The VLDB Journal
DOI 10.1007/s00778-015-0417-y

REGULAR PAPER

Bitwise dimensional co-clustering for analytical workloads

Stephan Baumann[1] · **Peter Boncz**[2] · **Kai-Uwe Sattler**[1]

Received: 7 May 2015 / Revised: 3 October 2015 / Accepted: 27 November 2015

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.

✉ Stephan Baumann
stephan.baumann@tu-ilmenau.de

Peter Boncz
p.boncz@cwi.nl

Kai-Uwe Sattler
kus@tu-ilmenau.de

[1] Technische Universität Ilmenau, Ilmenau, Germany

[2] CWI, Amsterdam, The Netherlands

Keywords OLAP · Data warehouse · Clustering · Indexing · Storage · Database design · Query processing · Sandwich operators

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 seven 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

Published online: 12 January 2016

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 to fit modern hardware architectures, with a particular focus on the memory hierarchy. When processing joins and aggregations in many small groups it is possible to maintain hash tables in a L2 cache-friendly size, significantly accelerating these operations.

In a nutshell, BDCC provides benefits for database design, data access and query processing.

(i) Replication-free clustering for millions of groups with benefits that only multiple (up to 6) replicated, clustered indices or sorted projections could provide.
(ii) Automated workload-agnostic schema design for fast and robust query execution tailored to modern hardware architectures.
(iii) Easy to implement query processing techniques for partitioned data that significantly save memory while at the same time accelerate all relevant foreign key joins and hash aggregations without the typical plan explosion.

The important design decisions to achieve this, are:

(i) The interpretation of partitioning or clustering as a tuple ordering problem with a scan operator to retrieve different orderings.
(ii) The creation of a co-clustered or co-ordered table layout according to foreign key definitions.
(iii) The seamless integration of partitioned operator execution into the execution engine while fully re-using existing operators and avoiding any operator duplication in query execution plans.

The remainder of the paper is as follows. Section 2 introduces the concept of co-clustering. Section 3 discusses related work. Section 4 provides a formal introduction of BDCC including all relevant definitions. Section 5 explains the steps to derive a physical BDCC schema. Section 6 introduces algorithms for efficient query processing with BDCC. Section 7 handles the updatability of BDCC. Section 8 discusses the applicability to row stores and future work and Sect. 9 provides results of experiments with TPC-H and SSB.

2 Co-clustering: What and why?

Creating locality for data with common characteristics has proven to be particularly beneficial for data warehouses. Based on star- and snowflake-schemas as a basis for data modeling, various solutions such as ADC clustering [13], IBM's MDC [29] or MDAM [22] have been proposed. The common idea here is to index the fact table by multiple dimension attributes and store tuples either sorted or clustered by this index. While MDC and MDAM only cluster on dimensions inside a table, ADC adjoins dimension columns from foreign key connected dimension tables in order to cluster the table.

These approaches show limitations when more complex schemas with multiple interconnected fact tables are used. In particular when not each fact table is directly connected with all its relevant dimension tables and heavy joins between the interconnected fact tables are required. Also, fact tables may not directly be connected but may share a set of common dimensions and still be used together in the same query. Applying the above methods results in optimizing each fact table by itself, providing only the best possible access for local or directly connected dimensions.

In a scenario of multiple different fact tables with a shared (sub)set of dimensions this is not sufficient. Joins and propagation of selection predicates between fact tables are not supported. This becomes particularly evident when selection predicates restrict attributes that are correlated to one or more dimension attributes and I/O is reduced based on more sophisticated algorithms executed on metadata of the fact tables. In case of independently clustered tables, it is impossible to automatically exploit this additionally derived knowledge on another fact table. Also, standard multi-dimensional clustering methods do not ensure that fact tables with shared dimension share this circumstance in their clustering to a point that can be exploited in query execution, in particular during join processing.

Our approach of organizing tables tackles exactly these situations while also providing the features of classical multi-dimensional clustering approaches for standard star- and snowflake-schemas. Creating a clustering we follow two guidelines: First of all, a table clustering should always use local dimensions and the dimensions the table is directly foreign key connect to. This covers star- and snowflake-schemas and is typically provided by other multi-dimensional clustering approaches. In addition, however, a table clustering should also take into account dimensions from other fact tables that are reachable over foreign keys. This ensures that two tables that are foreign key connected share a part of their clustering parameters. As a consequence, during query optimization matching clusters in both tables can be detected and techniques like selection propagation or partitioned joins can be applied. Following these two guidelines throughout the whole schema leads to a table layout that we call *co-clustered*.

Figure 1 provides a simple, TPC-H-based example of two fact tables `ORDERS` and `LINEITEM` and three-dimensional tables `D_CUSTOMER`, `D_DATE` and `D_PART`. `ORDERS` is foreign key connected to `D_DATE` (`FK_O_D`) and `D_CUSTOMER` (`FK_O_C`), `LINEITEM` is foreign key connected to

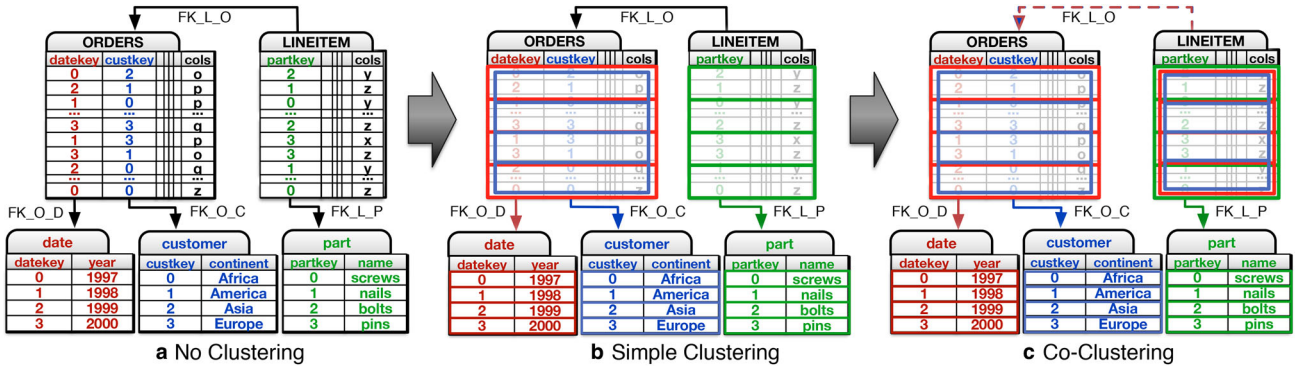

Fig. 1 Table setup with two fact tables and three dimensions. **a** Plain data without clustering, **b** simple clustering and **c** schematic visualization of a co-clustered table layout—*same color* means compatible clustering on that dimension (color figure online)

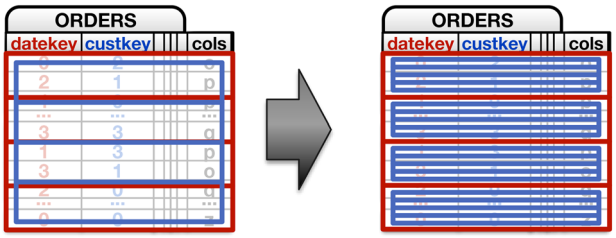

Fig. 2 Schematic illustration of one possible dimension interleaving for ORDERS

D_PART and to the other fact table ORDERS. While Fig. 1a shows the plain table layout, Fig. 1b sketches the idea of clustering each fact table without the aspect of co-clustering (i.e. ADC), and Fig. 1c schematically illustrates the clustering including the co-clustering concept. As ORDERS is foreign key connected to D_DATE with four distinct values (1997–2000), ORDERS is organized into four different groups, illustrated by the red grid. ORDERS is also foreign key connected to D_CUSTOMER and, thus, in addition grouped into four different D_CUSTOMER groups (blue grid). How these two groupings are realized at the same time is left open here and will be explained shortly, so for the illustration they are just placed on top of each other. LINEITEM is organized in a similar way using D_PART, leading to a clustering following the ADC guidelines or in case of a denormalized table layout MDC or MDADM Fig. 1b. LINEITEM, however, is not only directly foreign key connected to the D_PART dimension (FK_L_P), but also to ORDERS (FK_L_O). Following the concept of co-clustering, LINEITEM is additionally clustered according to all dimensions of ORDERS, adding D_DATE and D_CUSTOMER to its clustering. As the foreign key is only defined from LINEITEM to ORDERS, ORDERS is not clustered using D_PART.

One possible way to use dimensions in a clustering is to set one dimension as the major ordering of the table and another as the minor ordering as suggested by all approaches above. This is illustrated in Fig. 2. The red (D_DATE) dimension is used as major ordering, forming four groups in ORDERS and the blue (D_CUSTOMER) dimension is used as minor ordering forming an additional four groups for each of the D_CUSTOMER groups, leading to a total of 16 groups in ORDERS. Knowing about the structure of such a clustered table (with meta information), it is fairly easy for a scan to generate all possible orderings, namely <D_DATE>, <D_DATE, D_CUSTOMER>, <D_CUSTOMER>, <D_CUSTOMER, D_DATE>. However, co-clustering multiple fact tables at a time requires more flexibility as dimensions are needed at different granularities and in different tables. Details follow in Sect. 5.

The concept of co-clustering provides advantages in schema design, query execution and update handling.

(i) *Schema design* Due to the power of co-clustering, we see BDCC as a framework that is suitable for a workload-agnostic schema design process. In the end we expect to be able to provide a single replication-free schema that delivers very high and robust performance across a wide range of queries. Following the two guidelines above, we also expect to be able to derive such a schema automatically based on few design hints given by the DBA.

(ii) *Query execution* The goal of BDCC is to deliver fast and robust query processing. Due to the co-clustered table layout, we expect benefits to show for selection pushdown as BDCC not only supports selections for multiple dimensions as most other multi-dimensional clusterings but also supports the propagation of selection between multiple fact tables at the same time. Further, we expect speedup of operators that typically benefit from partitioning, e.g., HashJoin, Aggregation/Grouping and Sort. But we expect not only speedup, but also robustness in query execution, due to a lower memory footprint of each query executed with BDCC.

(iii) *Update handling* The way we designed BDCC, we included life cycle management, supporting not only

the addition but also the removal of batch data. Also, forward growing dimensions can be handled by BDCC without data reorganization.

3 Related work

In order to meet requirements of efficient processing of analytics workloads, database researchers and vendors have come up with many solutions. Most of them agree that the underlying data organization is of utmost importance and needs to facilitate efficient query execution and update handling. Research in this area covers a broad range of different solutions to improve performance of data access which can be classified into indexing, clustering, partitioning, and materialization.

One very important field to name here is certainly *indexing*. The B-tree [9] in all its variants, bitmaps [10] and other structures, e.g., [23,32], mainly focus on efficient access to the data and typically are used for different attributes but on a single attribute (or dimension) at-a-time basis. As a result systems require multiple indexes at a time, leading to redundancy in storage and significant processing overhead for updates. However, first of all it needs to be decided which indexes to create. The index selection problem [11] has been widely studied in the literature. In addition, before query execution an optimizer needs to decide on which indexes to use for table access, leading to additional processing and implementation overhead for a database system.

Orthogonal to indexing, *partitioning* strategies have been developed, e.g., [12,18]. Here, data are split into multiple groups accruing to attribute ranges (range partitioning) or hash-based functions (hash partitioning). Besides solving the space problems on single machines, efficient data access and query execution are a driving factor for developing these strategies. Partitioning for one part helps to reduce the volume of data that needs to be accessed (although this is almost exclusively relevant in range partitioning) but in addition is also used for a more efficient query processing. The focus here is mainly on join processing by exploiting the partitioning to only join matching partitions.

In *table-partitioning* schemes (Teradata [1] and many others), tuples are partitioned across disk storage units, allowing multiple processors to scan a relation in parallel [15]. The problem of deriving good partitioning schemes has been studied in physical database design tuning [4,35]. Other partitioning approaches perform dynamic techniques on unpartitioned tables, e.g., based on conditions [14] or selectivities [30]. A recent main memory system by IBM, Blink [6] employs frequency binning to create balanced multi-dimensional columnar table partitions, to aid its column encoding but also exploits these to push down selections and distribute work for parallelism. Table partitioning leads to the creation of some table object for each tuple group, which carries overhead, limiting the granularity typically to hundreds of partitions. Such coarse granularity leads to less precision for selection pushdown (partition pruning) and leads to partitions that are likely larger than the lowest level CPU cache. Query optimization on partitioned tables in addition may run into the problem of *plan explosion* when separate operators are added to the query plan for each partition, for which a solution is proposed in [18]. Optimizers of commercial systems implement partition pruning, e.g., partition-wise joins for pairs of tables which are co-partitioned [12,26]. Herodotou et al. [18] describes optimization techniques for partitioning, partially treating it as a logical property of a relation, where we start out with a logical partitioning. This way they achieve optimizations higher up in the query tree, something that is deeply integrated in BDCC query optimization and execution. BDCC avoids plan expansion (/explosion) altogether based on the sandwich approach explained in Sect. 6.3 and in detail in [7].

As selections in analytical workloads are typically multidimensional, i.e. restrictions apply to multiple attributes at the same time, simple forms of indexing have limitations and multi-dimensional indexing became a focus of research, and typically realized as a multi-dimensional *clustering*, i.e. the tuples with common characteristics are stored together on disk. The UB-tree, MDC or MDAM for B+-trees supports access via multiple attributes, MDC and the UB-tree [25] even balance dimensions and this way do not lose performance for data access when the selections occur on minor dimensions. Multi-dimensional clustering [29] (MDC) in DB2 takes a physical approach by partitioning row-organized data according to multiple dimensions into separate disk pages. MDC supports several specialized operators such as block index scans as well as ANDing/ORing bitmaps of block identifiers. The use of physical pages as cell units in MDC has the drawback that in skewed data distributions some pages will be mostly empty. Adjoined dimension column clustering (ADC) [13] was proposed in the context of column stores, but assumes major–minor dimension ordering only. Also, ADC unlike BDCC does not support to flexibly determine the access granularity depending on column density as described in Sect. 6.1. ADC does propose clustering with dimensions reachable over foreign keys, but ADC nor MDC co-cluster tables for join processing.

MADM [22] describes the usage of a B-tree for multidimensional indexing and clustering. All the advantages of MDAM, e.g., range-predicate support on leading or intervening access key columns, IN list support, access with missing predicates on leading or intervening columns and so on, are supported by BDCC. In addition, BDCC provides a flexible access granularity to the data and considers co-clustered table setups.

Clustering based on *Z*-order indexing, in contrast to table partitioning, avoids the creation of separate table objects per group and can handle millions of fine-grained groups. Our automatic schema creation algorithms use the concept of *Z*-ordering introduced in [28] and previously applied in Mistral [25], which explores bit interleaving in *Z*-order for multi-dimensional clustering. We add to this the insight that *Z*-order addresses the density difference problem of column stores, and study co-clustering of multiple tables and its query optimization opportunities. The framework BDCC also goes beyond *Z*-ordering as any desired bit interleaving is possible. Another multi-dimensional ordering based on Hilbert curves is used in Netezza [3] to cluster base tables, and "zone maps" can restrict scans to specific table ranges; but this clustering does not accelerate other query processing operators besides selections.

A third research field in this area is *materialization*, i.e. pre-calculation of (intermediate) results to accelerate query execution. However, this results in additional storage requirements and significant overhead for updates, not to mention the design problem of which views to materialize. Column stores have taken a slightly different approach that is closer to replication and is called *ordered projections* [16]. Different (groups of) columns are stored in various orders, this way providing efficient access to the data by different attributes. But, ordered projections come with all downsides of replication.

4 Physical organization of BDCC

In BDCC we co-cluster relational tables in order to share (at least partially) dimensional information between tables that are connected over foreign key relationships. In addition, we re-organize each relational table according to local and foreign key connected dimensions. However, co-clustering multiple fact tables by the various available dimensions throughout a schema is not straightforward. Each table has a maximum number of clusters that can be created before clusters become too small to be read from disk efficiently. For magnetic disks, this limit is about 2 MB and for solid state drives about 32 KB [8]. This limits a clustering for a 32 GB relation (or 32 GB column in case of columnar storage) to one million groups/clusters. A limit that is easily reached. For example sales fact data that are clustered by 100 months $\times$ 100 product types $\times$ 25 customer nations $\times$ 25 supplier nations leading to 6.25 million groups.

Additionally, as for each fact table the number of tuples and the number of dimensions (and the dimensions themselves) vary, an optimal choice for a clustering may require to either leave out dimensions or use the same dimension at different granularities in two different tables. Choosing dimensions becomes a gamble without the proper a-priory knowledge, e.g., analysis of a representative query set. In order to be able to vary a dimension's granularity, it is not suitable to simply adjoin dimension columns to a table as proposed in ADC [13], as the hierarchy attribute with the proper granularity may not exist. This is why it is necessary to abstract from physical columns and create dimensions.

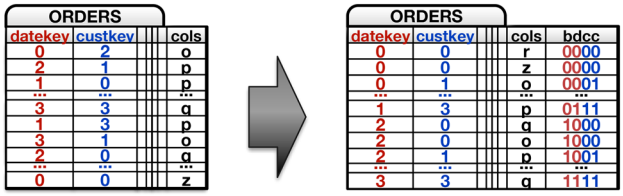

Fig. 3 Example of a BDCC table `ORDERS`, according to the example introduced earlier. An additional column *_bdcc_* is added and `ORDERS` is sorted after *_bdcc_*

In BDCC each table T is replaced by a clustered table version T_{BDCC}, where clusters are formed by consecutive tuples with the same dimensional characteristics. These characteristics are summarized in one clustering key, T_{BDCC}.*_bdcc_*, by mapping the various dimensions onto it. The table is stored as a sorted sequence[1] on T_{BDCC}.*_bdcc_*. This way tuples with equal values in T_{BDCC}.*_bdcc_* are clustered on disk, memory pages and cache lines. All necessary technical details for such a BDCC clustering follow in this section.

Figure 3 sketches how the example table `ORDERS` is modified until it becomes a BDCC table. A new column *_bdcc_* is derived from the two dimension identifiers `datekey` and `custkey`, by mapping `datekey` to bits 2 and 3 of *_bdcc_* and `custkey` to bits 0 and 1. Table `ORDERS` is then sorted after *_bdcc_*.

In order to define a BDCC table, we first

(i) define what exactly a dimension in BDCC is and
(ii) how such a dimension is used for clustering.

A BDCC dimension is an order-respecting surjective mapping from a subset of attributes K (the dimension key) of a given relation onto a finite sequence of identifiers. In other words, a single identifier (bin number) is assigned to each value of the considered key K, an identifier may be assigned to multiple key values, and smaller identifiers are assigned to smaller key values.

Definition 1 (*BDCC dimension*) A BDCC dimension $\boldsymbol{D = \langle T, K, S \rangle}$ is defined over a dimension key $K(D) = K = \langle attr_1, \ldots, attr_s \rangle, s \geq 1$, of table $T(D) = T$ as a sequence $S(D) = S = \langle \langle n_1, V_1 \rangle, \langle n_2, V_2 \rangle, \ldots \langle n_m, V_m \rangle \rangle$ of $m(D) = m = |S|$ dimension entries. Each entry consists

[1] In Sect. 7 we show that efficient updates can be accommodated by relaxing BDCC storage from one sorted sequence to a limited set of sorted runs.

D_DATE

binnr	max_value	unq
0 {000}	1993/03/02	0
1 {001}	1994/02/01	0
2 {010}	1995/01/23	0
3 {011}	1995/11/15	0
4 {100}	1996/08/02	0
5 {101}	1997/06/07	0
6 {110}	1998/01/07	0
7 {111}	no max	0

D_NATION

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

Fig. 4 Example dimensions `D_DATE` created on `o_orderdate` and `D_NATION` created on {`n_regionkey`, `n_name`}

of a bin number n_i and a bin (set) of values V_i such that $\cup_{i=1}^{|S|} V_i = \{v | v \in T \cdot K\}$. Further:

(i) dimension entries are in *ascending order*:
$\forall 1 \leq i < j \leq |S| : n_i < n_j \wedge \text{MAX}(V_i) < \text{MIN}(V_j)$.
(ii) dimension entries *never overlap*:
$\forall 1 \leq i < j \leq |S| : V_i \cap V_j = \emptyset$.

Based on this definition, we further introduce some naming conventions and characterizing functions:

- a *bin* $\langle n_i, V_i \rangle$ is *unique* if V_i is a singleton ($|V_i| = 1$).
- $bin_D(v) = n_i$ is the *bin number* of value $v \in V_i$ in dimension D.
- $bits(D) = \lceil \log_2(|S|) \rceil$ is the *dimension granularity*, i.e., the number of bits needed to represent the bin numbers.
- a dimension $D|_g$ with the *reduced granularity* of $g < bits(D)$ bits is derived from dimension D if one chops off the $(bits(D) - g)$ least significant bits of all bin numbers from D and unites all bins that now have the same number.

In our implementation, dimensions are represented by an array of $|D|$ triples $\langle$`binnr,max_value,unq`$\rangle$, where `max_value` is an inclusive upper bound (empty for the last bin) and `unq` is a boolean stating whether the bin is unique. The bin boundaries are chosen such that the binning is as frequency balanced as possible. We achieve this by combining histogram information with Hu–Tucker [19] encoding, explained in detail in Sect. 5.1. Figure 4 shows two simplified example dimensions for TPC-H. `D_DATE` has a granularity of 3 bits with eight non-unique bins. `D_NATION` was created on $\langle$`n_regionkey, n_name`$\rangle$ and has 10 bins of which 2 are not unique; e.g., the values Argentina and Brazil fall in the Canada bin, but only the bin range boundary Canada is stored. Note that only three regions are used for simplicity. The values in `D_NATION` are combinations of a foreign key to the `REGION` table (represented here by `r_name` for better readability) and a nation name. To determine the order in compound dimension keys, we use lexicographic ordering.

In many schemas, no matter whether star, snowflake or galaxy, the dimension key is typically not found in the fact table but is rather a key in a dimension table reachable over foreign keys. As we propose to co-cluster multiple fact tables and use the same dimensions, it becomes necessary to define the relationship of each dimension and the fact table using a *dimension path* P.

Definition 2 (*Dimension path*) A dimension path $\boldsymbol{P}$ is defined as a (possibly empty) chain of foreign key traversals $P = FK\_T_1\_T_2, FK\_T_2\_T_3, \ldots, FK\_T_{n-1}\_T_n$, from the fact table T_1 to table T_n hosting the dimension key. Here we assume that foreign key relationships have been declared using some identifiers $FK\_T_i\_T_{i+1}$ from table T_i to table T_{i+1}.

In order to support different granularities for each dimension for different BDCC clustered tables, we propose to use a bit mask per dimension and fact table. In addition to choosing the dimension granularity, this facilitates the choice of the exact influence of the dimension in a table's clustering bit by bit. The number of set bits in this mask defines the granularity for the dimension and the positions of set bits specify the positions where the dimension's bin number bits end up the artificial *_bdcc_* key used for sorting T_{BDCC}. This way we can prioritize the dimension by specifying the influence on the ordering of T_{BDCC}. Together with the dimension path, this bit mask defines the concrete usage of the dimension in a T_{BDCC}'s clustering.

Definition 3 (*Dimension use*) To specify a clustering criterion for table T, a dimension use $\boldsymbol{U = \langle D, P, M \rangle}$ combines a BDCC dimension $D(U) = D$, a dimension path $P(U) = P$ that leads from T to dimension key $K(D)$ and a bit mask $M(U) = M$, that defines the granularity that is used for D and the mapping of the dimension's bits to T_{BDCC}.*_bdcc_*. The granularity for dimension D is the number of set bits in M, short $ones(M)$, and $ones(M) \leq bits(D)$ needs to hold.

In order to map/recover dimension keys, respectively their bin numbers, to/from the artificial attribute *_bdcc_*, we introduce two *bitwise extraction functions*: $xtr_M(v)$ that extracts the $ones(M)$ major bits from an integer v and shifts these to the positions of '1's in M, and $xtr_M^{rev}(w)$ that extracts all bits from integer w at positions of '1's in M and condenses these to the right. In the following we use the bitwise operators & (and), | (or), $\ll$ and $\gg$ (shift left/right).

Definition 4 (*Extraction functions*) For a bit mask M, let p_i, $0 \leq i < ones(M)$, be the positions of set bits (i.e. '1's) in M from minor to major and counted from 0 to the length of M minus 1.

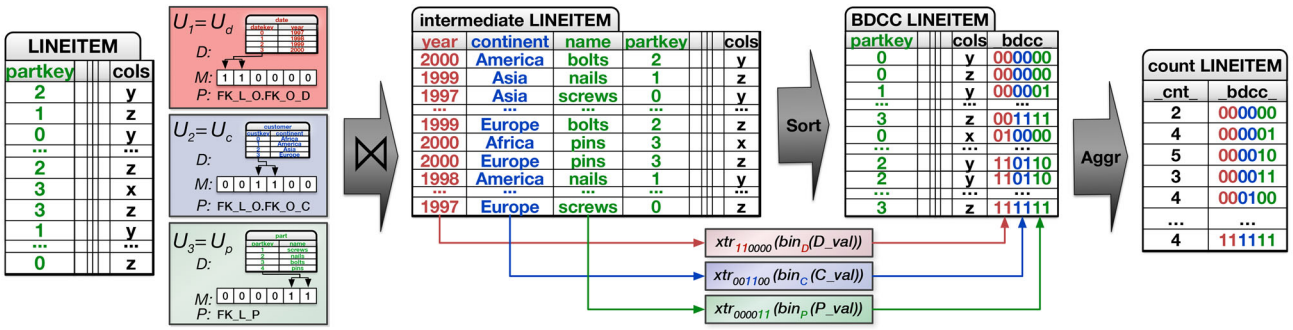

Fig. 5 Creating BDCC table `LINEITEM`. First join `LINEITEM` with the dimensions, then extract bits from the dimension number and merge these into a new cluster key *_bdcc_*. Finally, sort `LINEITEM` and create the count table

(i) Let $v = v_{n-1} \dots v_0$ be an n bit wide integer, $n \geq ones(M)$ (including possible leading '0's). We define:

$$\boldsymbol{xtr_M(v)} = (v_{n-1} \ll (p_{ones(M)-1}))| \dots | (v_{n-ones(M)} \ll (p_0))$$

(ii) Let m be the number of bits in M, $m = bits(M)$ and let $w = w_{m-1} \cdots w_0$ be an integer of m bits (including possible leading '0's). We define:

$$\boldsymbol{xtr^{rev}_M(w)} = (w_{p_{ones(M)-1}} \ll (ones(M)-1))| (w_{p_{ones(M)-2}} \ll (ones(M)-2))| \dots |w_{p_0}$$

For example, given Mask $M = 1001001$. Then $p_0 = 0$, $p_1 = 3$, $p_2 = 6$ as there are three set bits in M. Also given a 5-bit number $v = 20$ (i.e. $v_4v_3v_2v_1v_0 = 10100$), then $xtr_{1001001}(20) = 65$ as it extracts the three major bits $v_4 = 1, v_3 = 0, v_2 = 1$ of v (because of the three 1-bits in M) and assembles bitwise $v_400v_300v_2$, which is 65 (i.e. **1000001**). Reverse we get $xtr^{rev}_{1001001}(65) = 5$, extracting $w_6 = 1, w_3 = 0, w_0 = 1$ at the positions where M has set bits and assembling in extraction order (101).

The BDCC table definition is now straightforward:

Definition 5 (*BDCC table*) A BDCC table $\boldsymbol{T_{BDCC} = \langle T, U_1, \dots, U_d, b \rangle}$ clustered on b bits is defined over a source table T by specifying d dimension uses $U_1, \dots, U_d$ under the constraints

(i) *all* $\boldsymbol{b}$ *bits are set*:
$M(U_1)| \dots |M(U_d) = 2^b - 1$
(ii) *no bits overlap*:
$\forall i, j : 1 \leq i < j \leq d \wedge M(U_i) \& M(U_j) = 0$

Each tuple $t_{BDCC} \in T_{BDCC}$ is a copy of $t \in T$ with an additional attribute value

$$t_{BDCC}.\_bdcc\_ = xtr_{M(U_1)}(n_1)| \dots |xtr_{M(U_d)}(n_d),$$

where for each dimension use U_i, we look up bin number $n_i = bin_D(t.P.K)$ for the original tuple $t \in T$, with $P = P(U_i)$, $D = D(U_i)$, $K = K(D)$. This lookup is a join along dimension path P, leading to dimension key K. The value k of key K for tuple t is then looked up in the dimension entries $S(D)$ and mapped to $t_{BDCC}.\_bdcc\_$. T_{BDCC} is sorted on *_bdcc_* and replaces T.

In addition, a metadata table T_{CNT}(*_bdcc_*,*_count_*) is created, counting the frequencies in $T_{BDCC}.\_bdcc\_$.

Example Before we explain how to design a BDCC schema, recall the example table `LINEITEM` of Fig. 1. Its BDCC creation is depicted in Fig. 5, clustering it on $b = 6$ bits with dimension uses: `D_DATE` $U_1 = U_d$, `D_CUSTOMER` $U_2 = U_c$ and `D_PART` $U_3 = U_p$, which are based on the dimension tables in Fig. 1:

U_d: using all 2 bits d_2d_1 of dimension `D_DATE` over dimension path `FK_L_O.FK_O_D` and bit mask 110000,
U_c: using the 2 major bits g_2g_1 of `D_CUSTOMER`, via dimension path `FK_L_O.FK_O_C` and bit mask 1100,
U_p: using bits p_2p_1 of `D_PART`, but now over dimension path `FK_L_P` and bit mask 11.

Based on these three-dimensional uses, `LINEITEM` is joined with the dimensions in an intermediate step. Then `LINEITEM` is extended with the *_bdcc_* column, which is the bitwise OR of the bits extracted from the dimension bin numbers that are looked up by the $bin_D()$ functions. After the *_bdcc_* column is created, `LINEITEM` is sorted after this column, leading to its BDCC version. In addition to the BDCC version, also a count table is created, storing the number of tuples per *_bdcc_* value.

To explain in detail, take for example the first tuple $t = (2, \dots, y)$ of table `LINEITEM`. According to dimension uses U_1, U_2, U_3, the tuple is joined with three dimensions leading to $t =$ ("2000", "*America*", "*bolts*", $2, \dots, y$). Looking up each bin leads to $bin_{D(U_d)}$("2000") $= 3$, $bin_{D(U_c)}$("*America*") $= 1$ and $bin_{D(U_p)}$("*bolts*") $= 2$.

Applying the extraction functions to these bin numbers leads to $xtr_{110000}(3) = 110000$, $xtr_{1100}(1) = 100$, $xtr_{11}(2) = 10$. The bit-wise OR of these results defines the bddc value of this tuple, $t.\_bdcc\_ = 110110$.

Note that in this example dimension bin numbers and join keys are the same, as dimension tables and the abstract dimensions are the same for simplicity. This is not necessarily the case and requires to execute the join and to perform the lookup on the dimension key.

5 Design of a BDCC schema

In Fig. 5 we have seen, how a BDCC table is derived from a normal table based on given dimension uses. However, designing a fully co-clustered BDCC schema is more complex. First of all, dimensions need to be identified and created. Here, it is critical to balance dimensions across the whole schema, as they are typically used in different fact tables. Second, the explicit use of each dimension per table needs to be defined, which depends on the position in the schema, the table size and the number of the applicable dimensions for that table. Third, each table needs to be reorganized according to the defined dimension uses. In addition, the count table per BDCC table needs to be created.

We see BDCC as powerful enough to provide a replication-free schema design with fast and robust query performance. In order to achieve this robustness, we automated BDCC design, based on few design hints given by the DBA. We interpret CREATE INDEX statements as hints for interesting clustering columns and take these into account during our design process. In addition, we rely on existing foreign key declarations.

5.1 Creating a BDCC dimension

In a data warehouse, one can typically obtain statistics on the frequency distribution of dimension values; e.g., by creating a histogram over dimension values occurring in a fact table. To give BDCC predictable splitting power (ideally a factor of 2 per bit) and to avoid the "puff pastry" [25] effect, where one dimension has a much lower splitting power than the other, it is desirable to bin a fact table evenly. We provide two basic methods to create a dimension and show how combining both leads to a robust algorithm to balance dimensions.

The *histogram method* to create dimension D of granularity $bits(D)$ relies on creating an *equi-height* histogram of $2^{bits(D)}$ buckets. A bin b_j corresponds to a bucket and fits our implementation as follows: the bin number is `binnr` $= j$, the attribute `value` holds the inclusive upper bound of bucket j and the `unq` property is `false` if upper and lower bound differ.

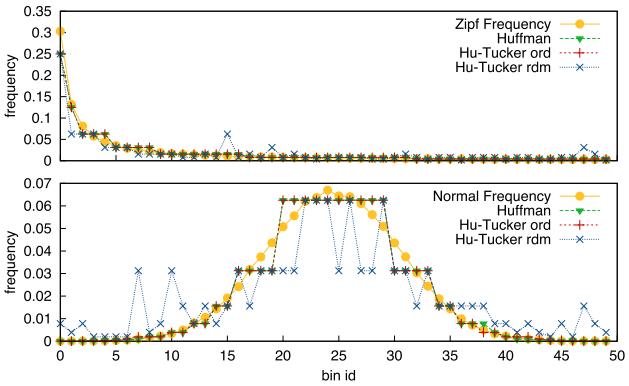

Fig. 6 Modeling of Zipf and normal distribution by Huffman and Hu–Tucker

In case of skew, however, we may get a bucket containing a single value, where its frequency (far) exceeds the average bucket size. Using such a dimension at full granularity is fine, it becomes critical, however, when the dimension is used at lower granularity (as BDCC does very often), and the lowest bit of the bin numbers is cut off. That means, that the "over-size" (very frequent) bin gets combined with a direct neighbor, and as a result any selection pushdown on the direct neighbor will result in reading all tuples related to the oversized bin, resulting in a very poor hit ratio for this pushdown. In order to avoid such combinations of bins, we use Hu–Tucker compression [19] during dimension creation, which encodes value v with a code c whose length $|c|$ inversely approximates the value frequency $f(v) \approx 2^{-|c|}$. It can be thought of as the order-respecting variant of Huffman coding [20]. Figure 6 shows the frequencies for two dimensions that are designed such that the values of its 50 bins exactly follow two typical skewed distributions: Zipf (above) and Normal (below). For each dimension, we also plot the "approximated frequencies" $2^{-|c_i|}$ produced by Hu–Tucker and Huffman encoding, where $|c_i|$ is the code length produced for the separator `value` of bin i. This shows that even if value frequency is totally uncorrelated with value order (a random permutation, lines *rdm*) Hu–Tucker code length is clearly correlated with value frequency.

Our *coded method* for dimension construction uses the Hu–Tucker algorithm [19]. It creates a binary tree, where the values are the leaves, and the root-to-leaf path determines the code (0 = left, 1 = right). An example tree is shown in Fig. 7 for the `D_NATION` dimension from Fig. 4. Frequent values, such as `China` and the `USA`, get a short code and are high up in the tree; infrequent values end up in the lower part. We cut this tree at height $m = 4$; each leaf of the resulting tree becomes one bin, using the largest leaf value below it as separator `value`. This way `Canada` becomes separator for bin `0000`. The `binnr` is equal to their Hu–Tucker code, suffixed with 0's if it is shorter than m bits. Nodes where children have been cut off have `unq` = `false`.

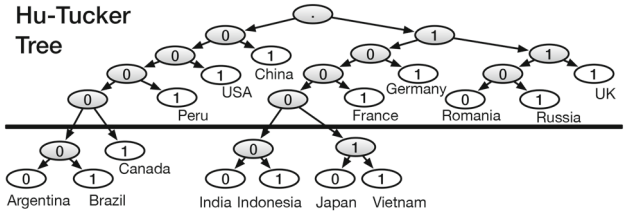

Fig. 7 Hu–Tucker tree for dimension D_NATION of Fig. 4

Hu–Tucker example The goal of the coded method is to avoid a large "false hit" factor when a dimension is used at a *reduced* granularity. Consider a skewed distribution where the lowest value (e.g., NULL) occurs 69 % of the cases, and the other 31 values occur just 1 % of the cases. The histogram approach would create D at $bits(D) = 5$ bit granularity with bins $0 \ldots 31$, where bin 0 holds the frequent value. If this dimension was used in its reduced form $D|_3$, then 28 of the 31 infrequent bins would get combined into 7 bins, each with frequency 0.04. The three "unlucky" bins 1, 2, 3 would get combined with bin 0 into one bin with frequency 0.72. If we now process a selection on such an unlucky value, BDCC would not be able to reduce the scan much, since bin-reduction caused a false hit factor of 72 (.72/.01). For the lucky bins, this false hit factor is 4, which is expected as the dimension was reduced by 2 bits. Such random degradation in unlucky cases is highly undesirable for users and is in conflict with our goal of delivering robust query processing.

In this example, Hu–Tucker would create D with just 17 bins (not 32), the frequent first bin with bits 00000 and 16 more bins from 10000 to 11111. It would thus already have an average false hit factor of 1.9375 (31 values mapped on 16 bins) for all bins but the first. In $D|_3$ we would only get 5 bins: 000 and 100 ... 111, so the maximal false hit factor is 7.75 rather than 72.

Dimension creation algorithm We require a dimension D to be usable at different granularities $D|_g$ in a range $g \in [e, bits(D)]$. A typical minimum is $e = 3$; i.e. clustering with less than $2^3 = 8$ groups is not worth the effort. This leads to our final algorithm:

Algorithm 1 (*Dimension creation*) A dimension D of $m = bits(D)$ bits is created from a set of key values V, by first computing an equi-height histogram $H(V)$ of 2^e buckets. For all $1 \leq j \leq 2^e$ each bucket $H[j]$ is then split again, by using those values V_j falling in that bucket to create another equi-height sub-histogram $H_j(V_j)$ of 2^{m-e} buckets. If no skew is detected in the resulting bins at granularity m ($\forall i : H_j[i] \approx \frac{H[j]}{2^{m-e}}$), we are done for this bucket. Otherwise, the bucket is split with the coded method instead—using the values and frequencies in H_j as input for the Hu–Tucker algorithm. The final bin list is the concatenation of all bins created for each bucket j, where the final `binnr` has its major e bits set to j, while its minor $m - e$ bits contain the computed bin number when splitting the bucket.

During BDCC creation all dimension uses need to be identified at first. Then dimensions are created one by one, using some fixed maximal granularity (e.g., $bits(D) \leq 15$). In order to take into account the distribution of dimension values across all tables $T_i, i > 0$, where the dimension is used, the set of key values V is the union of all dimension key projections after joining the dimension key with each table T_i according to the dimension path P_i. Note that a dimension may be used more than once for the same table, then each dimension use is treated as if it was a new table, adding keys to V.

The skew-induced so-called "puff pastry" problem where the splitting power of one dimension is much bigger than of the other was addressed in related work by frequency balancing the dimensions with so-called "split point trees" in VUB-Trees [25]. However, the VUB work did not discuss how to achieve frequency balancing in case of highly skewed values, this is solved here by using Hu–Tucker encoding.

That said, our approach cannot fully solve the problem of eliminating skew, because of updates on the one hand and when the frequency distribution used to create a dimension cannot fully match *all* uses of that dimension in a particular schema on the other hand. For instance, considering `D_DATE`, which is used to cluster both example tables `ORDERS` and `LINEITEM`, it might be that most tuples of `ORDERS` relate to 1999, but most items in `LINEITEM` relate to 2000. This is the case, when for 2000 an order has many foreign key related items, where for 1999 an order only has few items. In such a scenario, the correlated join hit ratio introduces skew.

5.2 Creating a BDCC table

With BDCC we try to cluster into as many groups as possible. The more the groups, the better the precision for selection pushdown and the more efficient partitioned operator execution becomes. However, there is a limit when more groups do not provide any more benefit while introducing more overhead. With the way BDCC is designed, this limit is typically reached, when clusters become smaller than block access units. Because then, scanning BDCC organized data results in repetitive random data access below the "efficient random access" granularity. For further observations assume that an efficient random access granularity A_R can be found for each I/O system. For magnetic disks, A_R is typically around 2 MB, for SSDs it can be around 32 KB and for main memory, it is even more fine granular. With A_R given by the system, the total number of bits that can efficiently be used for a table is already fixed and can simply be approximated by $N = \log(input\_size/A_R)$, where $input\_size$ is the

total size of the largest/widest column of a table in bytes. However, the way how these bits are assigned to dimensions is still an open question. Before this, another question is of importance.

Which dimension order to choose? Classical multi-dimensional approaches like MDAM [22] or ADC [13] require a DBA to order the dimensions from major to minor. This favors access along major dimensions as the granularity of I/O access for (selections on) minor dimensions is very small (scattered). BDCC can use any bit interleaving, hence also major–minor; For applications with clear major dimensions, this approach is fine.

However, major–minor ordering has as disadvantages that (i) dimension order is a knob that might get tuned wrongly, (ii) major dimensions get a much better access pattern than minor and (iii) the question of which dimensions to place where complicates an automatic design process. Following the UB-Tree work [25], we prefer *round-robin bit interleaving* instead, storing tuples in *Z-order*. This eliminates the task to order a table's dimensions and provides fast access for all dimensions.

From Definition 5 of a BDCC table, it follows that for each table of the schema a number of dimension uses and the clustering depth need to be specified. This boils down to providing a dimension path to each dimension and an interleaving pattern of all dimensions, expressed by the masks of the dimension uses. Fixing the interleaving pattern to be round-robin, we liberate a DBA from inferring about BDCC bits. Assuming a given set of used dimensions for a table T, we provide a *self-tuned* algorithm that automatically creates a round-robin clustered BDCC table T_{BDCC}. The idea is to bulk-load BDCC tables initially at a *maximal* granularity, but then to only create metadata (the count-table) on a lower granularity; exploiting statistics gathered during bulk-load. This keeps the count-table small for offset calculations and access to T_{BDCC} efficient.

Algorithm 2 (*Self-tuned BDCC table*)
Input: Table T with the original data.
Dimension uses $\{U_1, \ldots, U_k\}$ with empty masks.
Output: BDCC clustered table T_{BDCC} and T_{CNT}.

(1) *Generate masks* $M(U_i)$ Set $M(U_i)$ so that dimensions are round-robin interleaved in some arbitrary order, assigning one bit at a time (major to minor) per foreign key or local dimension. If two dimensions are used over the same foreign key, bits assigned over this foreign key are distributed round robin to each of these dimensions. This assures that all foreign key joins of the table are equally accelerated. Assign bits until all $B = \sum_1^k bits(D(U_i))$ dimension bits are used: the number of 1-bits in all masks is maximal.

(2) *Create the BDCC table* T_{BDCC} Compute the *_bdcc_* column with derived masks $M(U_i)$, store T_{BDCC} sorted on *_bdcc_* and analyze group sizes in a piggy-backed aggregation. Discard source table T.

(3) *Create the count table* T_{CNT} For the column—or group of columns when PAX or row wise storage is used—with the highest density (size on disk), choose the largest granularity $b \leq B$ such that the size in bytes of most *_bdcc_* groups is above the efficient random access size A_R. Create T_{CNT} with granularity b, in a single ordered aggregation, counting tuples with equal value $\_bdcc\_ \gg (B - b)$ (T_{BDCC} sorted on *_bdcc_* at granularity B is also clustered for b).

Note that in (1) other options are possible. One could simply round robin interleave all dimensions without respecting the foreign key, or each foreign key could be weighed according to size/cost of the resulting join, detailed weights could be calculated by workload/data analysis. As our goal is simplicity and robustness for many workloads, we advocate looking at foreign keys.

5.3 Creating a BDCC schema

The question remains, where dimensions for clustering a table come from. In our schema design approach we infer a co-clustered schema from design hints that identify foreign key joins and indicate that access to certain columns is important, just like in classic DDL.

Algorithm 3 (*Semi-automatic schema design*)
Input: Existing database schema with tables and data.
`CREATE INDEX` $(I_1, \ldots, I_Z)$ `on` T statements.
Foreign keys declarations.
Output: co-clustered BDCC database.

(1) *Generate initial dimension uses* Traverse the schema DAG (projection) from the leaves, identifying relevant dimensions and dimension uses. Observe for each table T its index declarations. If $\{I_1, \ldots, I_Z\}$ equals a foreign key, inductively add all dimension uses of the referenced table T_{fk} also to T, putting the FK-id in front of the dimension paths ($P = FK\_T\_T_{fk}.P_{fk}$). Otherwise, identify a new dimension with key $\{I_1, \ldots, I_Z\}$, and add a dimension use to T.

(2) *Create the dimensions one by one* Use a fixed maximal granularity derived from the usage and the number of distinct values of a dimension and create each dimension using Algorithm 1 from Sect. 5.1.

(3) BDCC*cluster each table* Cluster table by table at a self-tuned granularity using Algorithm 2.

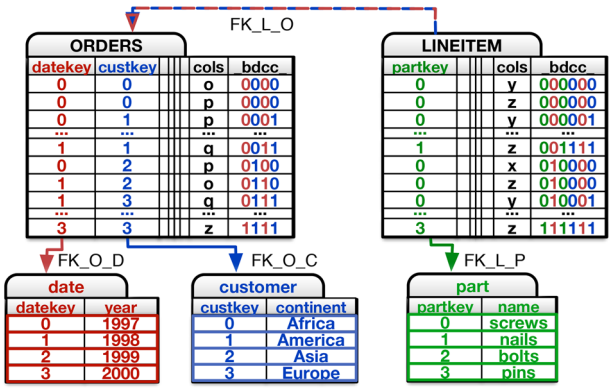

Fig. 8 Example of a BDCC schema, according to the example introduced earlier

We are aware of a limitation of Algorithm 3: on very large schemata (much larger than TPC-H), with many tables, foreign keys and index declarations (=hints), it will identify *too many* dimension uses per table. For example, in a table with 8G tuples (2^{33}) with a widest column of 64 bytes per tuple (2^6), and an efficient random access size $A_R = 32$ KB (2^{15}), one can use in total $33 + 6 - 15 = 24$ bits to cluster on, as with $2^{24} = 16$ M groups, each group of values for that widest column then takes $A_R = 32$ KB, hence can be read efficiently in scatter scans. One could cluster on 24 dimension uses of 1 bit each, but more realistically is limited to 5–8 dimension uses (3–5 bits each). This results in a maximum of 8× to 32× I/O reduction by selection pushdown and already significant acceleration and memory reduction for processing queries using BDCC (detailed explanations follow in Sect. 6), even for only a single dimension involved. Extending Algorithm 3 is beyond our scope here, directions are (i) ignore dimension uses with less impact on a workload, or (ii) re-consider replication and create multiple BDCC replicas (a question will be which dimensions to use for which replica).

Figure 8 illustrates, what these algorithms achieve, when being applied to the example introduced in Fig. 1. With CREATE INDEX statements on attributes `D_DATE.year`, `D_CUSTOMER.continent` and `D_PART.name` our algorithms cluster `ORDERS` by its two foreign key connected dimensions `D_DATE` and `D_CUSTOMER`, and `LINEITEM` by `D_DATE`, `D_CUSTOMER` and additionally by `D_PART`. For `LINEITEM` `D_DATE` and `D_CUSTOMER` are used because of the foreign key connection to `ORDERS` and the use of these dimension there. Assuming enough tuples in the tables, the algorithms use all possible bits.

In Sect. 9 we show detailed results of applying these algorithms to the full TPC-H schema. There, the number of used bits per dimension and per table is actually limited by the random access size A_R, showing all effects of the automatic tuning approach.

6 Query processing

With BDCC we combine advantages from indexing and partitioning with the goal of faster and more robust query processing. Enhanced with ideas from the concept of co-clustering, BDCC provides the following query optimization strategies:

(i) *Selection pushdown* selection predicates on one or more dimension keys can be pushed down to the scan on a clustered fact table and the data volume is reduced directly at the source.
(ii) *Selection propagation* a selection predicate is not only pushed down to one fact table but also to other fact tables that are foreign key connected.
(iii) *Selection pushdown and propagation on correlated attributes* a selection predicate on an attribute that is correlated to a dimension key is pushed down to the fact table scan and is also propagated to foreign key connected fact tables.
(iv) *Join elimination* a join to a dimension table is not executed when the selection predicate can be pushed down to the fact table and no additional attribute from the dimension table is needed for further query processing.
(v) *Join acceleration* a foreign key join between two fact tables is executed as a partitioned join while fully reusing the join operator itself.
(vi) *Sort, Group by, Aggregation acceleration* if any one of these operators is executed over an attribute that determines a dimension key, it can be executed as a partitioned operator, again fully reusing the original operator.
(v) *Scan adaption* the `BDCCscan` access granularity is automatically tuned according to the scanned columns and their densities.

The different optimizations are realized by the interaction of multiple parts of the system. In this section we will focus on three parts, namely `BDCCscan`, a scan operator suited to the requirements of BDCC, the Sandwich Operators, two operators PartitionSplit and PartitionRestart, that very efficiently implement strategies of classical partitioning for BDCC and the query optimizer that in addition to introducing `BDCCscan` and Sandwich Operators has to become co-clustering aware in order to make BDCC fly. Especially the Sandwich Operators contribute to the robustness of query processing, as they, among accelerating queries, drastically reduce memory consumption and, thus, permit a highly improved concurrent query execution.

6.1 BDCC scan

Storing data as multi-dimensional co-clustered data is only one part, efficient and flexible access is another. A scan not

only needs to support selection pushdown but also needs to serve out data in any desired dimensional order, which is important for further query processing. When two co-clustered tables are joined or a grouping, aggregation or sort is performed over a clustered table it is possible to accelerate these operators by applying query processing techniques similar to what is found in table partitioning. However, these techniques can only be applied when the tables are retrieved in the right or in case of a join in a compatible ordering. Depending on which co-clustered tables are involved in such a join, the required orderings can vary from query to query, imposing high flexibility on the scan.

In order to perform these tasks we introduce a special scan operator, called `BDCCscan`, that can retrieve a BDCC table in any dimensional order of the dimensions involved in clustering this table. In order to perform this task without the overhead of sorting the relation according to the requested order, `BDCCscan` needs to hide this sort activity in the retrieval of data itself. That means a differently requested order implies a different pattern for accessing the data on disk. That is why, we based `BDCCscan` on a fetch scan (sometimes also skip scan), a scan that reads (parts of) a relation from disk based on a number of table ranges given as an extra parameter. Each range consists of a *start* and an *end* row identifier and defines a consecutive part of the relation that needs to be retrieved. Following the order of the given ranges, any skipping or fetching pattern on a relation can be performed. In addition we added an extension to produce an additional attribute that characterizes the current ordering of the retrieved relation. This attribute is a simple group identifier, that is incremented, when the scan reaches the next attribute value of a dimension, and that is later used to identify (matching) groups when applying partitioning techniques on join, grouping or sort operators.

In order to match BDCC we modified the fetch scan syntax. In addition to input table T and a set of columns, it now receives a list of *dimension specifications* defining the fetch order, and performs three extra tasks. First, the translation of the dimension specifications into fetch scan ranges in order to re-use the standard fetch scan. Second, perform the selection pushdown, i.e. drop all ranges that do not match the selection predicates. And third, extract bits from T._bdcc_, create a new group identifier attribute named _gid_ on which the tuple stream is ordered, i.e. _gid_ represents the new dimension order. For _gid_ definition *any* desired interleaving of the extracted bits can be used.

Definition 6 (*Dimension specification*) A dimension specification $\boldsymbol{S = \langle U, M, l, h\rangle}$ is a quadruple, where $U(S) = U$ is a dimension use, $M(S) = M$ is a bit mask that determines where the bits of $M(U)$ in _bdcc_ will go to in _gid_, and $[l, h]$ is a range of bin numbers to filter on ($l(S) = l$ and $h(S) = h$), with $0 \leq l \leq h < 2^{ones(M)}$.

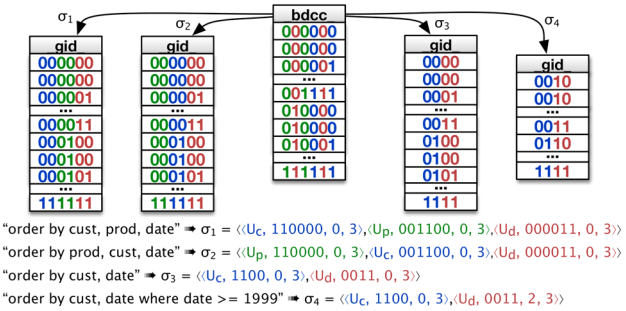

Fig. 9 Different relation orderings produced by `BDCCscan`

Note masks $M(S)$ and $M(U(S))$ are different. The first defines a mapping of _bdcc_ bits to _gid_, the latter how bin numbers are mapped to _bdcc_ at schema creation.

Definition 7 (`BDCCscan`) A `BDCCscan` $\boldsymbol{(T, C, \sigma)}$ gets three parameters: a table T, and a column sequence $C = \langle C_1, \ldots, C_c\rangle$, and a dimension specification list $\sigma = \langle S_1, \ldots, S_k\rangle$. The extracted (partial) dimension number for each tuple $t \in T$ per dimension S_i is denoted $n_i = xtr^{rev}_{M(U(S_i))}(t.\_bdcc\_)$. Computing

$$t.\_gid\_ = xtr_{M(S_1)}(n_1)|\ldots|xtr_{M(S_k)}(n_k),$$

`BDCCscan` produces T sorted on _gid_, but only emits t if it qualifies all range selections: $\forall i : l(S_i) \leq n_i \leq h(S_i)$.

Figure 9 illustrates some of the different orders a `BDCCscan` can retrieve for `LINEITEM` from Fig. 8. For better illustration we only focus on the original _bdcc_ column and the resulting _gid_ column. σ_1 extracts `D_CUSTOMER` as major dimension, followed by `D_PART` and `D_DATE`, while σ_2 has `D_PART` as major dimension followed by `D_CUSTOMER` and `D_DATE`. σ_3 ignores `D_PART` and only retrieves data with major `D_CUSTOMER` and minor `D_DATE` ordering. σ_4 in addition to σ_3 performs a selection pushdown on `D_DATE` only retrieving years 1999 and 2000.

Implementation The `BDCCscan` implementation requires no operator extension—it maps to the sub-plan:

`FetchScan(`T`, Sort(Select(Project(OrdAggr(Scan(`T_{CNT}`)))))`

(i) Observe that the T_{CNT} table, created on some granularity b, can be reduced to a smaller count-table at granularity $b' = b - \delta$ by throwing away the lowest δ bits of the _bdcc_ column and summing the adjacent rows with equal _bdcc_ $\gg \delta$ value (typically 2^δ rows). `BDCCscan` often does not access the table at maximum granularity and thus as a first step reduces the granularity of T_{CNT} using ordered aggregation `OrdAggr`.

(ii) The second processing step is computing _gid_ from _bdcc_, which uses a non-duplicate-eliminating

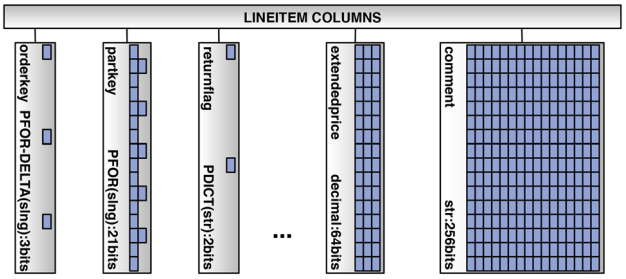

Fig. 10 Different column densities of TPC-H `LINEITEM`

`Project` operator that performs bitwise and/or/shift operations, and as an intermediate step computes the original n_i dimension numbers per tuple. Additionally, a running sum is computed over the bin `counts`, so that we get a row-id (RID) denoting the start and end of each range.

(iii) The next step is applying the range selections $[l_i,h_i]$ on the dimension numbers n_i using a `Select`.

(iv) Then we `Sort` on *_gid_*. Only sorting filtered and aggregated count values, this is no performance issue.

(v) Finally, the resulting stream $\langle$*_gid_*, $\texttt{RID}_{lo}, \texttt{RID}_{hi}\rangle$ is fed into a `FetchScan`, which is a `Scan` that, rather than reading the entire table T, just reads certain ranges $\langle \texttt{RID}_{lo}, \texttt{RID}_{hi}\rangle$ in their given order from T. Such an operator is commonly found in relational database systems. Our `FetchScan` adds *_gid_* to its result stream, so consuming operators can easily detect group boundaries (where *_gid_* changes).

6.2 Adaptive scan: a column-store optimization

Round-robin interleaving has a *column-store specific advantage*, relating to the often huge density differences between table columns. Some queries only access high-density columns, some only access low-density columns, others a mix. As a motivating example, Fig. 10 illustrates the density differences for a subset of the `LINEITEM` columns of the TPC-H benchmark, where differences of up to a factor 120 occur. Disk access to columns with high density (e.g., `comment`) profits from access at a fine granularity (many bits) as more precision results in less data to be scanned and better utilization of the Sandwich Operators, whereas access to low-density columns (e.g., `returnflag`) can only exploit a few major *_bdcc_* bits, as otherwise the access pattern gets too fined-grained for the I/O device and multiple reads will occur inside a block (i.e. thrashing). Each access granularity leads to a different scatter scan order, and therefore a column-store scan faces the dilemma of choosing a single granularity for all columns (as it needs values of different columns to appear in the same order). The ability to adaptively choose the access granularity, given any dimension of interest, is the critical column-store advantage of round-robin bit interleaving. For example, the scan of `LINEITEM` in Fig. 14 uses 9 round robin assigned bits from three dimensions. Another way of assigning the dimension bits could be major–minor ordering ($d_3d_2d_1c_3c_2c_1p_3p_2p_1$, d_i: `D_DATE` bits, c_i: `D_CUSTOMER` bits, p_i: `D_PART` bits). Assume the densities of the columns require to limit disk access to 6 bits for efficiency. Then a query selecting on `D_PART` cannot exploit BDCC in case of major–minor bit ordering, as there are no `D_PART` bits among the major six ($\mathbf{d_3d_2d_1c_3c_2c_1}\ \ p_3p_2p_1$). In contrast, round-robin interleaving allows to exploit two `D_PART` bits ($\mathbf{d_3c_3p_3d_2c_2p_2}\ \ d_1c_1p_1$), reducing I/O fourfold.

Analysis We provide bounds on RAM needs and the amount of `FetchScan` ranges generated by `BDCCscan`, assuming that bin sizes are evenly distributed. This analysis is needed to find an efficient scan granularity for `BDCCscan`. Refining the formulas using bin histogram information is omitted for simplicity.

Corollary 1 *(Number of ranges) In a* `BDCCscan` *of a table clustered on b bits, _gid_ is computed by selecting a number $\gamma \leq b$ of bits from _bdcc_. Let the bit-selection function $X : \{1,\ldots,\gamma\} \rightarrow \{1,\ldots,b\}$ return the source position $X(dst)$ in _bdcc_ for each destination bit dst in _gid_. Comparing the bits of _bdcc_ and _gid_, some bits in _gid_ may have retained their original major position (position from the left). Other bits in _gid_ were shifted in from deeper positions in _bdcc_. The deepest shifted bit from _bdcc_ is $\rho = \text{MIN}(\{b\} \cup \{X(dst) | X(dst) \neq dst + (b-\gamma)\})$. This deepest bit determines the access granularity $g = b - \rho$. At access granularity g there are maximal 2^g different clusters, so $I_{rng} = 2^{b-\rho}$ is the maximum number of ranges.*

A column C_i of N tuples with a data density of Δ_i bytes/tuple and disk block size of A_B gets stored in $|C_i| = \lceil N \cdot \Delta_i / A_B \rceil$ blocks. At ρ smaller than $\omega_i = b - \log_2(N \cdot \Delta_i / A_B)$, range sizes go below block sizes. Since data density between columns commonly varies by two orders of magnitude, the ω_i may differ by up to ≈ 7 bit positions. Scanning with a granularity $\rho \leq \omega_i$, causes `BDCCscan` to revisit each block up to $2^{\omega_i+1-\rho}$ times. Block sizes do not match range sizes exactly so that at $\rho = \omega_i$ each range is typically covered by two blocks; this causes the $+1$ in the above formula. This repetitive access causes exploding I/O cost with decreasing ρ, unless the system caches *repeatedly* accessed blocks. The cache footprint for a too-deeply-accessed column C_i ($\rho \leq \omega_i$) is the amount of accessed blocks times a *reduction factor* due to the access pattern.

The selectivity of a `BDCCscan` (T, C, σ), is the product of selection range sizes divided by the total amount of bins:

$$sel = \prod_{S_j \in \sigma} (1 + h(S_j) - l(S_j))/2^{ones(M(S_j))}.$$

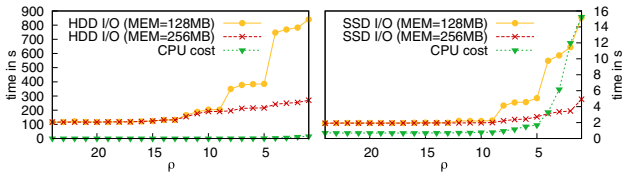

Fig. 11 BDCCscan micro-benchmarks. *X*-axis: scatter scan granularity (finer is lower ρ). *Y*-axis: I/O time (*left legend*), CPU time (*right legend*)

The amount of accessed blocks after selection $|C_i|_{sel}$ is $sel \cdot |C_i|$, but at small sel, blocks are only read partly and it becomes $sel \cdot I_{rng}$. As it cannot exceed column size $|C_i|$, we get $|C_i|_{sel} = \text{Min}(|C_i|, sel \cdot \text{Max}(|C_i|, I_{rng}))$.

Corollary 2 *(BDCCscan memory requirements) To avoid thrashing, a BDCCscan (T, C, σ) needs at most*

$$MEM_{T,C,\sigma} = A_B \cdot \sum_{C_i \in C : \rho \leq \omega_i} |C_i|_{sel} \cdot 2^{\alpha_i - \gamma + 1}.$$

α_i is the major bit in *_gid_* that came from a too-deep position in *_bdcc_*: $\alpha_i = \text{MAX}(\{0\} \cup \{dst | X(dst) \leq \omega_i\})$. The amount of buffering space needed for column C_i is *reduced by factor* $2^{\alpha_i - \gamma}$, as every bit to the left of α_i avoids repetition, halving the required buffer space ($\alpha_i \leq \gamma$). Since ranges do not align with blocks, twice the amount of blocks is needed worst case, hence $+1$.

Micro-benchmarks Figure 11 shows BDCCscan micro-benchmarks on a table of $N = 2G$ tuples, round-robin clustered on four dimensions of 6 bits, resulting in 2^{24} bins of even size: $\_bdcc\_ = a_6 b_6 c_6 d_6 a_5 b_5 c_5 d_5 \ldots a_1 b_1 c_1 d_1$. The x-axis determines a 1-dimensional BDCCscan where the deepest moved bit $\rho = x$. For example at $\rho = 8$ we request $\_gid\_\ a_6 a_5 a_4 a_3 a_2$, as bit a_2 is at position 8 in *_bdcc_*.

The *I/O cost* lines are obtained on a fast solid state disk (SSD raid, right) and a single magnetic hard disk drive (HDD, left), as specified in Sect. 9 using block size $A_B = 32$ KB. While we focus here on results for a column C_1 with $\Delta_1 = 1$ byte/tuple, we ran experiments on columns C_x with many densities, finding similar outcomes; and the expected differences in ω_x. The need for buffering appears in C_1 once ρ reaches $\omega_1 = 24 - \log_2(2G \cdot 1/32\,K) = 8$, because then a BDCCscan requests ranges equal or smaller than the block size A_B. We just access a single column without range selections here ($sel = 1$) so $MEM_{T,C,\sigma} = A_B \cdot |C_1| \cdot 2^{\alpha_i + 1 - \gamma}$. At $\rho = 8$, i.e. $\_gid\_ = a_6 a_5 a_4 a_3 a_2$, the too-deep part is just a_2 so $\alpha_1 = 1$, hence we need $2G \cdot 2^{\alpha+1-\gamma} = 2^{31} \cdot 2^{1+1-5} =$ 256 MB memory to avoid I/O thrashing. These experiments confirm our model: below 256 MB (i.e. 128 MB) I/O cost explodes.

Figure 11 also shows the *CPU cost* of BDCCscan, plotting the *same* values in both graphs, highlighting that CPU overhead only becomes a bottleneck on fast I/O systems such as SSDs. The exponential rise in the amount of ranges with smaller ρ causes CPU overhead bottlenecks in various parts of the system, e.g., in performing the RID-to-block lookup for each range and in column decompression start-up at each range. Our experiments show that CPU cost can become an issue if ρ is pushed significantly below ω_i. The practical bound for a "proper" granularity limit $\epsilon < \omega_i - \rho$ that we used is $\epsilon = 4$, as this is the point where CPU overhead exceeds I/O cost on a fast SSD system (Fig. 11, right).

Finding an efficient scan granularity We define an efficient BDCCscan access granularity, taking into account multiple columns C_i with varying data densities Δ_i, and give an algorithm to find it:

Definition 8 (*Efficient scan granularity*) Given a BDCCscan on a set of columns $C = \{C_1, \ldots, C_c\}$ of a round-robin interleaved BDCC organized table T with dimension uses $U = U_1, \ldots, U_k$ and a scan memory budget $MEM_{\max}$, the sequence of dimension specifications $\sigma = \langle S_1, \ldots, S_k \rangle$—with $\forall i : U(S_i) = U_i$—sets masks $M(S_i)$ such that it achieves the efficient scan granularity $\boldsymbol{\rho}_\sigma$, which is the minimum ρ_σ satisfying constraints:

(1) $MEM_{T,C,\sigma} \leq MEM_{\max}$: avoid I/O thrashing and
(2) $\forall i : \omega_i - \rho_\sigma < \epsilon$: avoid CPU thrashing.

Algorithm 4 (*BDCCscan granularity*) We initially set σ so that $\forall 1 \leq i \leq k : ones(M(S_i)) = ones(M(U(S_i)))$ (all bits are used). We then compute the resulting ρ_σ and check the constraints. Until this succeeds, set the single lowest 1-bit in the masks $M(S_i)$ to 0 and retry.

Given a fixed (small) memory budget $MEM_{\max}$ for scans, the ρ_σ of BDCCscan tends to be determined by constraint (1) to the ω_{big} setting of the biggest column(s), such that these do not need memory buffering ($\rho_\sigma < \omega_{big}$). Columns C_{big} with high data density Δ_{big} have a low ω_{big}, so we perform fine-grained access. The columns C_{small} with low data density Δ_{small} are accessed with $\rho_\sigma \leq \omega_{small}$, but since they fit in the memory budget, repetitive I/O is avoided. Constraint (2) protects against CPU thrashing on these small columns.

6.3 Query processing with sandwich operators

Being able to scan data from disk in any dimension order with an identifier that explicitly marks each group with the same characteristics, provides opportunities for partitioned processing. However, these approaches are designed to fit scenarios where data is grouped into tens, hundreds or maybe thousands of partitions. With BDCC we cluster data into up to millions of small groups and want to benefit from small groups to achieve better CPU cache locality. Also, the high number of groups would push BDCC over the edge when faced with typical partitioning problems like plan explosion.

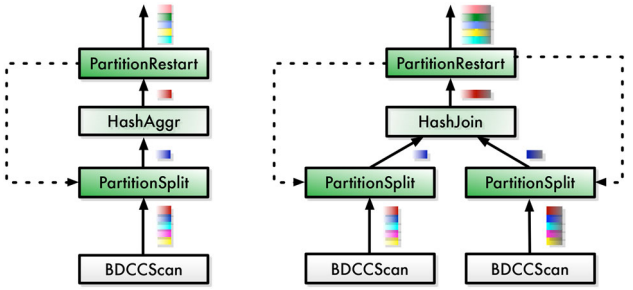

Fig. 12 Sandwich operators for Join and Aggr

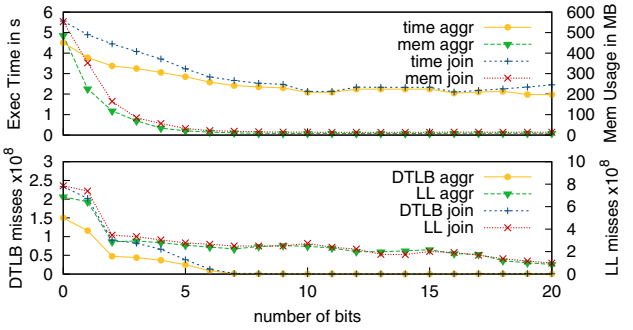

Fig. 13 Sandwiched `HashAggr` & `HashJoin`: Elapsed time, memory usage, DTLB and lowest level cache misses for counting the frequency of column `l_orderkey` and joining two tables `LINEITEM` and `ORDERS` using different number of groups

On top, it would be nice to avoid re-implementing a partitioned variant of each potential physical operator. Instead, we focus on reusing all existing `Aggregation/Grouping`, `HashJoin` and `Sort` operators.

Meeting the high demands of a million-group clustering and avoiding reimplementation of operators unites in what we named "Sandwich Operators". We devise a *split* and *restart* approach where the operator for partitioned execution is "sandwiched" in between, splitting the input stream at a group boundary and holding back the rest of the input. This way the operator is tricked into believing that end-of-group is end-of-stream, but after performing its epilogue action (e.g., `HashJoin` produced all tuples), the operator is restarted on the next group, reusing already allocated data structures.

For this purpose we added two new query operators `PartitionSplit` (*stream*, *_gid_*) and `PartitionRestart` (*stream*). Based on an iterator model for data processing, the `PartitionSplit` operator becomes the new root node of the to be partitioned operator's input stream(s) and the `PartitionRestart` operator receives the to be partitioned operator as input stream. The basic idea of these operators is illustrated in Fig. 12 for an unary operator, `HashAggr`, and a binary operator, `HashJoin`. `PartitionSplit` is used to detect the group boundaries in the input stream based on attribute values of *_gid_*, a changing value marks a new group, and stops producing data when reaching such a boundary. `PartitionRestart` controls the sandwiched operator's restart after this one finished producing tuples for a group, simply passes through the result tuples to the next operator and notifies its corresponding `PartitionSplit` operator(s), to start producing the next group. Note that this communication between `PartitionRestart` and `PartitionSplit` is a form of sideways information passing, for which `PartitionRestart` has to know the corresponding `PartitionSplit` operator(s). These are determined during query initialization, and typically are its grandchildren.

In order to show the effects, we created a set of micro-benchmarks. We created a co-clustering of `LINEITEM` and `ORDERS` of TPC-H at SF100. We used two dimensions, `D_DATE` and `D_CUSTOMER` with 10 bits each and clustered first on `D_DATE` and then `D_CUSTOMER`, providing a total of 1 million groups. This setup is comparable to our running example, only with adapted dimensions. This way, the access pattern generated by `BDCCscan` stays the same for all experiments, not falsifying the observations for the Sandwich Operators.

In order to get the different numbers of groups, we split groups from run to run by exploiting our `BDCCscan`, by adding the next bit in line to the dimension specifications, i.e. we started with a plain scan, then requested 1 bit for `D_DATE`, 2 bits, . . ., 10 bits while ignoring `D_CUSTOMER`. Then we also requested the first bit for `D_CUSTOMER`, the second bit, . . ., the 10th bit, so that the last scan produced 1 million groups in total. For the micro-benchmark of an unary operator (`Aggregation/Grouping`) we counted the frequencies of `l_orderkey` of `LINEITEM` and for the binary operator (`HashJoin`) we joined the two tables on the foreign key `orderkey`.

Figure 13 shows the important results of these experiments. Their behavior is nearly identical. The upper part of Fig. 13 shows that with more bits, i.e. more groups, memory consumption goes down while speed goes up. This is explained by the lower part: the hash table size decreases with higher number of bits, causing the number of TLB and lowest level cache (LL) misses to drop. At 128 (7 bits) groups cache misses reach a minimum, as the hash table then fits into cache (15 M distinct values; $15\,\mathrm{M}/128 \times 32\,\mathrm{B} \approx 3.6\,\mathrm{MB}$).

6.4 An example

In Fig. 14 we demonstrate BDCC in the query:

```
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
```

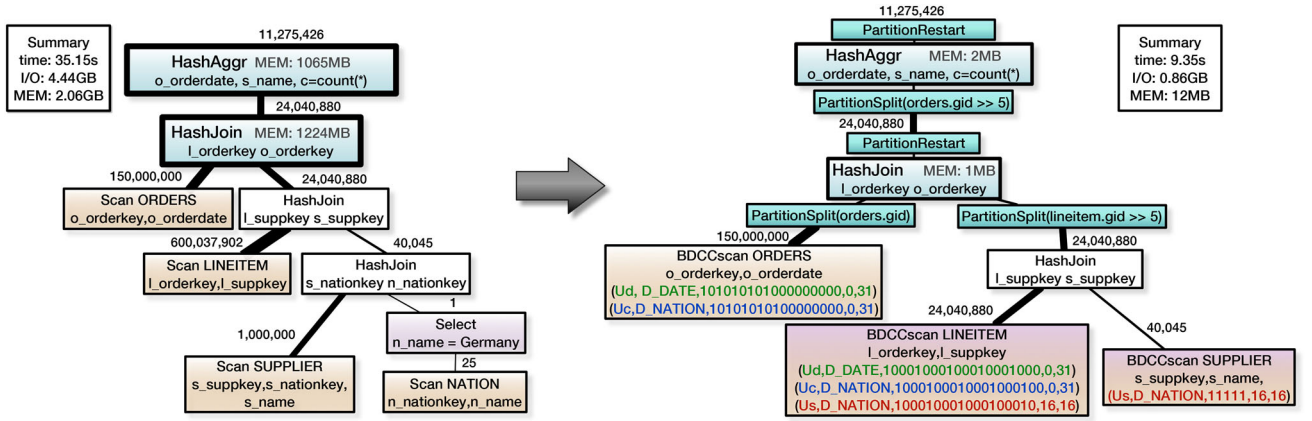

Fig. 14 Running example: normal query (*left*), exploiting BDCC (*right*)

The query uses a TPC-H schema SF100, where `ORDERS` is clustered on customer-nation and order-date, `SUPPLIER` on supplier-nation, `LINEITEM` on customer-`D_NATION` and order-`D_DATE`, but also on supplier-`D_NATION` and `D_PART`. Full details on the clustering are given in Sect. 9. BDCC provides the following optimizations:

- The `Select` operator filtering *Germany* can be dropped and is pushed to `BDCCscan` on `SUPPLIER` reducing I/O by a factor 25 as there are 25 nations. For dimension `D_NATION` only bucket 16 is selected.
- The `Select` operator filtering *Germany* can also be propagated to `BDCCscan` on `LINEITEM` also reducing I/O for by a factor 25. Again, for supplier-`D_NATION` (`Us`) only bucket 16 is selected.
- The `HashJoin` between `ORDERS` and `LINEITEM` uses Sandwich Operators, exploiting the two common dimensions (customer-nation and order-date), leading to a reduced memory consumption of 77 versus 1224 MB. Note `D_PART` is ignored for sandwiched execution by shifting `gid` five bits to the right. Also note that customer-`D_NATION` plays no role in SQL, yet BDCC exploits it for this join.
- The `Aggregation/Grouping` counting items per distinct <`o_orderdate`, `s_name`> is also accelerated by Sandwich Operators, exploiting the functional dependency of `o_orderdate` and `D_DATE`, leading to a reduced memory consumption of 186 versus 1065 MB.

In total the query is accelerated by 74 % (9 vs. 35 s) and memory consumption is reduced from 2 GB to 258 MB.

6.5 Query optimization

As `BDCCscan` can produce compatible dimension orders for co-clustered tables, the question arises, how this can be exploited during query execution. In Fig. 14 we shows that scans need to be replaced by `BDCCscans` and that Sandwich Operators need to be introduced. Overall BDCC adds three tasks to an optimizer:

(i) *Selection pushdown and propagation* of range selections on columns that functionally determine a dimension or correlate[2] with it (also across joins),

(ii) *Join elimination* If a dimension table is only accessed for a selection predicate that is pushed down in *(i)* with equivalent results, the join can be removed—see the Germany selection in Fig. 14.

(iii) *Operator sandwiching* `Sort`, `Aggr` and all (equi-, semi- and anti-) variants of `HashJoin` are sandwiched, if permitted by the clustering. The main challenge is to determine the group orders delivered by `BDCCscans`.

Selection pushdown `BDCCscans` can push down equi- or range selections on the dimension keys by translating the selection predicate into bin numbers. This translation is a simple lookup in the sequence $S(D)$ that maps dimension values to bin numbers. Once the bin numbers are found, the dimension specifications as defined in Sect. 6.1 can be used to restrict the scan. For Fig. 8, consider for example a selection predicate where tuples are restricted to years 1999 and later. This predicate maps to bin numbers 2–3 and is transformed to the dimension specification $\langle U_d, 11, 2, 3\rangle$ to restrict a `BDCCscan` on `ORDERS`. In case of a foreign key join to `LINEITEM`, we can propagate this selection to the `BDCCscan` on `LINEITEM` by $\langle U_d, 0011, 2, 3\rangle$ (comp. σ_4 in Fig. 9). In order to realize selection propagation, a foreign key join analysis of the query needs to be performed and in case of different bit masks, the dimension specifications need to be translated. Using less bits, simply requires to shift *low* and *high* values to the right by the bit difference, using more bits requires to left shift *low* while

[2] Vectorwise exploits MinMax indices [21] to determine that, e.g., a `n_name` restriction determines a `n_nationkey`.

minmax ORDERS

rid	$shipdate_{min}$	$shipdate_{max}$	vol_{min}	vol_{max}	d_{min}	d_{max}	c_{min}	c_{max}
0	1997	1997	1	6	00	00	00	01
125	1997	1997	5	10	00	00	10	11
250	1998	1998	2	4	01	01	00	01
375	1998	1998	6	9	01	01	10	11
500	1999	1999	1	5	10	10	00	01
625	1999	1999	4	11	10	10	10	11
750	2000	2000	2	5	11	11	00	01
875	2000	2000	7	9	11	11	10	11

Fig. 15 Example of a possible MinMax index for ORDERS from Fig. 3 with major–minor bit interleaving

$high_{new} = ((high_{old} + 1) \ll bit\_diff) - 1$. Generally speaking, if there is any metadata that proves that a certain attribute combination *functionally determines* a dimension key, then equi-predicates on that attribute combination may also be pushed down. For this to be exploited, the optimizer needs a way to quickly look up dimension keys based on such an attribute combination, which can, e.g., be realized if there is a B+-tree with that attribute combination as index key.

In order to push down selection predicates to scans, many systems use some form of lightweight index. In case of Vectorwise, this lightweight index is called MinMax index [21] and is a small (a few thousand entries) summary structures that divides a table into a small number of sections, and for each section keeps the starting row-id (RID) and the minimum and maximum value for each column (Netezza [3] has a similar concept known as "zone maps", Infobright [2] calls this "Knowledge Grid"). Figure 15 shows an partial example of a MinMax index for ORDERS with 8 entries for two columns x and y. The first row for example states that column y only has values between 1 and 6 in the first 124 rows of ORDERS. In the case of small (dimension) tables which have just fewer tuples than the size of a MinMax index, each tuple refers to a section, and the MinMax analysis performed in the query rewrite phase can, e.g., conclude that an equi-predicate on one attribute is also an equi-predicate on the dimension key in case of a functional dependency. Even without functional dependency information, and even in the case of just *correlated* behavior of attributes rather than full functional dependency, the MinMax index can be used to translate equi- or range-predicates on one attribute to range-predicates on the dimension key. This information can be used to enable BDCC selection push down. The benefit in this case is not so much the selection pushdown to the BDCC table the correlation is found in—as typically this is done by restricting the scan via the MinMax mechanisms itself—but as soon as we identify a range restriction on a dimension key this way, we can propagate this restriction to all foreign key connected BDCC tables in the query that use the same dimension, possibly restricting other fact tables.

In order to translate selection predicates on arbitrary attributes to dimension selections, it is necessary to maintain the MinMax index on decomposed parts of the *_bdcc_* key. This is illustrated in Fig. 15 for the D_DATE and D_CUSTOMER dimensions. This way a functional dependency of attribute *shipdate* and D_DATE can be detected and for example the predicate *shipdate* = 1999 can be translated to D_DATE bin 0. Also correlations as found between attribute *vol*, the order volume, and D_CUSTOMER can be found and in many cases, e.g., $vol = 6$ or $vol > 7$ can be translated in a range for D_CUSTOMER, i.e. range (2, 3). Note that in the example the MinMax index is *_bdcc_* aligned, which enhances the lookup results. However, this alignment is not strictly necessary. In the absence of a structure like MinMax, the count table itself can be extended to hold such minimum and maximum information per *_bdcc_* block. However, for large count tables, the lookup overhead becomes noticeable and a summarized count table with MinMax information is the better choice.

Typically, the granularity of the MinMax index is lower than the granularity of BDCC clustering, as this is an index that is kept per column and not per table. In case of major–minor bit interleaving of the dimensions, this mechanism can only exploit correlations on the major columns. Thanks to round-robin interleaving, correlation detection works for any dimension; yielding a reduction factor of $\text{MIN}(|\text{MinMax}|/d, 2^{ones(M)})$, with d dimension uses and mask M, of the original I/O volume for perfectly correlated equi-selections that produce a small result set. In major–minor clustering, the gain for the major dimension is of course greater, i.e. $\text{MIN}(|\text{MinMax}|, 2^{ones(M)})$, but for the minor dimensions, there is no gain at all. Concluding, round-robin interleaving adds to *robust* performance without the need for DBA-tuning, both, in terms of I/O performance independent of requested dimension order, and the ability to push down correlated selection predicates.

Join elimination In order to drop a join from a BDCC table to a table holding a dimension key, the dimension use of the BDCC table must use all bits of the dimension and no additional column of the dimension table can be present above the join. Also, the equi-selection predicate must be expressed over unique bins or the range-predicate must match exact bin boundaries of the dimension. Multiple joins along a dimension path can be dropped, if no attribute of the join path tables is used above the join to the BDCC table and no other tuple selections are performed along the path.

Operator sandwiching We now describe how Sandwich Operators can be integrated by bottom-up query optimizers working with "interesting orders" [31], assuming that join elimination was already performed.

Algorithm 9 (*Interesting dimension uses*) We collect the interesting dimension uses $IDU(T)$ of each table T on which each operator O depends. Each $idu \in IDU(T)$ is a subset of the dimension uses of T, holding those U_i functionally determined by the (i) any group by keys if $O =$ Aggr, (ii) major sort keys if $O =$ Sort, or (iii) join keys if $O =$ Join.

In case (iii), there must be matching dimension uses in both tables T_l and T_r (where the dimension path of one is a suffix to the other). These are added to $IDU(T_l)$ resp. $IDU(T_r)$.

This algorithm applied to the query plan from our example in Fig. 14 yields

$$IDU(\texttt{SUPPLIER}) = \{\{U_s\}\},$$
$$IDU(\texttt{LINEITEM}) = \{\{U_s\}, \{U_c, U_d\}\},$$
$$IDU(\texttt{ORDERS}) = \{\{U_c, U_d\}, \{U_d\}\}.$$

$\{U_d\}$ follows from the aggregate, the others from the joins, as LINEITEM and ORDERS in our TPC-H setup are co-clustered on order-date (U_d) and customer-nation (U_c), while LINEITEM and SUPPLIER are co-clustered on supplier-nation (U_s).

Algorithm 10 (*IDUpropagation*) We traverse the query DAG formed by FK-joins (as edges) bottom-up, pointing from referring to referred table (as nodes). In each table T, we intersect all $idu_i \in IDU(T)$ with the *idu* created for each incoming FK-edge. If this intersection is non-empty, it is added to *IDU* of the referring table. Then a similar top-down traversal is performed, adding non-empty intersections to the referred table.

This step ensures that *IDU* annotations reflect all relevant order opportunities in the query plan. The example DAG is ORDERS $\xleftarrow{\texttt{FK\_L\_O}}$ LINEITEM $\xrightarrow{\texttt{FK\_L\_S}}$ SUPPLIER. When checking $\{U_d\} \in IDU(\texttt{ORDERS})$ during the bottom-up traversal, we add $\{U_d\}$ to $IDU(\texttt{LINEITEM})$, as this is its intersection with the *idu* from $\texttt{FK\_L\_O}(\{U_c, U_d\})$.

Algorithm 11 (*Interesting dimension orders*) For each table T, the set $IDO(T)$ is generated by creating all permutations of each $idu_i \in IDU(T)$ as lists that contain dimension uses in some order. Lists are denoted $\langle \ldots \rangle$.

The interesting dimension orders algorithm applied to our example yields

$$IDO(\texttt{ORDERS}) = \{\langle U_d\rangle, \langle U_c, U_d\rangle, \langle U_d, U_c\rangle\},$$
$$IDO(\texttt{LINEITEM}) = \{\langle U_s\rangle, \langle U_d\rangle, \langle U_c, U_d\rangle, \langle U_d, U_c\rangle\},$$
$$IDO(\texttt{SUPPLIER}) = \{\langle U_s\rangle\}.$$

Since the amount of interesting orders strongly affects the search space and memory requirements of optimization, we describe a strategy that only generates *maximal* orders. This strategy is still guaranteed to find the best query plans and is based on two observations:

(i) BDCCscan (T, C, σ) for any σ as generated by Algorithm 4 costs roughly the same as a normal scan. BDCCscan cost is hence a non-issue in query optimization, e.g., scanning LINEITEM with $\langle U_c, U_d\rangle$ costs the same as with $\langle U_d\rangle$.

(ii) The cost of sandwiched operators monotonically decreases with more bits [7], so using more bits is better.

Thus, we can compare the relative benefit that different $ido \in IDO(T)$ offer for an operator using the length of the prefix covered in them by each $idu \in IDU(T)$. So, the FK_L_O join profits less from $\langle U_d\rangle$ than from $\langle U_d, U_c\rangle$ since its $idu = \{U_c, U_d\}$ covers a prefix of length one in the former and two in the latter. Hence, we can prune $\langle U_d\rangle$ and $\langle U_c, U_d\rangle$ from both $IDO(\texttt{LINEITEM})$ and $IDO(\texttt{ORDERS})$, because $\langle U_d, U_c\rangle$ is *superior* to them. For the query that means that some operators (i.e. the FK_L_O join resp. the aggregation) profit more from it, and no operator profits less.

Definition 9 (*Order superiority*) An interesting dimension order $ido_i \in IDO(T)$ is superior to another $ido_j \in IDO(T)$ if it benefits all $idu_x \in IDU(T)$ to an equal or higher degree. The degree to which ido_i benefits an idu_x is the length of the prefix of ido_i covered by idu_x (which may be 0).

Note that the benefit is defined in terms of members $idu \in IDU(T)$ rather than per operator generated *idu*-s, because Algorithm 10 adds extra *idu*-s for operators that indirectly (over FK-joins) profit from an ordering.

Order superiority straightforwardly leads to a pruning algorithm that keeps only maximal orders:

Algorithm 12 (*Order pruning*) For each $ido_i \in IDO(T)$, we iteratively check whether there exists another $ido_j \in IDO(T)$ that is superior to it, and if so prune ido_i. This is repeated until no more ido_i can be pruned. The result is called $MDO(T)$, i.e. maximal dimension orders.

As a result of the order pruning, our example has the following maximal dimension orders:

$$MDO(\texttt{LINEITEM}) = \{\langle U_s\rangle, \langle U_d, U_c\rangle\},$$
$$MDO(\texttt{ORDERS}) = \{\langle U_d, U_c\rangle\},$$
$$MDO(\texttt{SUPPLIER}) = \{\langle U_s\rangle\}.$$

Note that the addition of $\{U_d\}$ to *IDU* (LINEITEM) by Algorithm 10 is what made $\langle U_d, U_c\rangle$ superior to $\langle U_c, U_d\rangle$ in $IDO(\texttt{LINEITEM})$.

The optimizer thus has to choose between sandwiching the join of LINEITEM with SUPPLIER (using $\langle U_s\rangle$) versus the join of LINEITEM with ORDERS (using $\langle U_d, U_c\rangle$), which also benefits the aggregation. The cost model should determine the best strategy.

In a bottom-up BDCC-aware query optimizer, the set of interesting orders for each sub-plan are those result orders that follow the order of some BDCCscan in the sub-plan, where that BDCCscan on T produces an order from

$MDO(T)$. An order is propagated through the query plan according to [34]. Further, this order only remains interesting if some operator not yet part of the sub-plan benefits from it. For each interesting order, the best scoring plan is kept, in addition to the best plan without order (if faster).

For scans, the optimizer instantiates for each $ido \in MDO(T)$ a `BDCCscan` plan that produces *_gid_* in that order. The granularity for each `BDCCscan` is initially set by Algorithm 4, which results in some $\sigma = \langle S_1, \ldots, S_k \rangle$. For dimension uses U_i involved in joins, we reduce the bit number $ones(M(S_i))$ to the minimum of the inputs.

BDCC adds very little need to extend an existing cost model. Costs of the newly introduced operators `PartitionSplit` and `PartitionRestart` are essentially zero [7]. `BDCCscan` achieves performance comparable to a normal scan, so standard cost estimation can be used. However, selection pushdown should be taken into account when estimating the number of scan tuples. With BDCC statistics from the count tables and dimension specifications S_i, these estimations become very precise. The costs of a sandwiched operator are adjusted using the cumulative granularity $\gamma_{op} = \sum_{U_i} ones(M(S_i))$ of those dimension uses U_k exploited in it. `Sort` costs decrease by factor γ_{op} from $O(N \cdot \log(N))$ to $O(N \cdot \log(N/2^{\gamma_{op}}))$. For hash-based aggregation and join, one simply reduces the hash table size fed into any existing cost model (e.g., [24]) by factor $2^{\gamma_{op}}$.

We implemented BDCC optimization in the query rewriter of Vectorwise [21]. This rewriter is a post processing step after the main optimizer and does not change join order—this may affect BDCC plan quality.

7 Handling updates

Warehousing systems in very many cases have load cycles during which data are regularly appended, and regularly older data are phased out. Increasingly they have to support a less voluminous, but continuous, trickle of more arbitrary updates (insert, delete, modify). For BDCC we have to differentiate between updating BDCC clustered tables and updating dimensions.

Updating BDCC tables Updating ordered tables is normally not possible without data re-organization. BDCC can be combined with physical table partitioning, as one can bulk-load by simply appending new BDCC clustered data, both, to table T and T_{CNT} —even if updates are scattered across the whole table according to its clustered organization. So a batch of inserts simply results in a new (logical) partition in T_{BDCC} and T_{CNT} by creating autonomous T^U_{BDCC} and T^U_{CNT} for the update batch and appending these as logical partition the original T_{BDCC} and T_{CNT}. Note that each partition on the inside is ordered according to the *_bdcc_* key and partitions

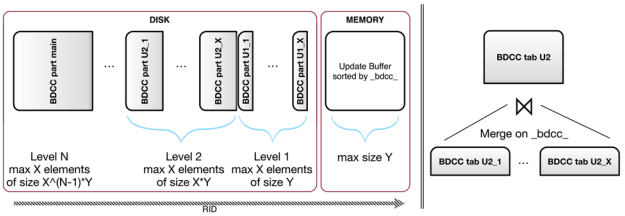

Fig. 16 Illustration of update behavior of BDCC tables. *Left side* logical structure of BDCC table under multi level update partitions. *Right side* merge step to create next level

are totally independent of each other. This way, the mechanisms to access a batch updated BDCC table are the same as described in Sect. 6.1 are not affected by update partitions. In order to avoid scanning the *_bdcc_* column of a table when searching for a partition, e.g., for merging or phasing out data, a small summary structure of start and end row IDs is used. This structure can also hold additional information such as for example the append date.

Logical partitioning is useful in data life cycle management, where new data are regularly appended (e.g., each day) and old data gets phased out, because a warehouse keeps, e.g., only the last 3 months. Appending new data to a new partition is fast but may lead to many small partitions; with too small access granularity as the BDCC clustering depth is based on the number of tuples in the initial data set. This leads to reading a disk block many times on order generate a scan order and should be avoided in order to keep `BDCCscan` a no extra cost operation. Therefore, multiple small partitions should be merged periodically to form larger ones and after enough larger partitions are created these again are merged into even larger partitions and so on; leading to a scheme where a table is stored as a set of partitions of exponentially increasing size; amortizing update cost to the logarithm of table size, similar to log-structured merge trees [27]. However, in order to avoid a large number of writes as explained in [5], it is critical to merge all partitions of one level in the tree into a single partition of the next level when the threshold for partitions for a level is reached. Figure 16 illustrates this. This way a 1TB BDCC table can be created from 16 MB update sets by only writing the data five times when using 16 update sets per level.

Note that BDCC clustering depth as chosen by Algorithm 2 is adapted as the amount of data grows, always guaranteeing optimal data access. The adaption process consists of a count of the bin frequencies based on the new clustering depth and an update of T_{CNT}. The ability to phase out batches efficiently limits the size of merged batches and thus the clustering depth.

Trickle or scattered updates, where only very small amounts of tuples are inserted, deleted, or modified, should in column stores best be handled using differential techniques, such as a Write–Store [33] or, in case of Vectorwise,

PDTs [17]. For inserts and deletes, not only the PDTs of the BDCC organized table T get modified, but the group counts in T_{CNT} need to be updated, also using e.g., PDTs. While a set of trickle inserts can be collected into a batch update, the trickle deletes can be integrated during the merge step of the update batches.

Updating BDCC dimensions The appearance of new values in BDCC tables through updates and inserts entails the need to maintain *BDCC dimensions* under updates. But the immediate need to reorganize existing BDCC organized tables should be avoided. However, update patterns that insert always into the same bin will cause domain skew ("puff pastry" [25]) induced loss of efficiency in the I/O access patterns. In general, removing skew in the face of hammer updates can only be achieved by periodically recreating the dimension and fully re-organizing all existing BDCC tables that use it, e.g., during a checkpointing operation typically found in systems with differential updates [17,33].

In any other case immediate reorganization of BDCC tables is avoided, however, depending on the kind of change and dimension, different actions are necessary.

For *inserts* no immediate action becomes necessary. Either the inserted dimension value falls into a non-unique bin or a unique bin becomes non-unique. In the second case, some query optimization techniques such as join elimination will not be available anymore for this particular bin. However, dealing with a forward growing domain (e.g., date-time related) we provide an additional mechanism. Appending new bins to the dimension will potentially cause the number of bits of the dimension to grow. When a dimension D expands by one bit, it is sufficient to only modify metadata of all dimension uses U_i of all BDCC organized tables that use D ($D(U_i) = D$), by adding a '1' at position $m + 1$ (at the front) in its bit mask $M(U_i)$. In effect, we declare these BDCC tables to be already clustered on one more bit; as this bit is '0' for all existing groups, existing *_bdcc_* column values are *not affected* and *no table reorganization* is required.

For *deletes* it can be assumed that the matching tuples in the BDCC tables are already deleted, so differential techniques are sufficient.

For *updates* we have to observe different cases to explain all opportunities of BDCC dimensions under updates. While an update of a dimension value that does not move this value from one bin to another naturally requires no action, a bin change results in varying actions depending on the kind of bin. For a *non-unique* bin, this is best treated as delete and update operation. However, in this case all tuples in the BDCC organized tables have not previously been deleted and, thus, also need to be deleted and re-inserted according to standard BDCC table update mechanisms. For *unique* bins, only actions on the metadata are required and BDCC organized tables are left untouched. The bin numbers need to be re-assigned using differential techniques and this way the *_bdcc_* key in T_{CNT} needs to be updated before the `Project()` step when generating the *_gid_* and RID ranges that are fed into `FetchScan`. Note that tuples in the T_{CNT} must not be reordered in order to not lose the offset mapping to the T_{BDCC}. If for generating the scan ranges in `BDCCscan` the structures are used, the right *_gid_* order is produced by the scan. However, if a BDCC table is further updated with bulk appends, the original dimension excluding these updates needs to be used. This way tuples in the new partitions are stored in the same order as in previous partitions and the mapping provided by the differential structure is sufficient for query processing. Note that in step (i) in the `BDCCscan` implementation nothing changes, as for adjacent rows the right shifted *_bdcc_* value may not be equal anymore—thus, `count` values for these rows are not aggregated and result in more scan ranges.

8 Discussion

Transferring to row stores Working with Vectorwise, our work is based on a column store, but we believe the general framework is also transferrable to row stores. For BDCC creation, a row can simply be viewed as an extra wide column. This, of course, requires more bits for clustering in order to meet the I/O parameter requirements for efficient scatter scans. Update mechanisms as described for batch updates also work in a row store, however, row stores typically favor in place updates that are not supported by BDCC. All query optimization techniques like selection pushdown or join elimination can also be transferred; however, summary data structures as minmax indexes may not be present, and thus, correlation detection would not be available. In case, such summary structures exist, a row store would need to provide separate summaries for the decomposed *_bdcc_* column in order to keep this optimization fast. Query execution techniques like Sandwich Operators and `BDCCscan` are also easily transferrable; however, the adaptive scan becomes unnecessary in a row store. With the absence of the adaptive scan and correlation detection, being two main reasons for round-robin interleaving, other bit interleaving patterns may become more interesting again.

Future work aims at investigating parallelization of sandwich processing. First implementations have shown a high potential of scaling BDCC execution across a large number of cores. Also, additional clustering of *_bdcc_* groups will be investigated to further exploit modern CPU technology. Providing a robust BDCC schema as a starting point also opens the question whether workload driven approaches can fine tune a setup.

9 Evaluation

To show the effectiveness of BDCC, we use the SF100 and SF1000 TPC-H and SF100 SSB benchmarks:

PLAIN This scheme simply holds the data without any key or index definitions.

PK As a baseline for indexing, we store data clustered on the primary keys.

OP This is an optimized Vectorwise indexing. Vectorwise does not support C-Store style [33] ordered join projections, but approximates this by storing tables ordered on an extended key, adjoining dimension keys reachable over a foreign key to the table.

BDCC This scheme is created with our design algorithms with fully automated query processing.

ADC As a multi-dimensional clustering version, we provide ADC [13]. To keep comparisons fair we use similar dimensions in ADC and BDCC. For ADC we re-wrote all queries by hand exploiting selections.

All schemes use automatic compression and are implemented end-to-end in the same system (Vectorwise), so the comparison is apples-to-apples. We also provide a micro-benchmark to compare BDCC with partitioning based on query Q03 of TPC-H as an example and provide insight on update performance for TPC-H.

System setup We evaluated on an Intel Xeon E5560 with 32 GB RAM and 32 KB L1, 256 KB L2 and 8 MB L3 cache. The OS is a 64 bit Debian, kernel 2.6.32. Databases were stored on a RAID0 of 4 Intel X25M SSDs with stripe 128 KB and max. bandwidth of 1 GB/s. As BDCC does not yet support parallelization, queries are only executed on a single core. We used 4 GB buffer space and 28 GB query memory. The page size was 32 KB.

TPC-H benchmark The PK setup turns `LINEITEM-ORDERS` and `PARTSUPP-PART` joins into merge joins. However, the important dimensions (e.g., date for the largest table `LINEITEM`) cannot be recognized by this scheme.

OP can recognize these dimensions. We add the `orderdate` column to `LINEITEM`, and store it with a `CLUSTERED INDEX on orderdate,orderkey, linenumber`, with the latter two being the original primary key. In `SUPPLIER` and `CUSTOMER` the sort key is (`r_regionkey`, `n_nationkey`) plus primary key. Here, columns are adjoined to other tables for the purpose of ordering, which enables this scheme to recognize important dimensions.

For ADC and BDCC, we recognize four dimensions: date, customer, supplier and part. In ADC we extract year and month from `o_orderdate` and adjoin it to `ORDERS` and `LINEITEM`. In addition we adjoin `r_name` and `n_name` to `CUSTOMER`, `SUPPLIER`, `ORDERS` and `PARTSUPP` and twice, once for customers and once for suppliers to `LINEITEM`. And finally we adjoin `p_brand` and `p_size` to `PARTSUPP` and `LINEITEM` and keep all tables ordered on these attributes following the order of presentation.

For BDCC we used Algorithm 3 to semi-automatically design the physical schema given as input DDL the usual foreign keys for TPC-H, plus `CREATE INDEX` DDL on `o_orderdate`, `(n_regionkey, n_nationkey)` and `p_partkey`. The latter compound key allows the query rewriter to detect that a region equi-selection determines a consecutive `D_NATION` bin range.

Using this DDL, Algorithm 3, which treats `CREATE INDEX` as hints for BDCC, creates the dimensions:

BDCC dimension (D)	$bits(D)$	Table $T(D)$	Key $K(D)$
`D_NATION`	5	`NATION`	`n_regionkey, n_nationkey`
`D_PART`	13	`PART`	`p_partkey`
`D_DATE`	13	`ORDERS`	`o_orderdate`

We also declare indices on the foreign key references `l_orderkey`, `o_custkey`, `c_nationkey`, `l_suppkey`, `l_partkey`, `ps_partkey`, `ps_suppkey` and `s_nationkey`.

Algorithm 3 clusters `NATION` on `D_NATION` and `PART` on `D_PART`. Dimension uses also get included in the referencing tables, over the foreign keys with a declared index (treated as a hint). Thus, `SUPPLIER` and `CUSTOMER` are clustered on `D_NATION`, and `ORDERS` is clustered on `D_NATION` (via `CUSTOMER`), as well as on `D_DATE`. `PARTSUPP` gets clustered on `D_PART`, and on `D_NATION` via `SUPPLIER`. `LINEITEM` gets clustered on all dimensions. In fact, as in the TPC-H schema graph two *different* join paths exist between `LINEITEM` and `NATION`, it gets clustered twice on `D_NATION`: both for customer and supplier nations. This yields the dimension uses:

BDCC Table	$D(U_i)$	$P(U_i)$	$M(U_i)$
NATION	D_NATION	–	11111
SUPPLIER	D_NATION	FK_S_N	11111
CUSTOMER	D_NATION	FK_C_N	11111
PART	D_PART	–	1111111111111
PARTSUPP	D_PART	FK_PS_P	101010101011111111
	D_NATION	FK_PS_S. FK_S_N	10101010100000000
ORDERS	D_DATE	–	101010101011111111
	D_NATION	FK_O_C. FK_C_N	10101010100000000
LINEITEM	D_DATE	FK_L_O	1000100010001001000
	D_NATION	FK_L_O. FK_O_C. FK_C_N	1000100010001000100
	D_NATION	FK_L_S. FK_S_N	100010001000100010
	D_PART	FK_L_P	10001000100010001

Given that the highest density column l_comment has 550,000 pages (i.e. $A_B = 32$ KB) for SF 100, Algorithm 2 clustered LINEITEM at granularity $\lceil \log_2 550000 \rceil = 20$ bits. For SF1000 we dropped l_comment for space reasons and since it is never used, now, l_orderkey dictates the number of bits $\lceil \log_2 971819 \rceil = 20$.

In comparison to PK and OP, ADC and BDCC lose the efficient merge joins between LINEITEM and ORDERS and PARTSUPP and PART respectively, but gain selection pushdown on all dimensions across the whole schema. BDCC, however, provides sandwiched execution.

Storage Storage for the various systems is as follows:

in GB	SF	PLAIN	PK	OP	ADC	BDCC
Total	100	54.59	59.79	59.23	67.58	59.12
Index	100	0	1.27	1.27	7.06	0.06
Total	1000	280.04	339.29	336.35	415.52	303.57
Index	1000	0	12.69	12.69	46.14	0.58

While PLAIN has no overhead, OP and PK create a join index to support the merge joins. ADC, even with compression, requires significant storage overhead for the adjoined columns, where BDCC in combination with RLE compression only requires minimal overhead. However, as comparison to PLAIN shows, a table's ordering also has influence on compression rates that can be more significant than indexing overhead. For SF1000 the dropped l_comment is the reason for reduced space.

Execution time Figure 17 (top) shows cold execution times for all 22 SF100 TPC-H queries. The dark-shaded parts show time spent in scans. The PK setup gains 146 s compared to PLAIN, mostly via the LINEITEM-ORDERS merge join. ADC profiting from selection pushdown, but loosing the merge joins, has a similar total execution time compared to PK (303 s); however, the power score is 55 % higher. OP outperforms both as it covers the important dimensions and provides merge joins. BDCC however is always close, often better because co-clustering across multiple dimensions permits to push down these *and more* selections to *all* tables affected. Vectorwise's MinMax indices allow both schemes to push down selections correlated with accelerated dimensions (e.g., shipdate selection push down, as tables have orderdate locality). However, BDCC accelerates more joins—this way compensating for loosing the merge join between LINEITEM and ORDERS. In the end BDCC clearly has the upper hand: its inverse geometric mean—the Power score[3]—is 24 % higher (90877 vs. 69145) and elapsed time is 16 s less.

For SF1000 as shown in Fig. 18 (top), the picture is similar. In general all setups perform not quite as well as for SF100 according to the power score. While BDCC performance drops by 18 %, PK looses 26 %, OP 28 %, PLAIN 32 % and ADC 39 %. Some of the additional performance loss for the non BDCC systems can be explained because of spilling joins and aggregations, but most of the other queries still loose proportionally more time when compared to BDCC. Only for queries that exploit merge joins and ordered aggregation PK and OP scale better than BDCC.

Memory consumption Figure 17 (bottom) shows that for SF100 on average BDCC needs much less memory than PLAIN (0.07 vs. 1.10 GB), and peak memory usage drops from 6.76 GB to 283 MB. ADC has comparable memory consumption to PLAIN due to no optimization for joins. Compared to PK and OP, BDCC is a factor 5 (peak 13×) more efficient, even with the "big" join gone, as BDCC reduces memory for *all* significant joins due to its co-clustering approach across the whole schema. We configured query memory such that hash-operators would never spill for SF100; Fig. 19 shows execution times for the three queries with the highest memory consumption (Q10, Q13, Q14) for PK, OP and BDCC for varying query memory. While BDCC execution times stay constant (compare to 4 GB bars), PK and OP times shoot up with lower memory limits, as operators start to spill. At 1 GB query memory, OP is an additional 82 s slower than BDCC and PK even 235 s.

For SF1000 as shown in Fig. 18 (bottom), again, the picture is similar. However, *all* systems but BDCC spill joins or aggregations to disk, even with query memory at 28 GB, which results in sublinear scaling for memory consumption but significantly lower execution times. At 10× larger SF and the same number of bits, the adaptive scan does not trigger as often. This way, e.g., Q14 and Q15 preserve more memory compared to SF100 as Sandwich operators benefit from the extra bits.

[3] These results are reported for academic interest only, are not available in any product, have not been audited and are not official TPC-H scores.

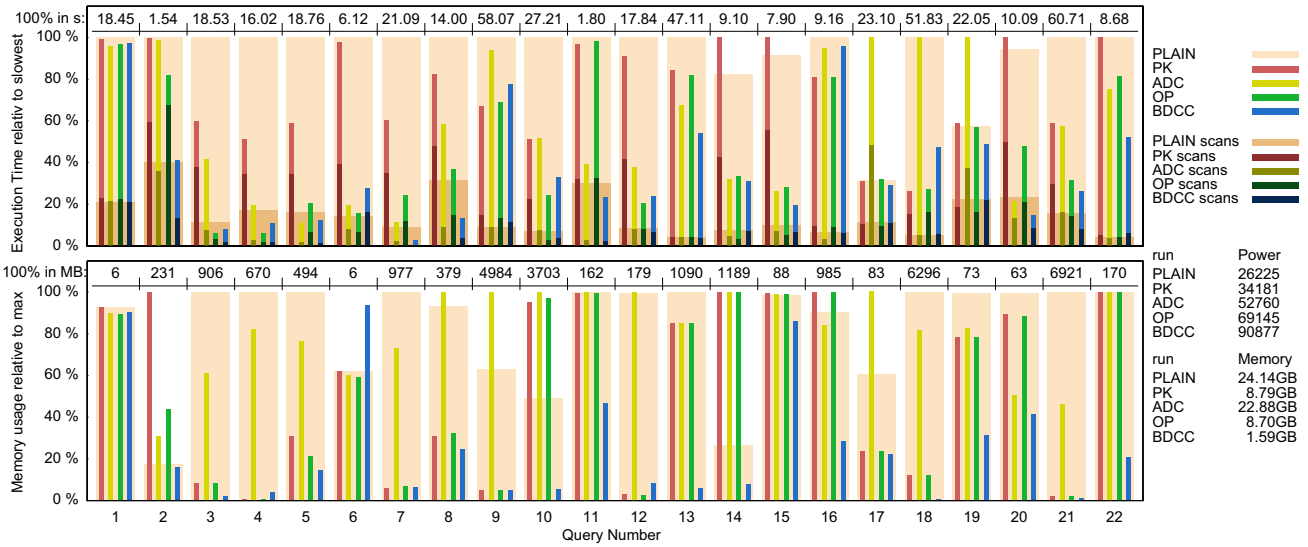

Fig. 17 TPC-H SF100 execution and scan times (*top*) and memory consumption (*bottom*) for PLAIN, PK, ADC, OP, BDCC

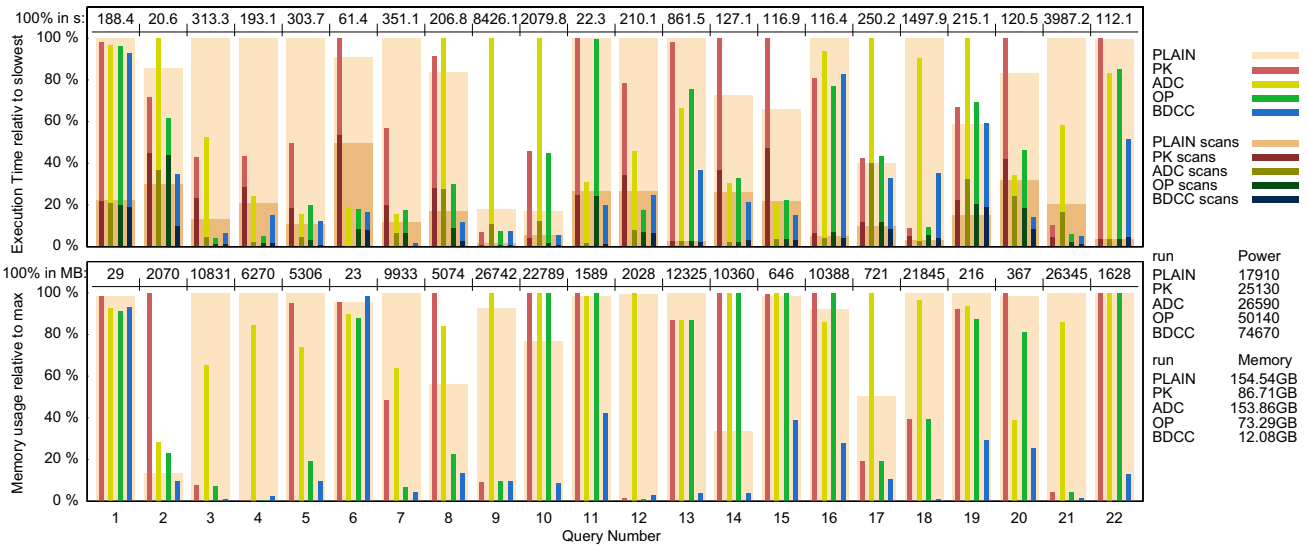

Fig. 18 TPC-H SF1000 execution and scan times (*top*) and memory consumption (*bottom*) for PLAIN, PK, ADC, OP, BDCC

Detailed analysis For PK and OP queries, Q2–Q6, Q7–Q12, Q16, Q18, Q20, Q21 benefit from merge joins instead of hash joins. In addition OP Queries Q3, Q4, Q5, Q8 and Q10 are accelerated by selection pushdown and Q3, Q6, Q7, Q12, Q14, Q15, Q20 and Q21 benefit from the correlation of `o_orderdate` and `l_shipdate`, allowing MinMax indices to identify pushdown ranges. These selections are also recognized by ADC and BDCC but in addition and due to their multi-dimensional nature, these schemes additionally push down nation and region selections in queries Q2, Q5, Q7–Q11, Q20 and Q21. For BDCC sandwiching is applied throughout the whole schema, supporting more joins than just `LINEITEM-ORDERS` and `PARTSUPP-PART`. In Q13, the `ORDERS-CUSTOMER` join is sandwiched for customer `D_NATION`, although `NATION` is not involved in the query, but the join key `c_custkey` implies the nation of a customer. Sandwiching largely reduces memory usage compared to the others, where a full materialization of the `CUSTOMER` columns is required. Same holds for Q10 and Q18. In Q14 the join `LINEITEM-PART` is reduced. In Q16 the count of the distinct `s_suppkeys` is sandwiched, shrinking the hash table by a factor of 25. Q18 performs a full aggregation of `LINEITEM` on `l_orderkey`; sandwiching helps with respect to PLAIN and ADC, but the streaming aggregate in PK and OP cannot be beaten.

Star schema benchmark The PK setup in contrast to PLAIN results in slightly different query plans and for-

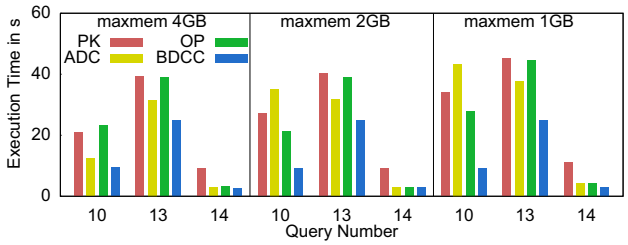

Fig. 19 TPC-H SF100 execution times for varying memory

eign key joins instead of standard M to N hash joins. OP orders `LINEORDER` after `l_orderdate` and, thus, can push date selections to `LINEORDER`. For ADC we adjoined `d_year`, `c_region`, `c_nation`, `s_region`, `s_nation` and `p_mfgr` and `p_category` to `LINEORDER` and sorted in the respective order. For BDCC we used Algorithm 3 to semi-automatically design the schema based on foreign keys and `CREATE INDEX` on `d_datekey`, (`s_region`, `s_nation`), (`c_region`, `c_nation`) and (`p_mfgr`, `p_category`), leading to four dimensions and a round robin clustered `LINEORDER` with 5 bits per dimension.

Storage Storage for the various systems is as follows:

in GB	PLAIN	PK	OP	ADC	BDCC
Total	15.37	16.05	15.79	27.77	21.20
Index	0	0	0	6.70	0.04

Here, PLAIN, OP and PK have no overhead, OP's index is a single page. ADC, again, has significant overhead, while BDCC only has minimal. However, compression differences show, leading to extra storage overhead for the multidimensional approaches.

Execution time Figure 20 shows the results of all 13 SSB queries. PLAIN and PK are comparable and show the lowest performance. OP has good performance where a query has date selections and otherwise performs as bad as PLAIN and PK. ADC and BDCC are both by far superior to the other setups, which is expected as in PLAIN and PK `LINEORDER` is always fully scanned and in OP only the date dimension is really recognized. When comparing ADC and BDCC it shows that BDCC is 28 % faster, a result from more precise selection propagation as minor dimensions cannot be recognized by Vectorwise's scan pre-selection. For a few queries, the Sandwich Operators add additional benefit. Looking at queries that are tailored to ADC, e.g., Q3.1 to Q3.4, we see that both approaches perform similar. However, with better recognition of deeper dimensions, it is to be expected that ADC performance is comparable to BDCC—but this cannot simply be achieved by adjoining dimensions columns and re-writing SQL queries.

Memory consumption All schemes need about the same amount of memory, on average 22 MB (OP) to 32M B (PLAIN), as all joins and aggregations use small hash tables and neither merge joins nor sandwiched execution provides a significant benefit.

Updates Figure 21 provides execution times for updates on TPC-H (RF1 and RF2). For a *single* insert set PLAIN and BDCC updates are handled as bulk appends, which are faster than PDTs for PK, OP and ADC. BDCC only sorts the insert batch (which makes it slightly slower than PLAIN) and then appends to the BDCC- and count-table. We also show amortized insert costs, as PDT memory consumption (110 MB for one batch) and maintaining the logarithmic update structure for BDCC come with overhead not shown in the single run. We set the PDT reorganization limit to 4 GB, triggering with the 38th batch, which leads to a full table re-organization. The amortized overhead for a run is 118 s, where for BDCC's logarithmic update structure, it is only 18 s. Deletes (del) are in all cases handled by PDTs, but as they require less memory (4.5 MB per batch), the amortized overhead is lower (5 s per run).

Executing all 22 TPC-H queries after loading 10 insert batches (RF1) results in a slight slowdown for all systems. In PK, OP and ADC the PDTs need to be processed, in BDCC the appended partitions. However, BDCC needs 30–90 % less extra time than the other setups, i.e. processing the

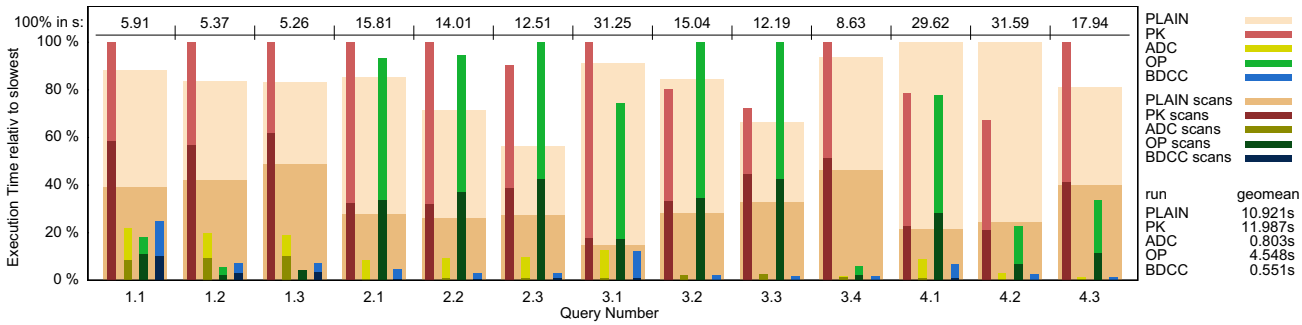

Fig. 20 SSB SF 100 execution and scan times

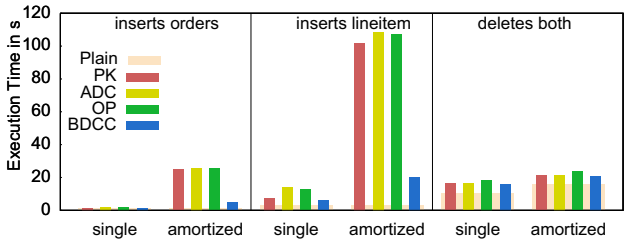

Fig. 21 TPC-H SF100 execution times for update sets

extra partitions is faster than processing the PDTs. BDCC needs 41 MB extra query memory, while PK and OP need an additional 1.08 GB memory for holding the PDTs (per query).

Adaptive scan granularity The best indicator of a correctly chosen scan granularity is the number of disk blocks transferred to memory. The 4 GB buffer pool in the SF100 TPC-H experiments does not require an adaptively chosen granularity. However, a limited 1 GB buffer pool, e.g., by concurrent queries, causes thrashing for large scans as in Q09: Requesting all 5 bits of both dimensions (`D_DATE`, `D_NATION`) from `LINEITEM` leads to 1 M requests with 540 K transfers for 270 K blocks, i.e. blocks are read twice. The adaptive `BDCCscan` reduces both dimensions to 4 bits per dimension, only transferring 300 K blocks, close to 270 K as with 4 GB buffer space. Similar behaviors show in Q18, Q19 for 512 MB buffer and Q06, Q08, Q12, Q21 for 256 MB.

Partitioning As partitioning is not yet supported by Vectorwise, we simulated it, partitioning `CUSTOMER` by the 25 nations and `ORDERS` and `LINEITEM` by 32 equally distributed `o_orderdate` partitions and the 25 nations from the customer dimension, a total of 800 partitions. We executed Q03 as an example, where only these tables are involved. With selection pushdown Q03 only requires two of the 32 `o_orderdate` partitions. No partitions can be pruned by nation. We generated two plans, (a) joining matching `ORDERS` and `LINEITEM` partitions first, before joining with the matching customer partition and (b) union each of the two `LINEITEM` and `ORDERS` partitions and joining to customer on a per nation partition basis. For OP, plan (a) executed in 4.9 s and plan (b) in 3 s, which is similar to the execution without sandwiched joins but with selection pushdown. Both are much faster than PK and PLAIN, but 100 % (a) and 25 % (b) slower than BDCC. Plus, memory consumption is much worse, BDCC needs only 16.3 MB, where (a) needs 937 MB and (b) needs 494 MB, mostly because of the exploding number of `HashJoins`. Similar plan explosion is expected for the other TPC-H queries.

10 Conclusions

We introduced bitwise dimension co-clustering, an elegant and powerful framework for multi-dimensional clustering for analytical workloads. We provided algorithms for database design, query optimization and query execution as well as updating the database. Experiments with TPC-H and SSB in the Vectorwise system show the high potential of the framework.

References

1. Born To Be Parallel. teradata.com
2. Infobright Enterprise Edition. infobright.com
3. Netezza Admin Guide. support.netezza.com
4. Agrawal, S., Narasayya, V.R., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD (2004)
5. Athanassoulis, M., Chen, S., Ailamaki, A., Gibbons, P.B., Stoica, R.: Masm: efficient online updates in data warehouses. In: SIGMOD, pp. 865–876. ACM (2011)
6. Barber, R. et al.: Blink: Not Your Father's Database! In: BIRTE (2011)
7. Baumann, S., Boncz, P., Sattler, K.U.: Query processing of pre-partitioned data using sandwich operators. In: Enabling Real-Time Business Intelligence, vol. 154 (2013)
8. Baumann, S., de Nijs, G., Strobel, M., Sattler, K.: Flashing databases: expectations and limitations. In: DaMoN (2010)
9. Bayer, R.: The universal b-tree for multidimensional indexing: general concepts. In: WWCA (1997)
10. Chan, C.Y., Ioannidis, Y.E.: Bitmap index design and evaluation. In: SIGMOD. ACM (1998)
11. Chaudhuri, S., Datar, M., Narasayya, V.: Index selection for databases: a hardness study and a principled heuristic solution. TKDE (2004)
12. Chen, W.J., Fisher, A., Lalla, A., McLauchlan, A., Agnew, D.: Database partitioning, table partitioning, and MDC for DB2 9. IBM Redbooks (2007)
13. Chen, X., O'Neil, P., O'Neil, E.: Adjoined dimension column clustering to improve data warehouse query performance. In: ICDE (2008)
14. Deshpande, A., Guestrin, C., Hong, W., Madden, S.: Exploiting correlated attributes in acquisitional query processing. In: ICDE '05
15. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**, 85–98 (1992)
16. Harizopoulos, S., Liang, V., Abadi, D., Madden, S.: Performance tradeoffs in read-optimized databases. In: VLDB (2006)
17. Héman, S., Zukowski, M., Nes, N.J., Sidirourgos, L., Boncz, P.A.: Positional update handling in column stores. In: SIGMOD (2010)
18. Herodotou, H., Borisov, N., Babu, S.: Query optimization techniques for partitioned tables. In: SIGMOD (2011)
19. Hu, T., Tucker, A.: Optimal computer search trees and variable-length alphabetical codes. SIAM J. Appl. Math. **21**(4), 514–532 (1971)
20. Huffman, D.: A method for the construction of minimum-redundancy codes. In: Proceedings of the IRE (1952)
21. Inkster, D., Boncz, P., Zukowski, M.: Integration of vectorwise with ingres. SIGMOD Rec. **40**(3), 45–53 (2011)
22. Leslie, H., Jain, R., Birdsall, D., Yaghmai, H.: Efficient search of multi-dimensional B-trees. In: VLDB (1995)

23. Li, Y., Patel, J.M.: Bitweaving: fast scans for main memory data processing. In: SIGMOD (2013)
24. Manegold, S., Boncz, P., Kersten, M.: Generic database cost models for hierarchical memory. In: VLDB (2002)
25. Markl, V.: MISTRAL: Processing relational queries using a multi-dimensional access technique. Institut für Informatik TU München (1999)
26. Morales, T.: Oracle database VLDB and partitioning guide, 11*g* Release 1 (11.1). Oracle (2007)
27. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree. Acta Inf. **33**(4), 351–385 (1996)
28. Orenstein, J.A., Merrett, T.H.: A class of data structures for associative searching. PODS '84
29. Padmanabhan, S., Bhattacharjee, B., Malkemus, T., Cranston, L., Huras, M.: Multi-dimensional clustering: a new data layout scheme in DB2. In: SIGMOD (2003)
30. Polyzotis, N.: Selectivity-based partitioning: a divide-and-union paradigm for effective query optimization. In: CIKM (2005)
31. Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T.: Access path selection in a relational database management system. In: SIGMOD (1976)
32. Sidirourgos, L., Kersten, M.: Column imprints: a secondary index structure. In: SIGMOD (2013)
33. Stonebraker, M., et al.: C-Store: a column-oriented DBMS. In: VLDB (2005)
34. Wang, X., Cherniack, M.: Avoiding sorting and grouping in processing queries. In: VLDB (2003)
35. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G.M., Storm, A.J., Garcia-Arellano, C., Fadden, S.: DB2 design advisor: integrated automatic physical database design. In: VLDB (2004)