	C3	P3		x
D3	C1	P3		z
D2	C0	P1		v
...	...	...		...
D0	C0	P0		z

D: date — $U_1 = U_d$

bin	datekey	year
0	D0	1997
1	D1	1998
2	D2	1999
3	D3	2000

M: 1 1 0 0 0 0 — P: FK_L_D

D: customer — $U_2 = U_c$

bin	custkey	continent
0	C0	Africa
1	C1	America
2	C2	Asia
3	C3	Europe

M: 0 0 1 1 0 0 — P: FK_L_C

D: part — $U_3 = U_p$

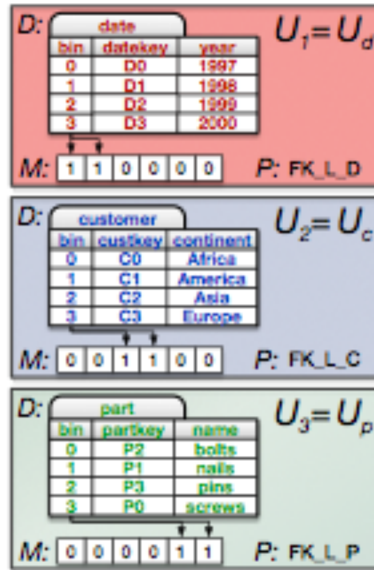
bin	partkey	name
0	P2	bolts
1	P1	nails
2	P3	pins
3	P0	screws

M: 0 0 0 0 1 1 — P: FK_L_P

BDCC Structures

LINEITEM

datekey	custkey	partkey		cols
D0	C2	P2		y
D2	C1	P1		z
D1	C0	P0		y
...	...	...		...
D3	C3	P2		z
D1	C3	P3		x
D3	C1	P3		z
D2	C0	P1		y
...	...	...		...
D0	C0	P0		z


D: date — $U_1 = U_d$

bin	datekey	year
0	D0	1997
1	D1	1998
2	D2	1999
3	D3	2000

M: 1 1 0 0 0 0 P: FK_L_D

D: customer — $U_2 = U_c$

bin	custkey	continent
0	C0	Africa
1	C1	America
2	C2	Asia
3	C3	Europe

M: 0 0 1 1 0 0 P: FK_L_C

D: part — $U_3 = U_p$

bin	partkey	name
0	P2	bolts
1	P1	nails
2	P3	pins
3	P0	screws

M: 0 0 0 0 1 1 P: FK_L_P

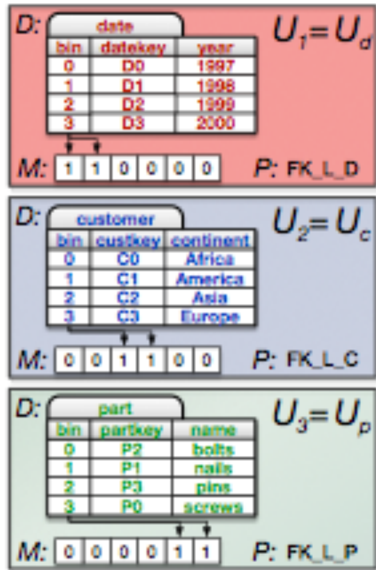
intermediate LINEITEM

year	continent	name	datekey	custkey	partkey		cols
1997	Asia	bolts	D0	C2	P2		y
1999	America	nails	D2	C1	P1		z
1998	Africa	screws	D1	C0	P0		y
...	...	...	...	...	...		...
2000	Europe	bolts	D3	C3	P2		z
1998	Europe	pins	D1	C3	P3		x
2000	America	pins	D3	C1	P3		z
1999	Africa	nails	D2	C0	P1		y
...	...	...	...	...	...		...
1997	Africa	screws	D0	C0	P0		z

BDCC Structures

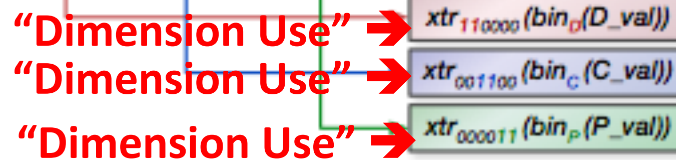
LINEITEM

datekey	custkey	partkey	cols
D0	C2	P2	y
D2	C1	P1	z
D1	C0	P0	y
...	...	...	...
D3	C3	P2	z
D1	C3	P3	x
D3	C1	P3	z
D2	C0	P1	y
...	...	...	...
D0	C0	P0	z



intermediate LINEITEM

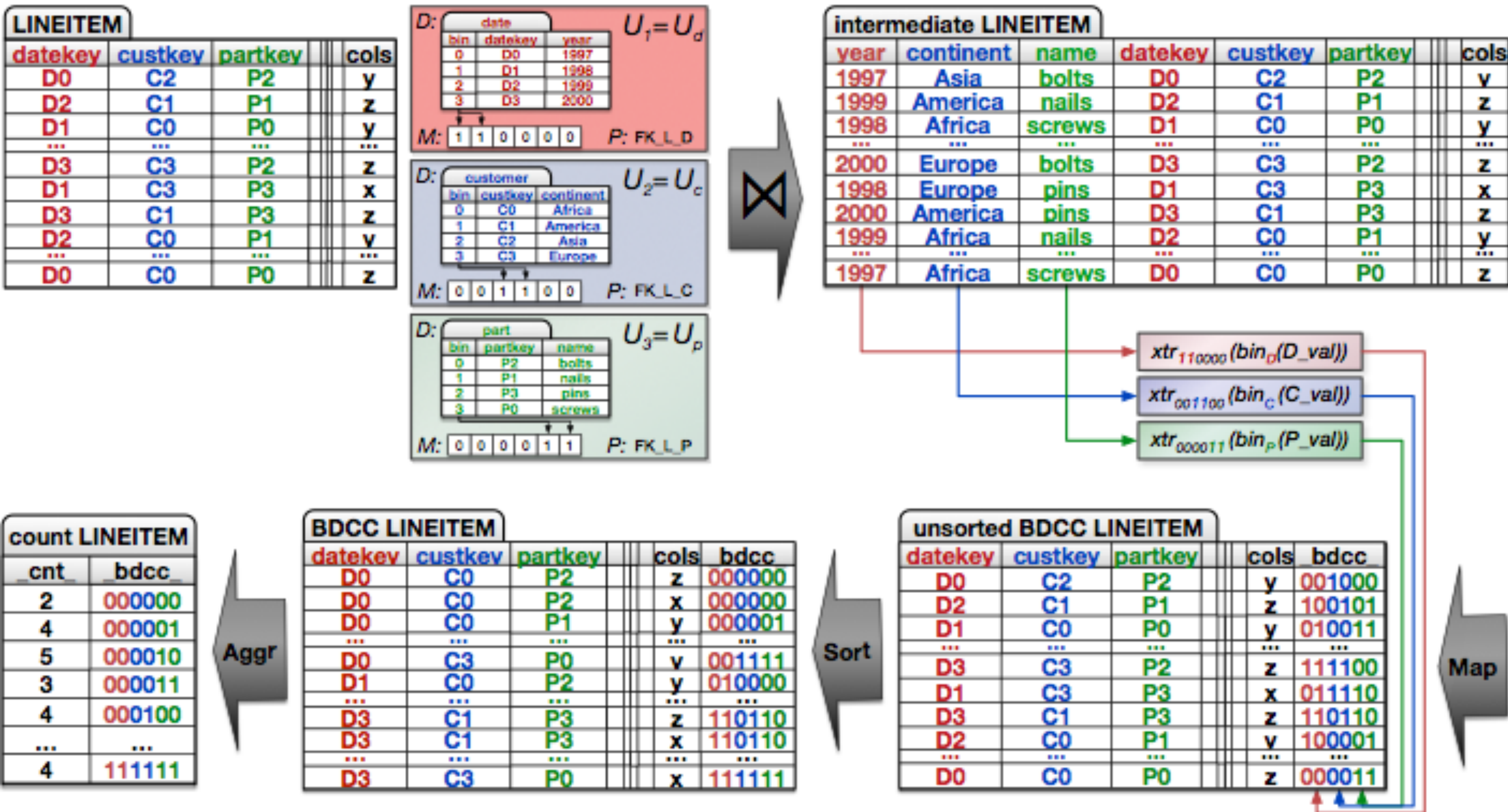
year	continent	name	datekey	custkey	partkey	cols
1997	Asia	bolts	D0	C2	P2	y
1999	America	nails	D2	C1	P1	z
1998	Africa	screws	D1	C0	P0	y
...	...	...	...	...	...	...
2000	Europe	bolts	D3	C3	P2	z
1998	Europe	pins	D1	C3	P3	x
2000	America	pins	D3	C1	P3	z
1999	Africa	nails	D2	C0	P1	y
...	...	...	...	...	...	...
1997	Africa	screws	D0	C0	P0	z



unsorted BDCC LINEITEM

datekey	custkey	partkey	cols	bdcc
D0	C2	P2	y	001000
D2	C1	P1	z	100101
D1	C0	P0	y	010011
...	...	...	...	...
D3	C3	P2	z	111100
D1	C3	P3	x	011110
D3	C1	P3	z	110110
D2	C0	P1	y	100001
...	...	...	...	...
D0	C0	P0	z	000011

BDCC Structures



CWI Centrum Wiskunde & Informatica

BDCC Structures

count LINEITEM

cnt	_bdcc_
2	000000
4	000001
5	000010
3	000011
4	000100
...	...
4	111111

BDCC LINEITEM

datekey	custkey	partkey		cols	bdcc
D0	C0	P2		z	000000
D0	C0	P2		x	000000
D0	C0	P1		y	000001
...	...	...		...	...
D0	C3	P0		y	001111
D1	C0	P2		y	010000
...	...	...		...	...
D3	C1	P3		z	110110
D3	C1	P3		x	110110
...	...	...		...	...
D3	C3	P0		x	111111

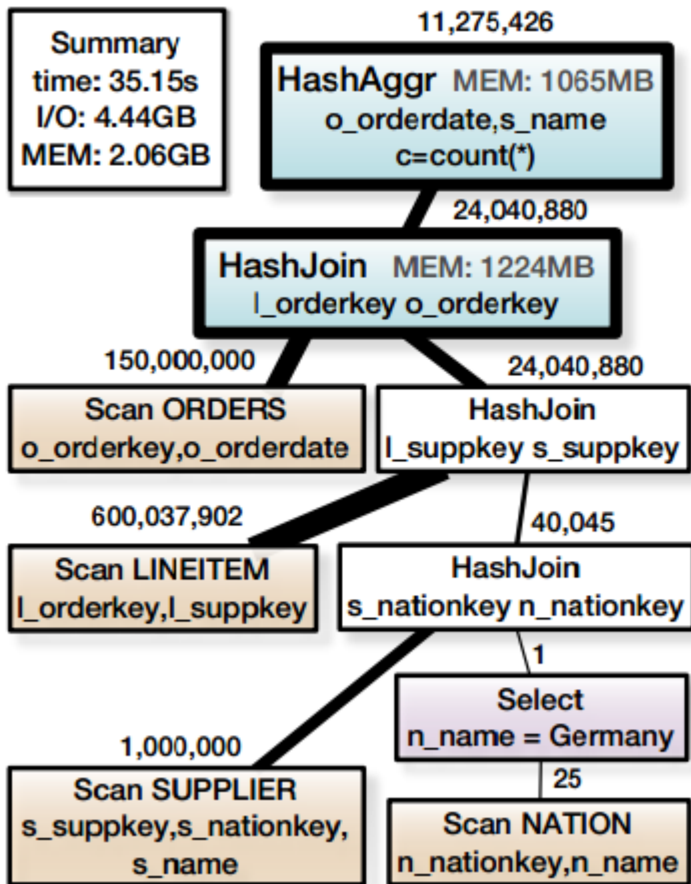
Centrum Wiskunde & Informatica

Example

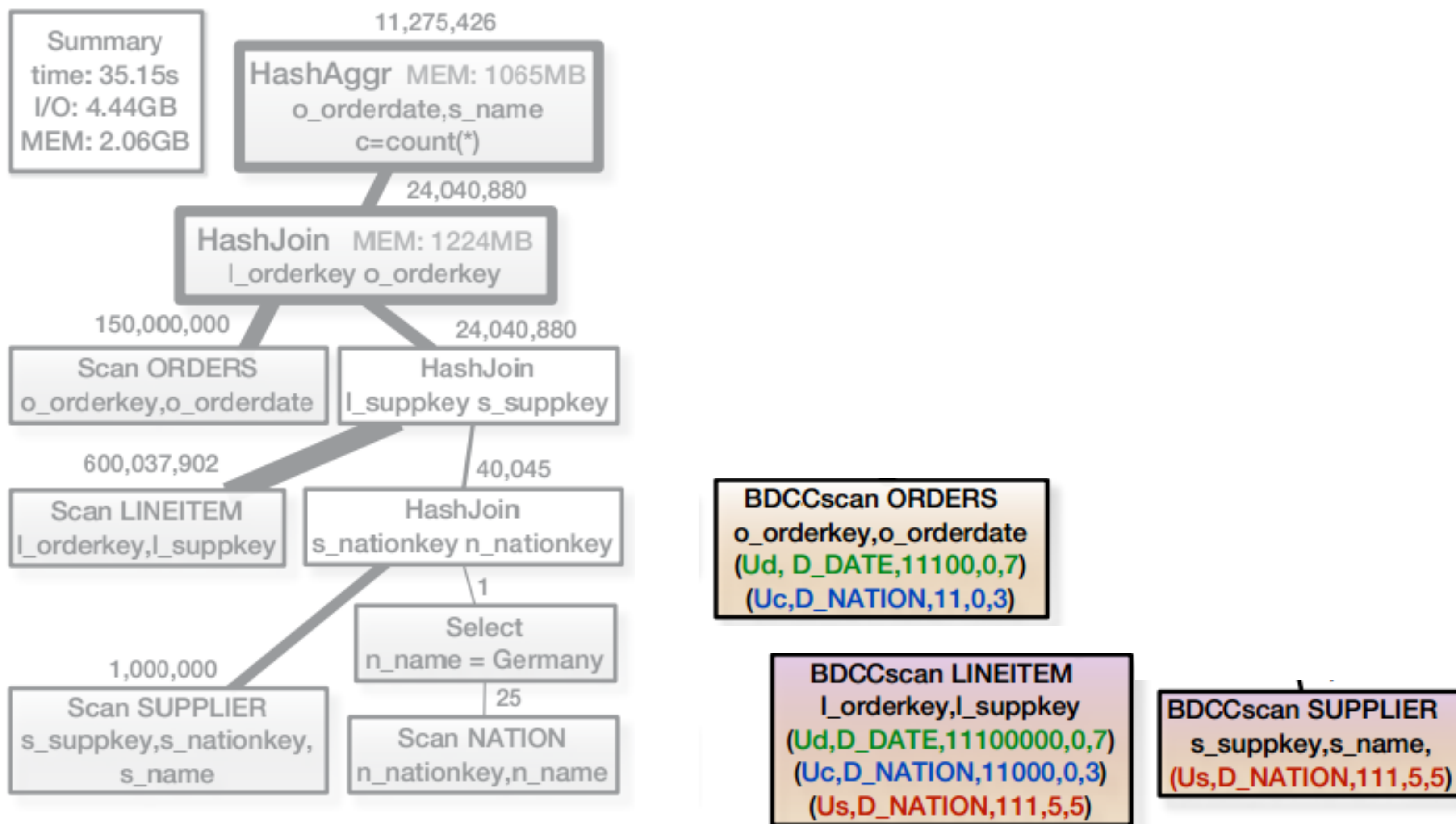
"count total ordered items from Germany per day and supplier"

```
SELECT o_orderdate, s_name, count(*)
FROM   NATION, SUPPLIER, ORDERS, LINEITEM
WHERE  n_nationkey=s_nationkey
  AND  s_suppkey=l_suppkey
  AND  l_orderkey=o_orderkey
  AND  n_name='Germany'
GROUP BY o_orderdate, s_name
```

Relational Algebra Plan



Relational Algebra Plan



BDCC-scan

Scans a BDCC table

BDCCscan LINEITEM
l_orderkey,l_suppkey
(Ud,D_DATE,11100000,0,7)
(Uc,D_NATION,11000,0,3)
(Us,D_NATION,111,5,5)

In any desired dimension order
Here:
1: orderdate
2: customer nation
3:supplier nation

Pushes down selections:
[0,7]= all
[0,3] = all
[5,5] = germany

At a desired granularity using bitmasks

3+2+3 bits set ➔ use 8 bits (256 groups)

BDCC-scan

BDCCscan LINEITEM
l_orderkey,l_suppkey
(Ud,D_DATE,11100000,0,7)
(Uc,D_NATION,11000,0,3)
(Us,D_NATION,111,5,5)

- extracts _bdcc_ bits → _gid_ column

$d_3 s_3 c_3 d_2 s_2 c_2 d_1 s_1 c_1 \rightarrow d_3 d_2 d_1 c_3 c_2 s_3 s_2 s_1$

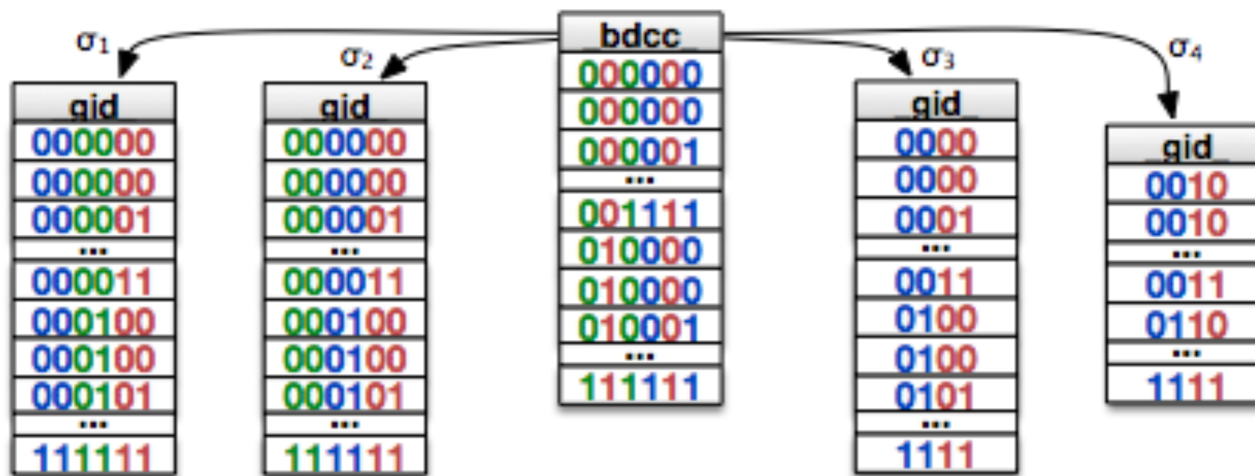
- delivers tuples **ordered** on _gid_
- performs **selection pushdown** ([lo-hi])

Basic Idea:

- BDCC-scan delivers sorted stream
  but sorting is free! As fast as a normal scan
- carefully **controlled** scatter access pattern
  we clustered on |_bdcc_| bits, but can BDCC-scan with less

BDCC FetchScan

- uses **count-table** to find the needed _bdcc_ ranges
- fetches tuple ranges in a particular order
- returns an ascending _gid_ column in the tuples



"order by cust, prod, date" ➡ $\sigma_1 = \langle\langle U_c, 110000, 0, 3\rangle, \langle U_p, 001100, 0, 3\rangle, \langle U_d, 000011, 0, 3\rangle\rangle$

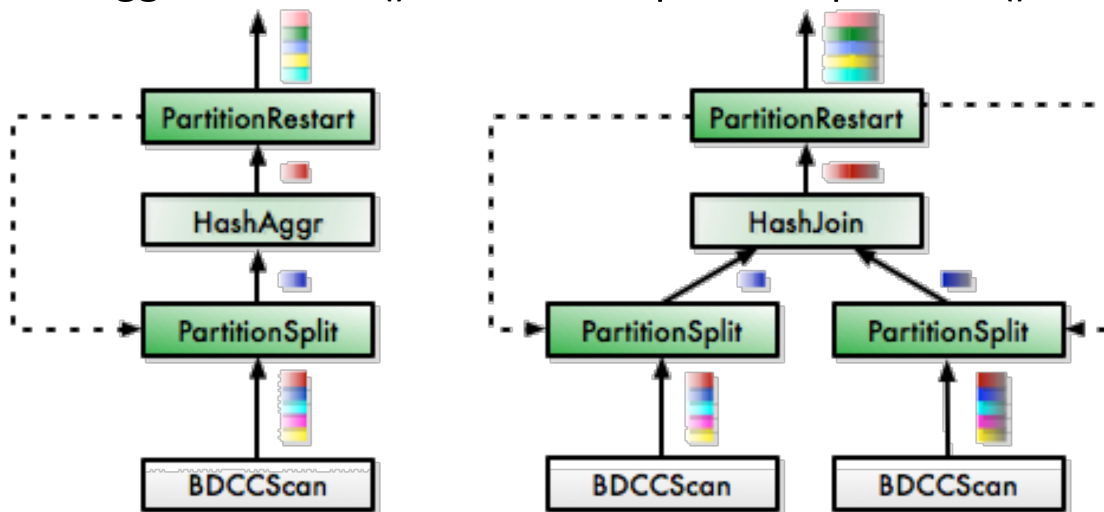
"order by prod, cust, date" ➡ $\sigma_2 = \langle\langle U_p, 110000, 0, 3\rangle, \langle U_c, 001100, 0, 3\rangle, \langle U_d, 000011, 0, 3\rangle\rangle$

"order by cust, date" ➡ $\sigma_3 = \langle\langle U_c, 1100, 0, 3\rangle, \langle U_d, 0011, 0, 3\rangle\rangle$

"order by cust, date where date >= 1999" ➡ $\sigma_4 = \langle\langle U_c, 1100, 0, 3\rangle, \langle U_d, 0011, 2, 3\rangle\rangle$

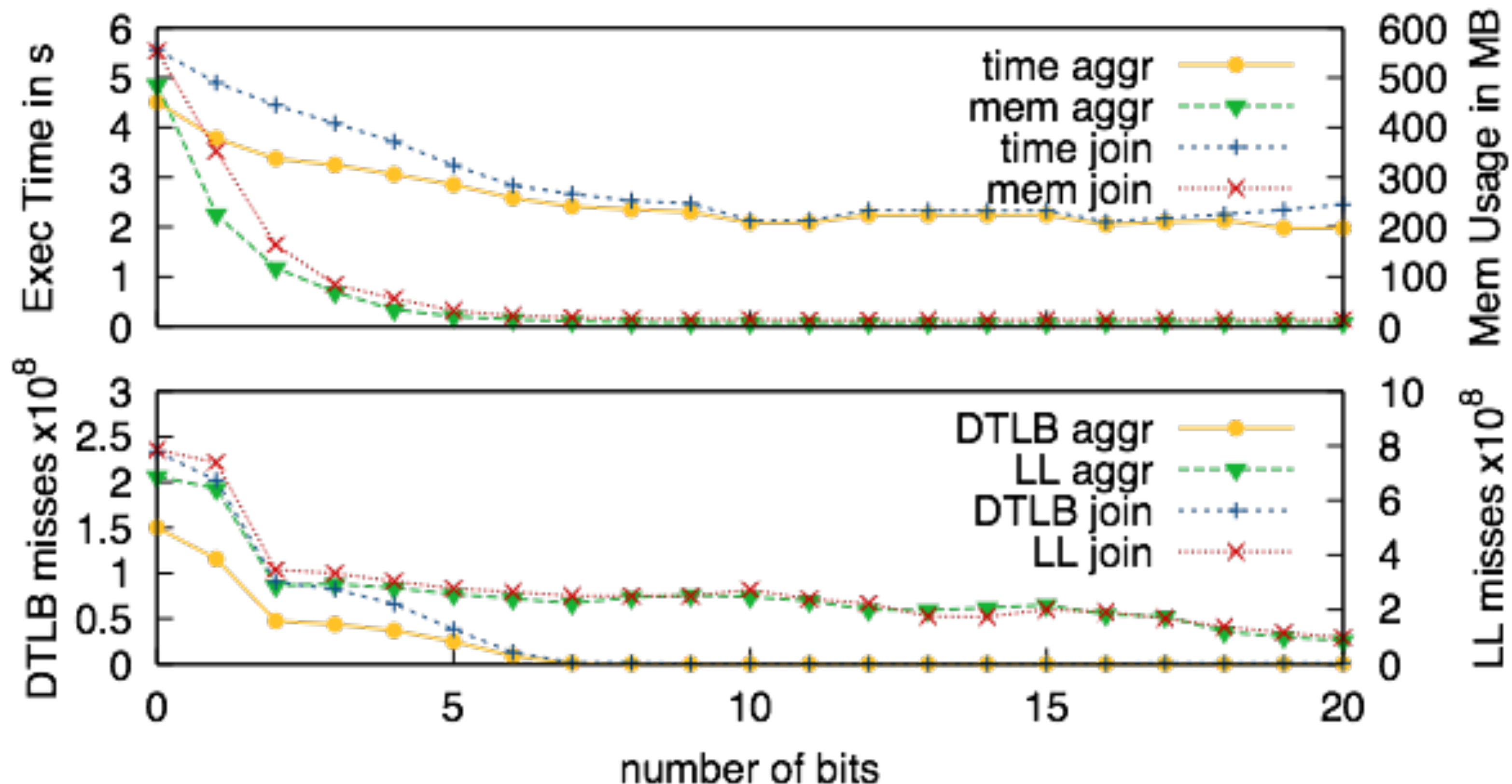
BDCC - Query Processing

- **Partition-wise** operator execution
  - hash based join, grouping/aggregation
  - better cache utilization
- **Sandwich Operators** ➔ PartitionSplit & PartitionRestart
  - sideways information passing: PartitionRestart.cross-partition? (_gid_ change)
    ➔ HashAggr/Join.flush() & PartitionSplit.next-partition()

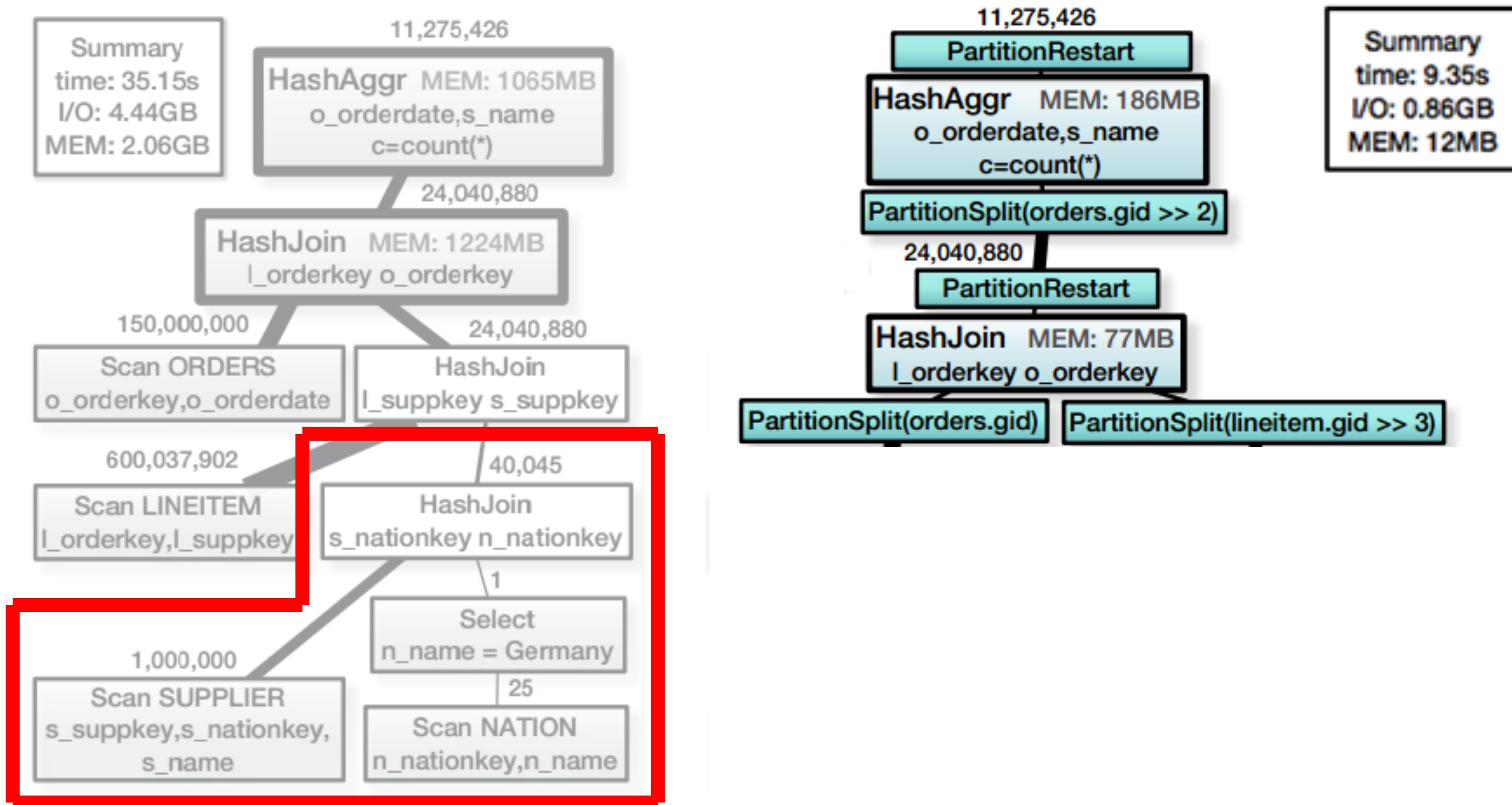


BDCC - Performance Sandwich Operators

- Micro-Benchmarks (TPC-H SF10, LINEITEM-ORDERS)

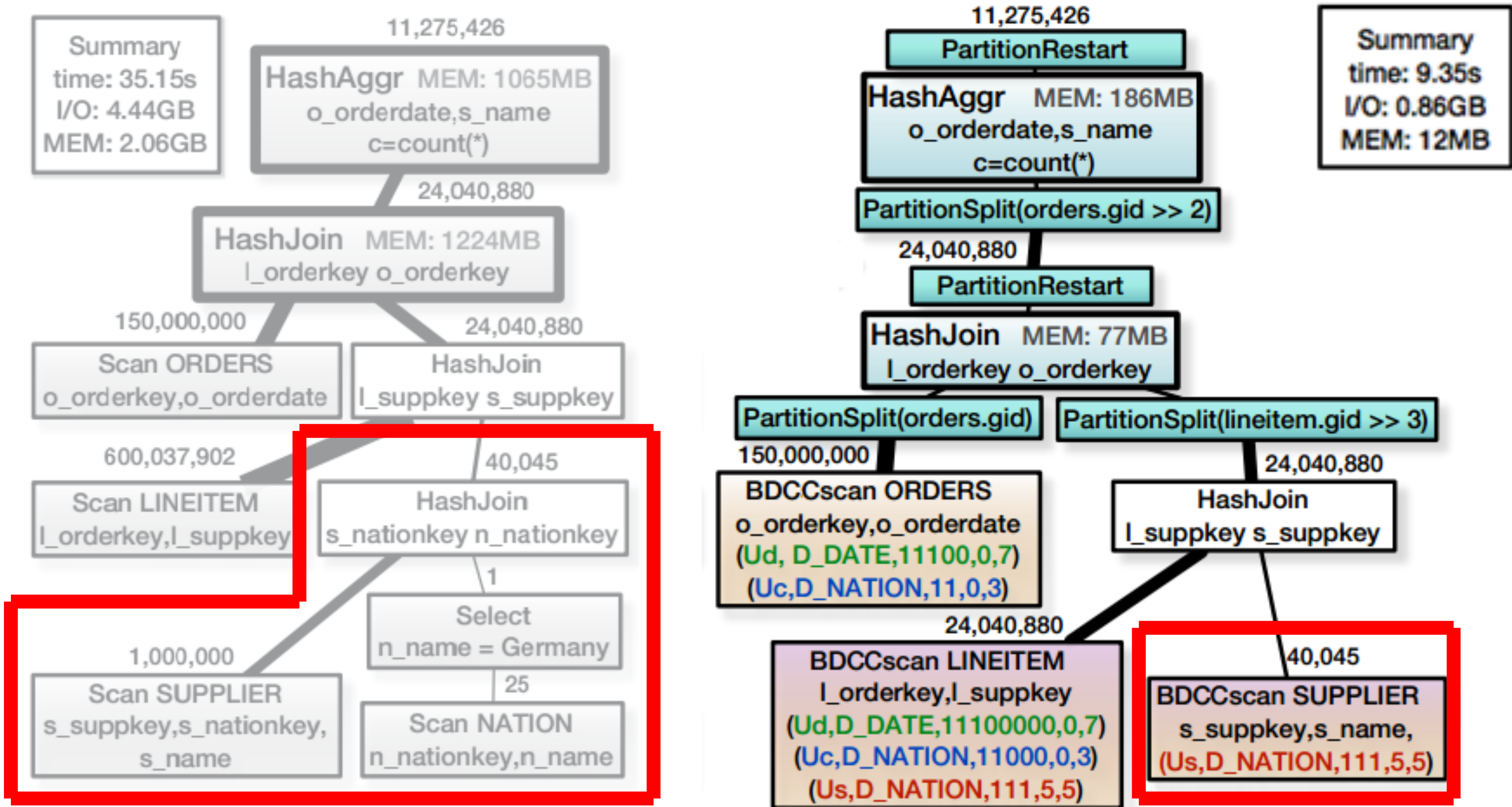


Relational Algebra Plan



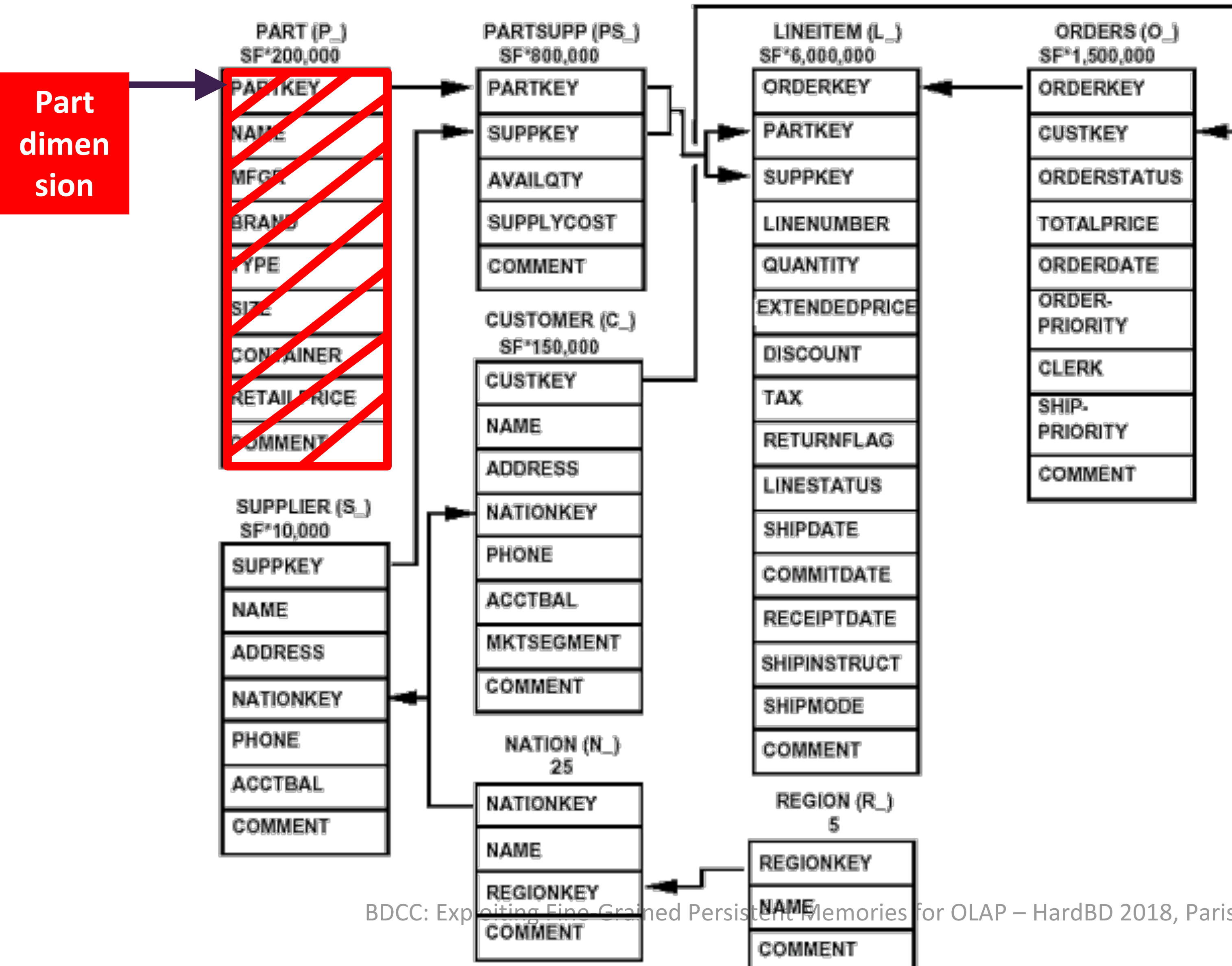
Selection Pushdown +
Dimension Join Elimination

Relational Algebra Plan

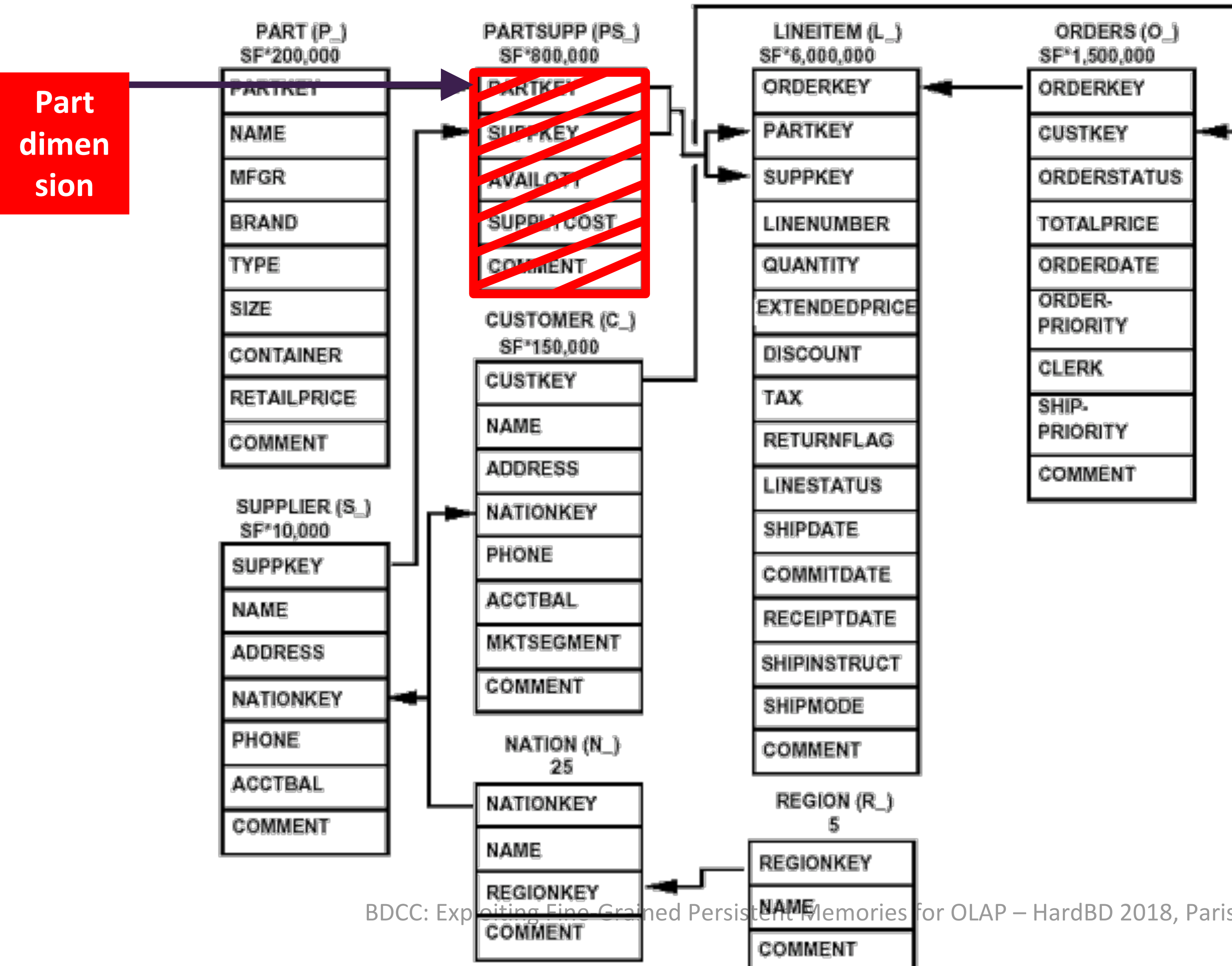


Selection Pushdown +
Dimension Join Elimination

Co-Clustering Close-up

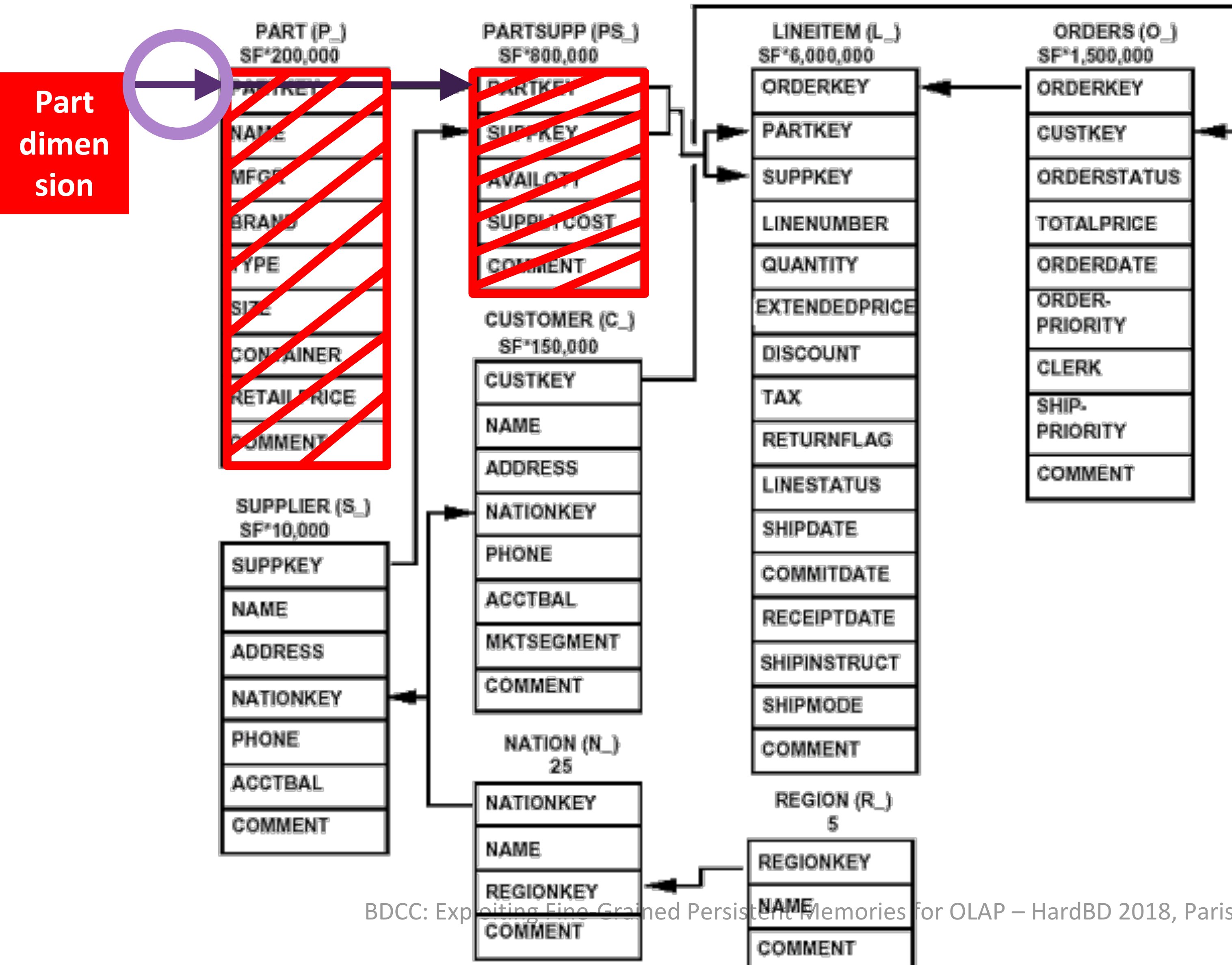


Co-Clustering Close-up

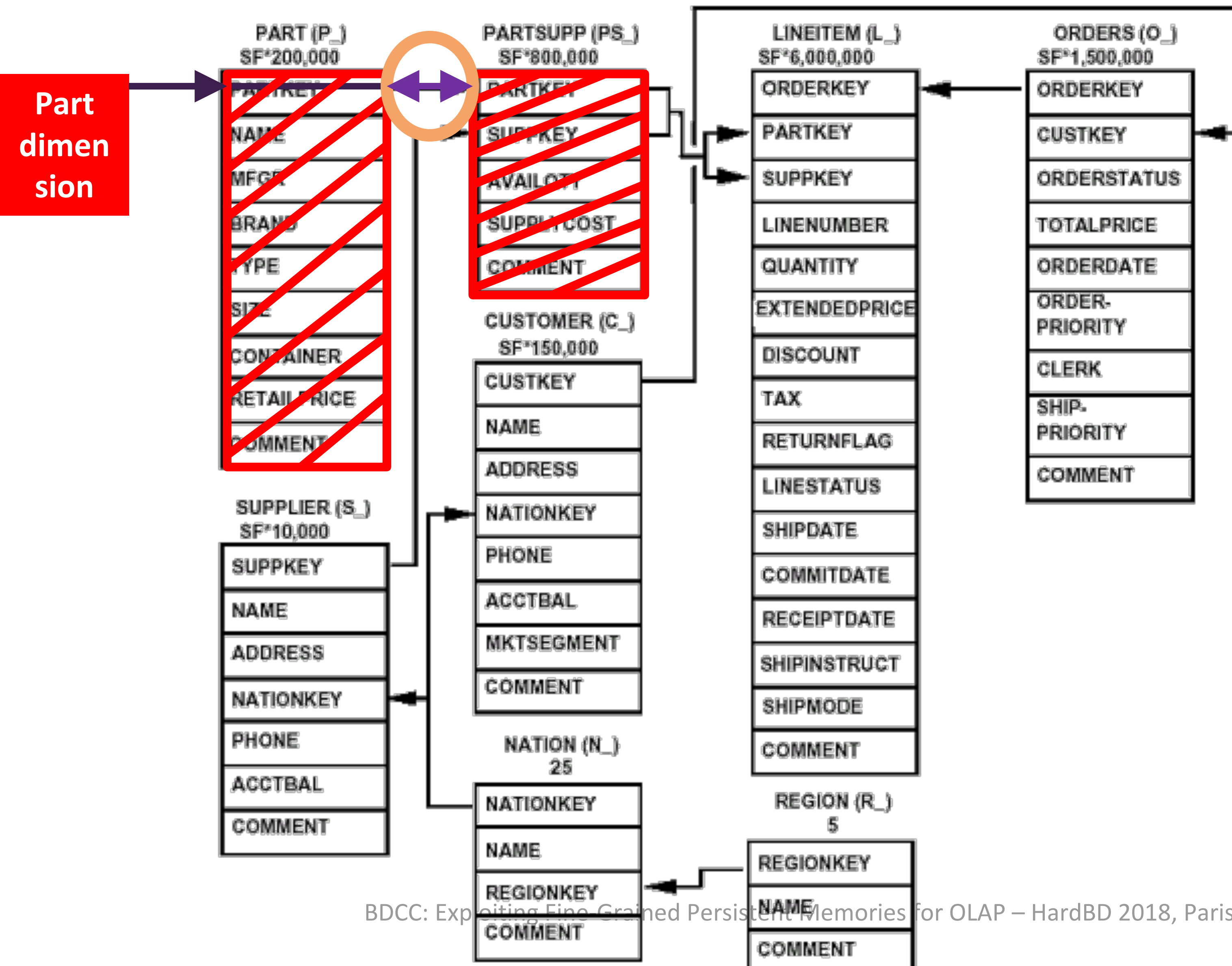


Common Path = Co-Clustering

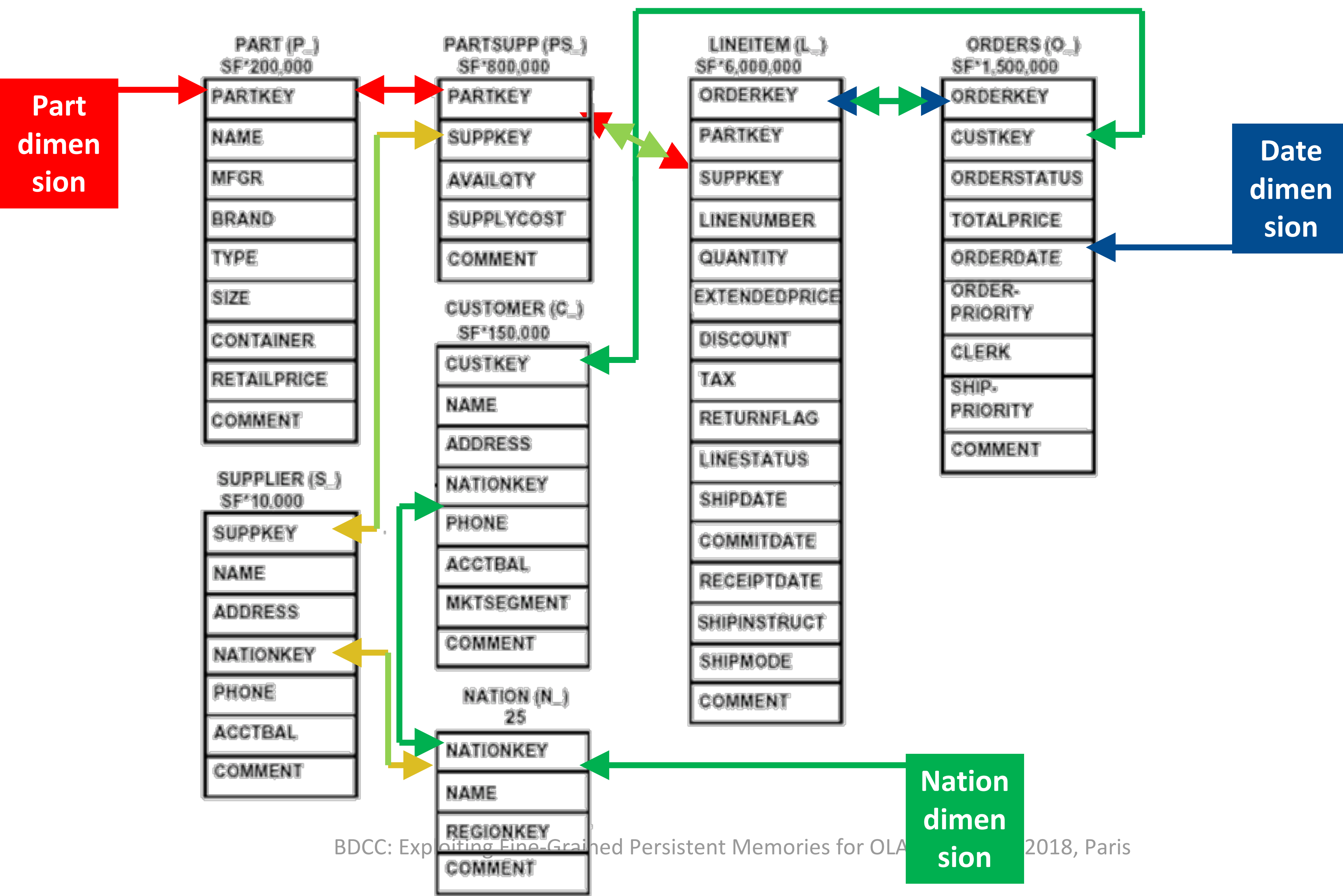
Part dimension



Common Dimension = Accelerated Join



BDCC: All FK Joins Accelerated!



BDCC - Schema Design

- **Semi-automatic**
  - Input: CREATE INDEX() and FOREIGN KEY()
- Schema **traversal along foreign key paths**
  - propagation of „Index“ dimensions
  - weighted according to FK paths
- **automatic** creation of dimensions and tables
  - **round robin bit interleaving**
  - clustering depth based on column densities, typically 32KB (SSD) and 512KB (HDD) blocks

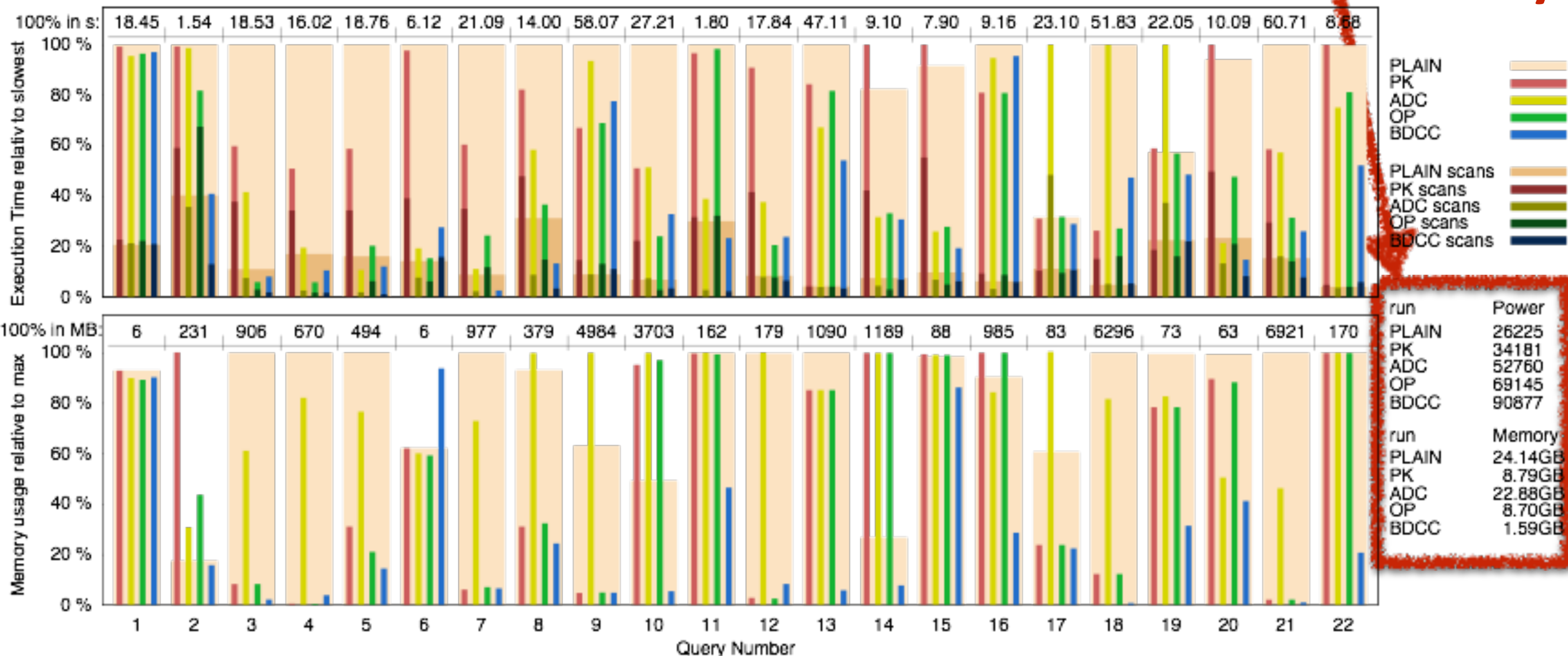
BDCC - Optimizer

- IDU: Interesting Dimension Uses
  - all dimensions determined by join, sort or aggregation attribute
- IDO: Interesting Dimension Orders
  - all dimension order permutations of each IDU
- MDO: Maximal Dimension Orders
  - Pruning of dominated sort orders of IDOs
- MDOs represent „interesting orders“ for enumeration

BDCC Performance

- TPC-H SF100 execution time for BDCC, cold buffer pool

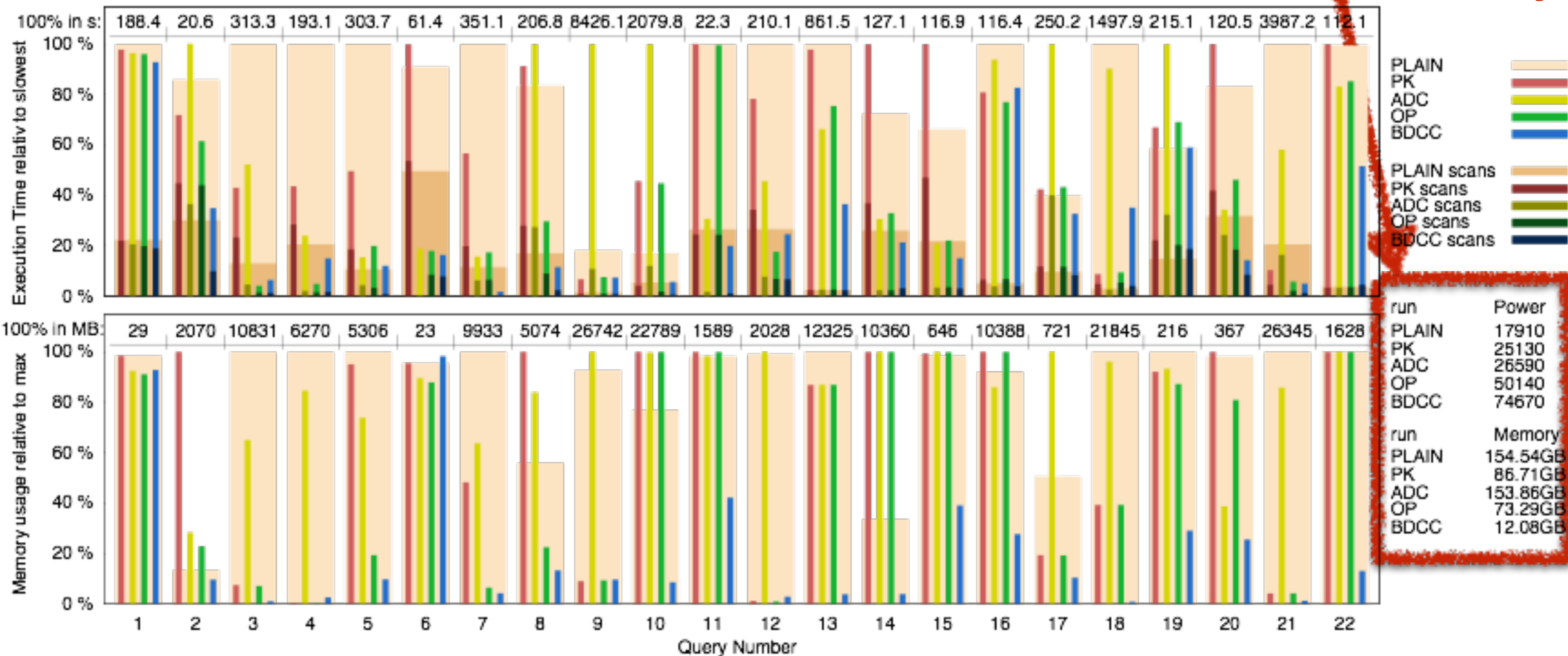
much better power scores with much less memory



BDCC Performance

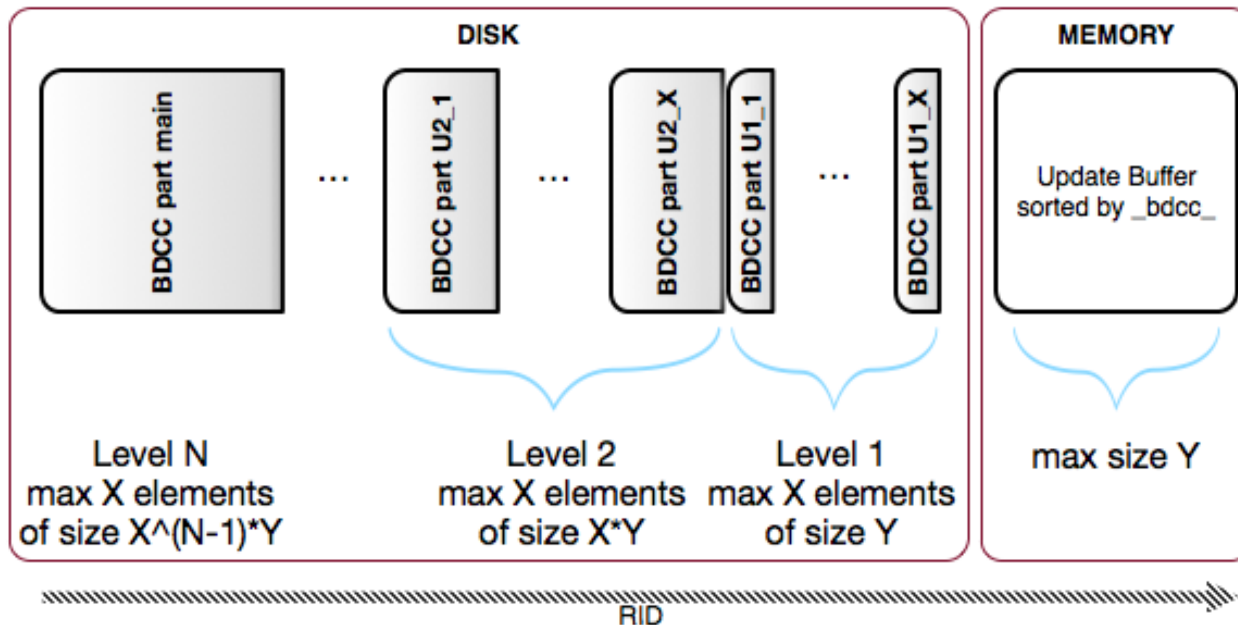
- TPC-H SF1000 execution time for BDCC, cold buffer pool

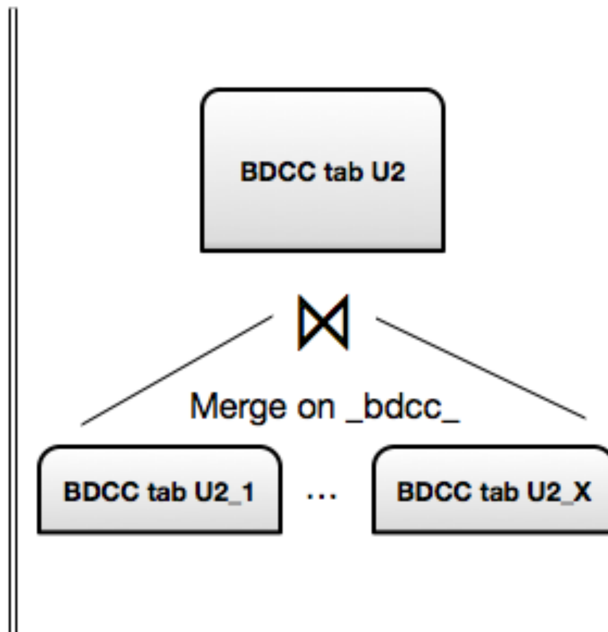
much better power scores with much less memory



BDCC - Updates

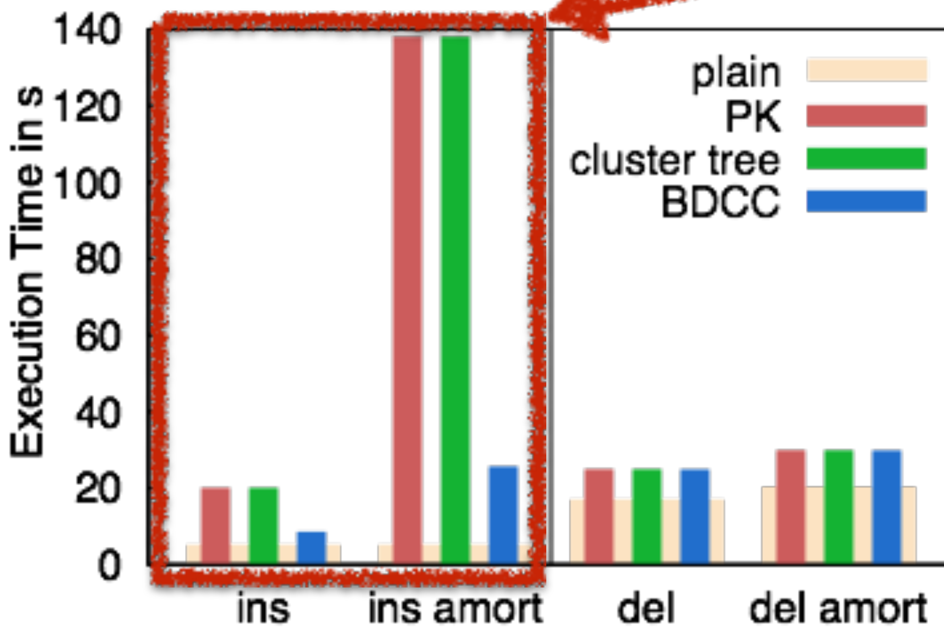
- Batch Update Support
  - in-memory buffer
  - „log-structured merge“





BDCC Updates

- TPC-H SF100 update set

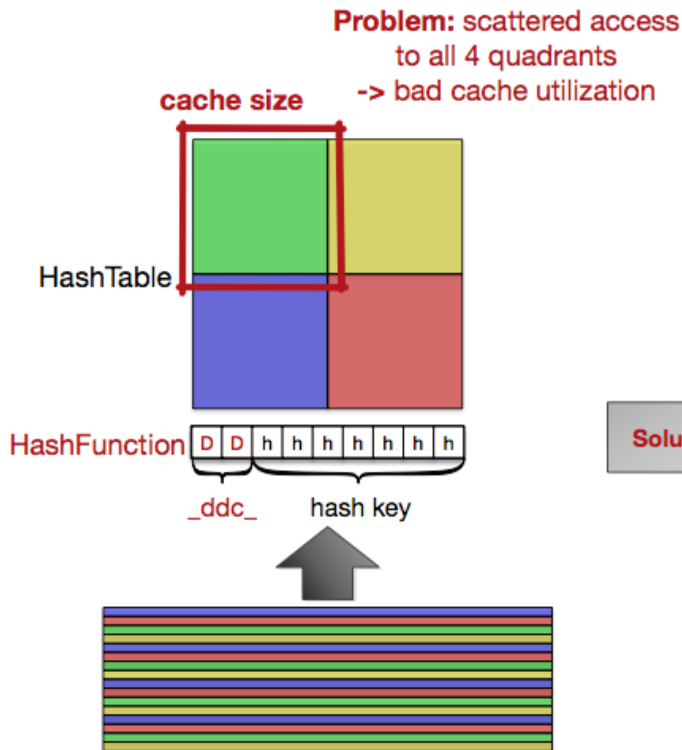


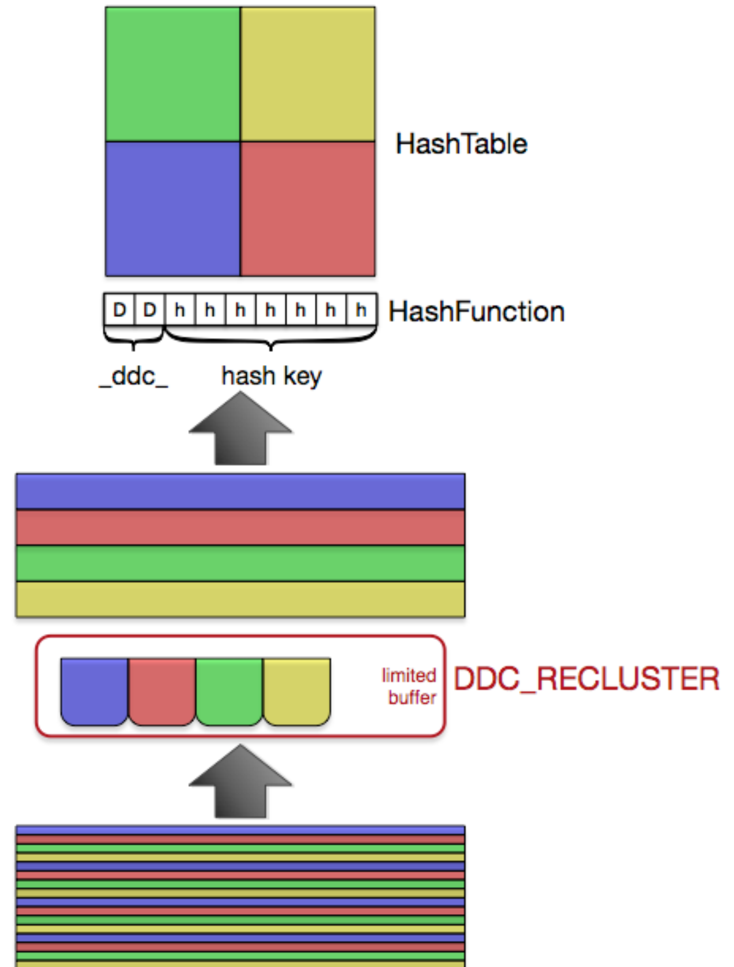
- **60% bulk append speedup comp. to cluster trees (ordered projections, using PDTs)**
- **for many update sets, BDCC only merges with previous updates instead of PDT merge with full table**

Deep Dimensional Clustering (DDC)

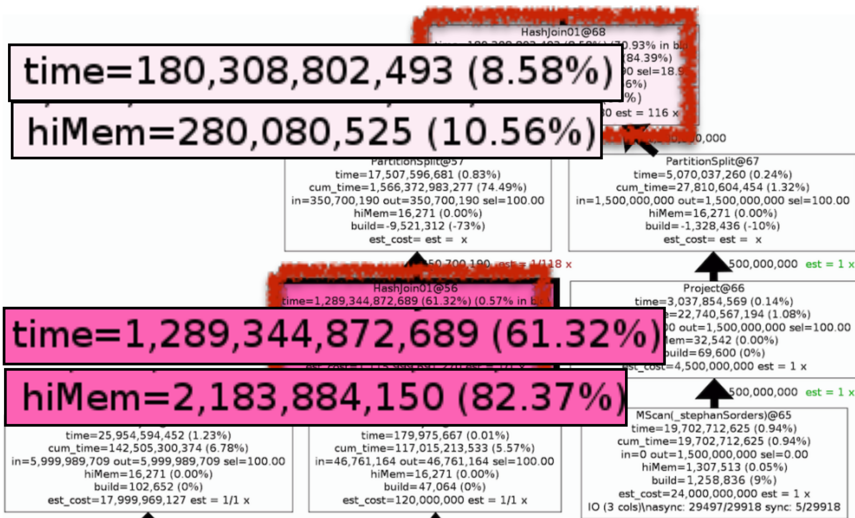
- Idea:
  - Make **_bdcc_** have as many bits as possible
  - For I/O (BDCC-scan) only use the major bits (groups of ~32KB)
  - Note, inside the 32KB tuple block, there is more clustering
    - Inside a cache line tuples tend to belong to the same group
  - Idea: exploit this locality (these deep bits) in operators
    - For really cheap **cache partitioning**
    - make joins **cache-conscious again**

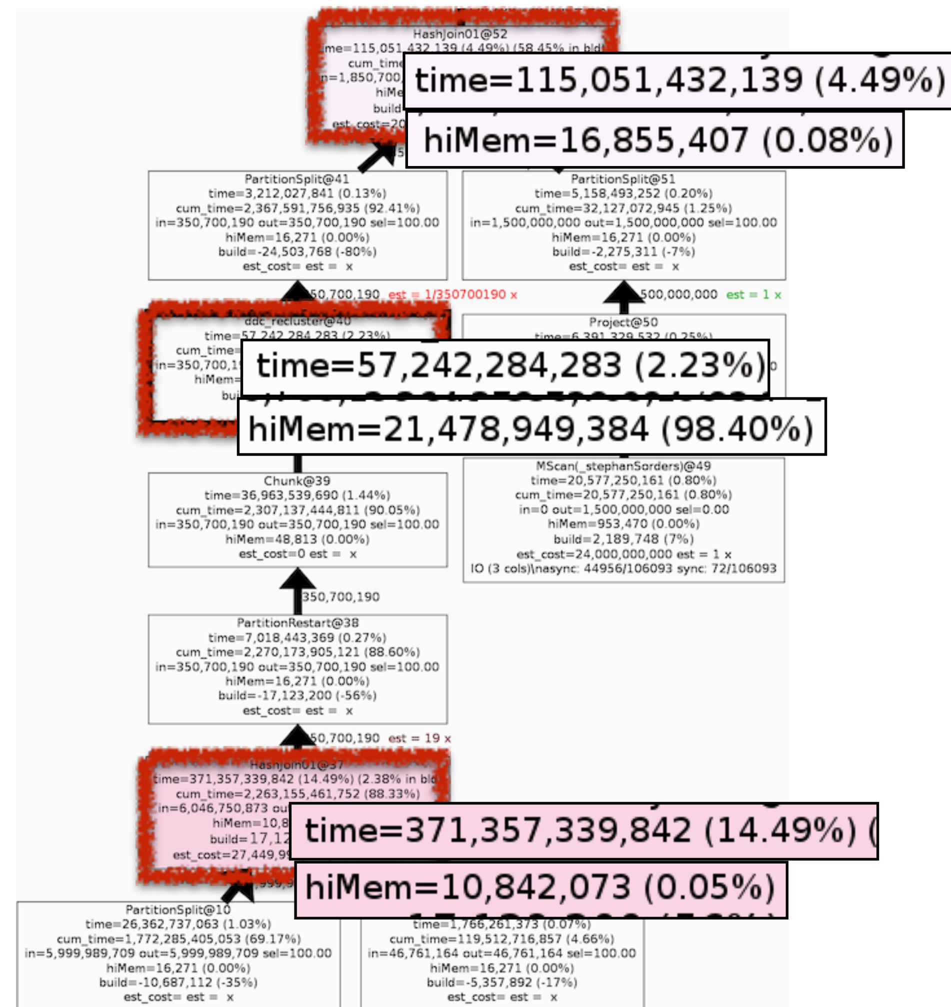
DDC Extensions





DDC Performance





BDCC

DDC

Conclusion

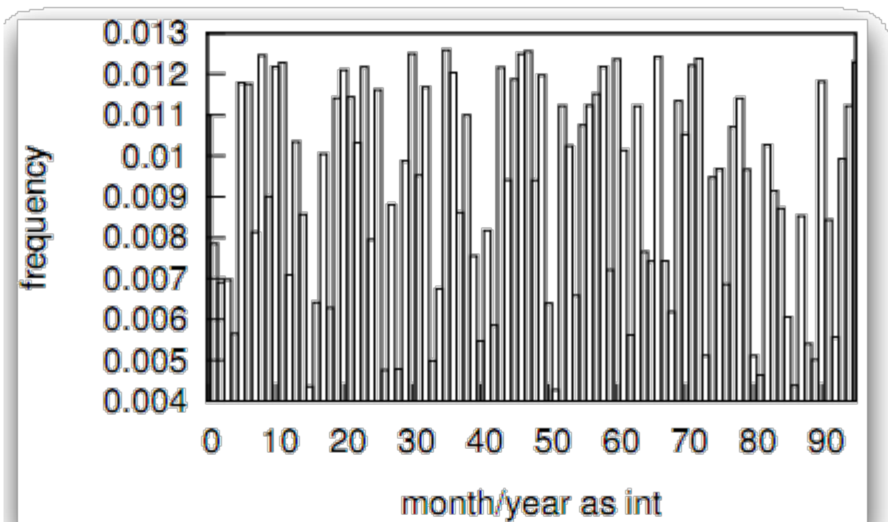
- BDCC & DDC
  - clever ordering of tables, and **co-ordering** of tables
  - **millions of tiny groups** (NVRAM friendly!)
  - All the goodies in one go:
    - fast selections (even cross-table propagation)
    - fast joins, fast groupbys, fast sorts (little RAM needed)
  - Sideways info passing **sandwich operators**
    - No need for new join/aggr operators
  - QOPT framework that extends interesting orders
  - Updatable using LSM ideas – data is stored only once

Dimension Construction

Dimension = set of bins

- Range-Binning of a domain
- Histogram-based approach
  - Needs frequency information

ORDERDATE



binnr	value	unq
0	02/03/1993	0
1	01/02/1994	0
2	23/01/1995	0
3	15/02/1996	0
4	02/04/1997	0
5	07/01/1998	0
6	21/12/1998	0
7	*no max*	0

Assigning Bin Numbers: Naïve Way

- Skew/frequent values (➔single-value bins)

value	frequency	code	c2	c1
(null)	.70	000	00	0
Polytech	.15	001	00	0
Bachelor	.08	010	01	0
Master	.06	011	01	0
PhD	.01	100	10	1

Hu-Tucker Binning

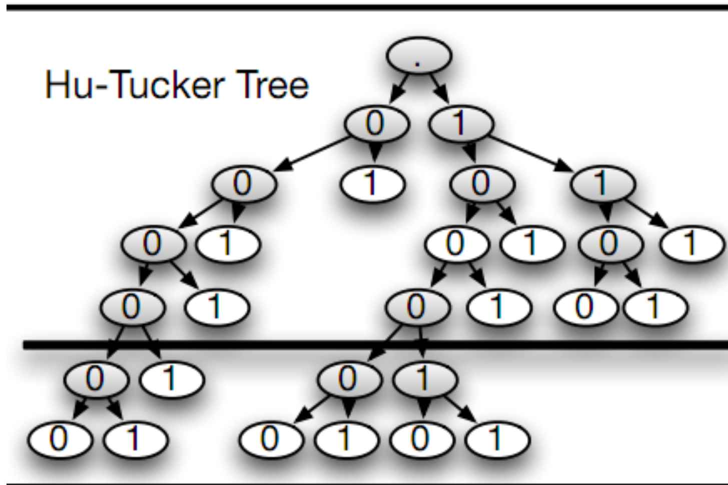
- Frequency-based Bin Number Assignment

value	frequency	code	c3	c2	c1
(null)	.70	0000	000	00	0
Polytech	.15	1000	100	10	1
Bachelor	.08	1100	110	11	1
Master	.06	1110	111	11	1
PhD	.01	1111	111	11	1

Hu-Tucker = Order Respecting **Huffman** Coding

Hu Tucker Dimension Binning

NATION



binnr	value	unq
0 {0000}	(Am,Canada)	0
1 {0001}	(Am,Peru)	1
2 {0010}	(Am,United States)	1
4 {0100}	(As,China)	1
8 {1000}	(As,Vietnam)	0
9 {1001}	(Eu,France)	1
10 {1010}	(Eu,Germany)	1
12 {1100}	(Eu,Romania)	1
13 {1101}	(Eu,Russia)	1
14 {1110}	(Eu,United Kingdom)	1

but why is this relevant?

Variety in Data **Density** of Columns

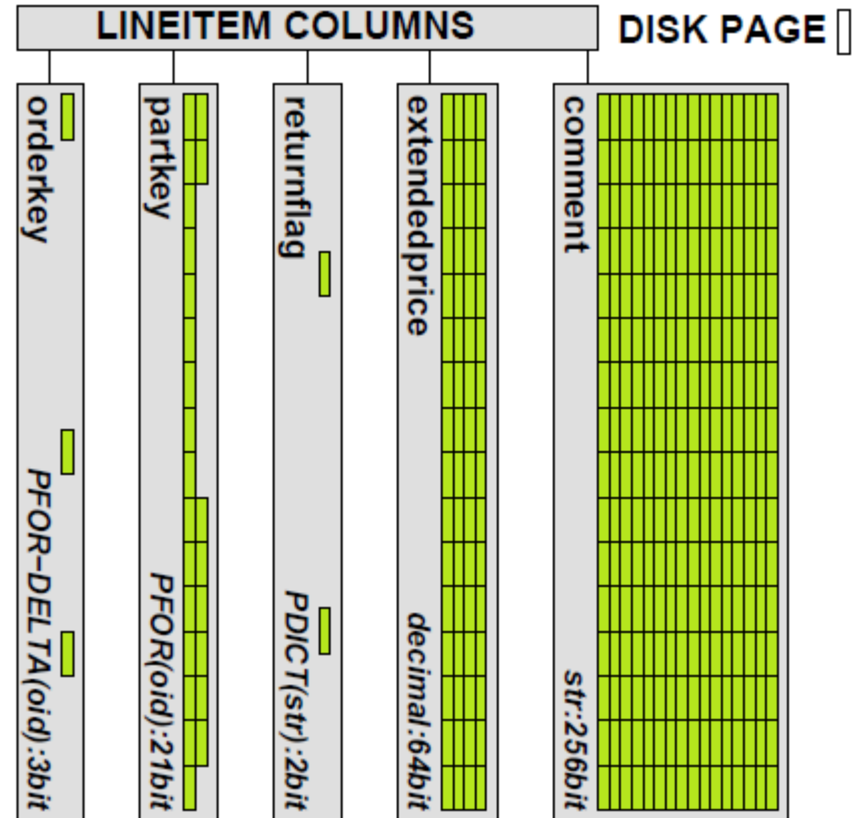
- l_linestatus **0.25** b/tuple
- l_comment **30** b/tuple

Factor 120 difference

What is the optimal BDC bin size?

- Depends on **disk block** size
- Depends on **column density**

What to do if a query accesses multiple columns of **very different** densities?



Granularity Tuning in BDCC

1. Is an issue during **table creation**
   – A dimension is used in multiple tables
   – each table needs a different granularity

2. Is an issue during **query execution**
   – Table is clustered at some granularity
   – Given a **set of columns** to scan:
     at what granularity to scan the table?

Z-Ordering for Column Stores

there is a **column-store specific** argument for bit interleaving, also:

- suppose **BDCC-scan(T,C_1)** is efficient at **8** bits, needing sorted access to supplier (**s**)
- suppose **BDCC-scan(T,C_2)** that selects other columns **C_2** that are on average much smaller than those in **C_1**, is efficient only up to **5** bits granularity

Takeaway: **column stores need a variable access granularity**

- Major-minor clustering leaves the minor dimension unusable for thin columns (**C_2**)
- Bit-interleaving (Z-ordering) allows thin column scans to profit from all dimensions