Automatic Schema Design for Co-Clustered Tables

Stephan Baumann #, Peter Boncz *, Kai-Uwe Sattler #

#*Databases and Information Systems Group, Ilmenau University of Technology, Ilmenau, Germany*
first.last@tu-ilmenau.de

**Centrum Wiskunde & Informatica, Amsterdam, Netherlands*
boncz@cwi.nl

***Abstract*—Schema design of analytical workloads provides opportunities to index, cluster, partition and/or materialize. With these opportunities also the complexity of finding the right setup rises. In this paper we present an automatic schema design approach for a table co-clustering scheme called Bitwise Dimensional Co-Clustering, aimed at schemas with a moderate amount dimensions, but not limited to typical star and snowflake schemas. The goal is to design one primary schema and keep the knobs to turn to a minimum while providing a robust schema for a wide range of queries. In our approach a clustered schema is derived by trying to apply dimensions throughout the whole schema and co-cluster as many tables as possible according to at least one common dimension. Our approach is based on the assumption that initially foreign key relationships and a set of dimensions are defined based on classic DDL.**

I. Introduction

Physical schema design is an important task for database development, maintenance and tuning. Particularly, analytic workloads with complex queries can take benefit from indexes, clustering, partitioning and materialized views. However, even moderately sized schemas lead to a huge search space of possible configurations of these database objects. Typically, i.e. provided by most commercial systems, this problem is addressed by so-called design advisors which are able to analyze a given workload and to derive recommendations for an optimal configuration. Though, this works quite well for static scenarios with fixed workloads, dynamic environments (in terms of data characteristics and query workload) still pose a serious challenge. One possible solution is to monitor the workload and adapt the configuration continuously [1], [2]. However, changing a configuration at runtime, e.g. create new indexes, change clustering strategies or materialize query results can become quite expensive and affect the response times significantly.

An alternative approach is to derive a schema that is agnostic to the workload, i.e. not optimal for a specific set of queries but supports a wide range of queries of a given patterns. In this work, we present such an approach for analytic workloads on typical star or snowflake schemas. *Bitwise Dimensional Co-Clustering (BDCC)* is a scheme for co-clustering plain relational tables and sharing multiple dimensions at once. BDCC replaces the original relational table and accelerates typical star joins on foreign key relationships and selections.

In order to design a BDCC schema two main questions have to be answered: which tables should be replaced by a BDCC table and which dimensions are taken into account. For this purpose we could look

- at the original schema mainly for primary and foreign key constraints and perhaps some basic data characteristics (e.g., number of distinct values on dimensions)
- at the workload to identify frequently used joins and selection predicates.

The goal of this work is to capture this information by a simple set of *schema hints* which basically specify only dimensions and their relational representation. Based on this, we present a strategy to derive a BDCC schema without requiring workload information. The remainder of this paper is structured as follows. In Section II we briefly introduce our co-clustering scheme BDCC, in Section III we present algorithms for our automated approach to a physical design of BDCC, Section IV shows that our automated design widely accelerates the TPC-H workload, Section V discusses related work, followed by a conclusion in Section VI.

II. Introducing BDCC

With BDCC we co-cluster plain relational tables in order to share (at least partially) dimensional information between tables, when these are connected over foreign key relationships or use the same dimensions. In addition, we re-organize each relational table according to local and foreign key connected dimensions. As a result of this re-organization we can later accelerate joins between co-clustered tables and benefit from efficient access to these tables over dimension attributes. In BDCC, each relational table `T` is replaced by a new table $\texttt{T}_{\texttt{BDCC}}$, where inside the table clusters are formed by consecutive tuples with the same dimensional characteristics. These are represented by equal values of an artificial attribute $\texttt{T}_{\texttt{BDCC}}.\texttt{\_bdcc\_}$, that is later used during query processing to perform the above query optimizations.

The main benefits of a co-clustered table layout are:

(i) Selection pushdown for a dimension, or a correlated attribute to that dimension, that is used for clustering a table. Similar to most index structures.
(ii) Selection propagation between co-clustered tables, if the tables share a common dimension. Even propagation of selections on correlated attributes to the shared dimension.
(iii) Join acceleration based on the co-clustered layout that results in a pre-grouping on both join inputs. In [3] we explain how to exploit this pre-grouping using Sandwich Operators with the result of faster execution times and significantly reduced memory while processing the same amount of data.

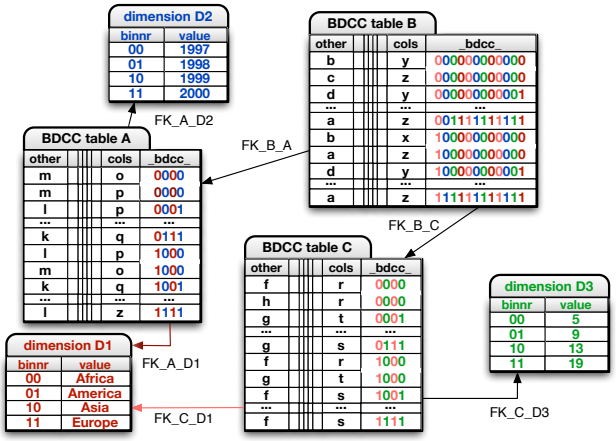

Fig. 1. Example of a co-clustered BDCC schema.

To co-cluster tables using BDCC, we need to (a) identify dimensions in a schema, respectively dimension keys and (b) specify how these dimensions are used for clustering. But first, we need a definition of a dimension.

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 tuple based on the values of the considered key K, and multiple tuples can be mapped onto a single bin, where smaller attribute values are mapped to smaller identifiers. Formally:

*Definition 1 (**BDCC Dimension**). A BDCC dimension* $\boldsymbol{D = \langle T, K, S \rangle}$ *is defined over dimension key* $K(D) = K = \langle attr_1, \ldots, attr_s \rangle, s \geq 1$, *of table* $T(D) = T$ *as a finite 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 of a bin number* n_i *and a bin (set) of values* V_i *such that* $\cup_{i=1}^{|S|} V_i = \{v | v \in T.K\}$. *Further:*

*(i) bin numbers are **ascending**:* $\forall 1 \leq i < j \leq |S| : n_i < n_j$.
(ii) bins of $S(D)$ ***never overlap**:* $\forall 1 \leq i < j \leq |S| : V_i \cap V_j = \emptyset$.
(iii) bins of $S(D)$ *are **ordered**:*
$\forall 1 \leq i < j \leq |S| : \; MAX(V_i) < MIN(V_j)$.
(iv) a bin $\langle n_i, V_i \rangle$ *is **unique** if* V_i *is a singleton* ($|V_i| = 1$).
(v) $\texttt{bin}_D(v) = n_i$ *is the **bin number** of value* $v \in V_i$.
(vi) $\texttt{bits}(D) = \lceil log_2(|S|) \rceil$ *is the **dimension granularity**, i.e., the number of bits needed to represent bin numbers.*
(vii) a dimension $D|_g$ *with reduced granularity* $g < \texttt{bits}(D)$ *bits is derived from dimension* D *if one chops off the* $\texttt{bits}(D)$*-*g *least significant bits of all bin numbers from* D *and unites all bins that now have the same number.*

Figure 1 shows three dimensions for a simplified BDCC schema. Note, that we only show the dimensions in this example and also use them as representatives for the relational tables behind them. Dimension D1 is a simple geographical dimension with four continents, dimension D2 is a time dimension with four years and dimension D3 is a dimension using range binning with four bins (2 bits), where the bin boundaries are represented by `value`.

Because of the co-clustered layout and shared dimensional information between co-clustered tables, dimension keys may not only be local attributes of `T`, but also attributes from other tables reachable over foreign keys along a *dimension path* P.

*Definition 2 (**Dimension Path**). A dimension path* $\boldsymbol{P}$ *is defined as a (possibly empty) chain of foreign key traversals* $P = FK\_T_1\_T_2.FK\_T_2\_T_3.\cdots.FK\_T_{n-1}\_T_n$, *from a context 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 the example in Figure 1 we also illustrate the foreign key relationships inside this schema by the notation FK_TableX_TableY. Table C, for example, is foreign key connected to dimension D1 via FK_C_D1. Table B also is foreign key connected to dimension D1, while this time not directly, but over table A resulting in the dimension path FK_B_A.FK_A_D1.

The explicit usage of a dimension in a table T is then defined via a *dimension use*, where a dimension path identifies the dimension with its keys and a bitmask controls how the dimension is used for clustering. The number and positions of set bits in this mask define the granularity and the priority on the combined ordering inside a table's clustering.

*Definition 3 (**Dimension Use**). To specify a clustering criterion for a 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 bitmask* $M(U) = M$, *that defines the number and the positions of the dimension bits in* T_{BDCC}.*_bdcc_. The number of bits used from dimension* D *for clustering is* $\texttt{ones}(M)$, $\texttt{ones}(M) \leq \texttt{bits}(D)$.

Lets take a look at the dimension uses for table C in Figure 1. This tables uses two dimensions D1 and D3. So $U_{D1} = \langle D1, FK\_C\_D1, 1010 \rangle$ and $U_{D3} = \langle D3, FK\_C\_D3, 0101 \rangle$. $M(U_{D1}) = 1010$ places the two bits of dimension D1 in the first and third position (red) of the ordering key `_bdcc_`.

A BDCC table definition is then straightforward:

*Definition 4 (**BDCC Table**). A BDCC table* $\boldsymbol{T_{BDCC} = \langle T, U_1, \ldots, U_d, b \rangle}$ *clustered on* b *bits is defined over a source table* T *by specifying* d *dimension uses* $U_1, \ldots, U_d$ *under the constraints*

*(i) **all b bits are set**:* $M(U_1) | \ldots | M(U_d) = 2^b - 1$
*(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_, where for each dimension use* U_i *we look up bin number* $n_i = \texttt{bin}_D(t.P.K)$, *with* $P = P(U_i)$, $D = D(U_i)$, $K = K(D)$ *and map the major* $ones(M(U_i))$ *bits of* n_i *to _bdcc_ according to mask* $M(U_i)$. T_{BDCC} *is stored sorted on attribute* T_{BDCC}*._bdcc_.
In addition a metadata table $T_{\texttt{COUNT}}$`(_bdcc_,count)` *is created, counting the frequency of each _bdcc_ value.*

The example in Figure 1 contains three fact tables that are BDCC clustered. We can see that table C is clustered using two dimensions D1 and D3 (light red and green) using a round robin bit interleaving pattern for the clustering key `_bdcc_`.

In a similar way, table A is clustered using dimensions D1 and D2 (dark red and blue). Because tables A and B are foreign key connected (FK_B_A) and table A is clustered by D1 and D2, we keep A and B co-clustered on these two dimension (both use dark red and blue bits in _bdcc_). As B is also foreign key connected to table C (FK_B_C), we also keep B and C co-clustered using again D1 (light red) and D3 (green). Here, we have to distinguish between the two dimension uses of D1 in table B as the foreign key paths are different, and thus, each use can logically be a different dimension. Note, that A and C, though not foreign key connected, are still co-clustered on dimension D1. This is useful in situations when we are looking for tuples in A and C from matching nations. A scenario very likely if both fact tables contain data related to for example the same customers.

Scanning BDCC tables To access a BDCC table we use a specialized scatters-scan, that is able to retrieve any major-minor order of the dimension that are interleaved in the _bdcc_ column. The offsets for the scatter-scan are calculated from $\mathtt{T}_{\mathtt{COUNT}}$. This scan adds an additional group identifier to the stream, that is used during query optimization, see [3]. In the example from Figure1 this means that for table A this scan can retrieve data in the orders (D1), (D2), (D1,D2), (D2,D1).

III. Automatic Schema Design

Designing a BDCC schema raises various questions. Primarily (a) which dimensions to choose, (b) how to create them, and (c) how to co-cluster a schema based on the chosen dimensions. For (b) we refer to [4], where we explain how to create balanced dimensions when faced with skew. In the following we outline answers on:

(i) how to self-tune a BDCC table for given dimensions?
(ii) how to find BDCC dimensions?
(iii) how to make a BDCC schema-design and parameter setting auto-tuned?
(iv) how to treat hierarchical and correlated dimensions?

Selecting data from any multi-dimensional structure on a *subset* of the dimensions typically leads to a scattered disk access pattern. Magnetic disk and – contrary to common belief – also flash-based storage performs sequential access more efficiently than random. Still, for any device, one can determine an efficient random access size A_R such that random reads approach the efficiency of sequential reads (e.g. such that throughput is 80% of sequential throughput). Multi-dimensional schemes, hence, must make sure that the access pattern they generate on average conforms to this efficient random I/O size. Currently, the efficient access size is roughly a few MB for magnetic disks, for Flash devices just 32KB [5].

Which dimension order to choose? Classical multi-dimensional approaches like MDAM [6] 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 kind of bit-interleaving, hence also major-minor ordering; For applications with clear major, minor dimensions this approach is fine, and is supported in BDCC by manual definitions.

However, major-minor ordering has as disadvantages that (i) dimension order is a knob that might get tuned wrongly and (ii) major dimensions get a much better access pattern than minor. Following the UB-Tree work [7], we prefer *round-robin bit interleaving* instead, storing tuples in *Z-order*. This eliminates the question of how to order a table's dimensions and distributes fast access among all dimensions.

Self-tuned BDCC table. From the definition of a BDCC table (Definition 4) it follows that for each table of the schema a number of dimension uses and the clustering depth need to be specified, which boils down to providing a dimension path to each dimension and an interleaving pattern of the bits from 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 $\mathtt{T}_{\mathtt{BDCC}}$. The idea is to bulk-load BDCC tables initially at a *maximal* granularity, but then to only create meta-data (the count-table) on a lower granularity; exploiting statistics gathered during bulk-load. This keeps the count-table small, such that BDCCscan can access it quickly.

*Algorithm 1 (**Self-Tuned BDCC Table**).* To BDCC cluster a table T we assume initial dimension uses $\{U_i, \ldots, U_k\}$.

(i) Initially set masks $M(U_i)$ 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 to 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 the full granularity of each dimension is used: the number of 1-bits B in all masks is maximal: $B = \sum_1^k \mathtt{bits}(D(U_i))$.
(ii) Compute the _bdcc_ column with this maximal granularity and store T sorted on _bdcc_, analyze the group sizes in a piggy-backed aggregation (together with the "correlated dimensions" analysis below).
(iii) Find the column that is largest on disk (highest density), choose the largest granularity $b \leq B$ such that the size in bytes of most groups is still above the efficient random access size A_R for this column.
(iv) Create $\mathtt{T}_{\mathtt{COUNT}}$ for this reduced granularity b, in a single ordered aggregation counting consecutive tuples with equal value _bdcc_ $\gg (B-b)$ (Since T sorted on _bdcc_ at granularity B is also clustered for $b \leq B$).

Note, that in (i) 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 the size of the join, detailed weights could also be calculated from a workload analysis and so on. Our goal here is, to keep it simple and robust for many workloads, so we advocate to look at the outgoing foreign keys.

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

*Algorithm 2 (**Semi-automatic Schema Design**).* An existing database is BDCC clustered in three phases. Interpreting `CREATE INDEX(`$I_1, \ldots, I_Z$`) on` T statements as BDCC hints and exploiting declared foreign keys:

(i) 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 it as a dimension use to T.

(ii) Create the dimensions one by one, using a fixed maximal granularity derived from the usage and the number of distinct values of a dimension (e.g. $\texttt{bits}(D) \leq 13$). Our algorithm in [4] takes into account the distribution of dimension values across *all* tables T_i where the dimension is used, creating a histogram on the union of all tables T_i joined over dimension path P_i, projecting only the dimension keys.

(iii) BDCC cluster all tables at a self-tuned granularity using Algorithm 1.

We are aware of a limitation of Algorithm 2: 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 for a 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, because with $2^{24} =$ 16M 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 1bit each, but more realistically is limited to 5-8 dimension-uses (3-5 bits each). This results in a maximum of 8x to 32x I/O reduction in selection pushdown and already significant acceleration and memory reduction for sandwiched operators (see [3]), even if only a single dimension is involved. Extending Algorithm 2 is beyond our scope here, but future directions are (i) ignore dimension uses with less impact on a workload, or (ii) reconsider replication to create more opportunities for BDCC (one of the questions will be which dimensions to use for which replica).

Hierarchical dimensions as found in snowflake schemas occur if the key of one dimension determines the key of another. This is for example the case for `n_nationkey` with respect to `r_regionkey` in the TPC-H schema. Though our implementation exploits hierarchy among dimensions in rewrites, it treats all dimensions independently, because hierarchical bin numbering schemes [7] are very hard to maintain under updates. Hierarchical dimensions are an extreme form of correlated dimensions.

Correlated dimensions in a BDCC table $\texttt{T}_{\texttt{BDCC}}$ that is round-robin clustered on d dimension uses with b bits each, may lead to non-evenly distributed group sizes or even significantly less groups than the expected $2^{d \cdot b}$, even though the individual dimensions were created with a frequency-based binning algorithm. BDCC gathers statistics and adapts to such effects. During bulk-load, a special aggregation operator creates for each of the $d \cdot b$ possible count-table bit granularities a logarithmic group size histogram (entry x counts groups of size $[2^{x-1}, 2^x)$). Algorithm 1 uses this to choose a count-table granularity with the byte-size in the highest density column $\geq A_R$ for the vast majority of groups. If correlations or hierarchies among dimensions cause missing groups, this algorithm hence automatically chooses a higher count-table bit granularity to maintain good average group sizes. Hence, "puff pastry" does not hurt; BDCC always uses enough bits to get good selectivity.

After bulk-load, the low percentage of data in very small groups $\ll A_R$ tolerated by Algorithm 1 is copied and appended once more to table `T`, and the original very small groups are marked invalid in the count-table. Thus, very small groups get stored consecutively, generating better caching of these frequently re-accessed pages in the buffer pool.

IV. Evaluation

System setup. We evaluated on an Intel Xeon E5505 with 16GB main memory and 32KB L1, 256KB L2 and 4096KB L3 cache. The operating system is a 64 bit Debian, kernel version 2.6.32. Databases were stored on a RAID0 of 4 Intel X25M SSDs with a stripe 128KB and maximal bandwidth of 1GB/s. As BDCC does not yet support parallelization, queries are only executed on a single core. Vectorwise was set to use 4GB buffer space and 12GB query memory. The page size was 32KB. The group size was set to 1.

TPC-H Experiments. We use the 100GB TPC-H benchmark to compare an automatically created BDCC schema with a plain database without any indexing, and with a primary key indexed (PK) database and show that it is by far superior to both of them. Further we show that the BDCC setup achieves its performance gain across the full query set, showing that one BDCC schema without replication is sufficient in such a case. All three schemes use automatic compression, take roughly 55GB on disk, and are implemented end-to-end in the same system (Vectorwise), so the comparison is apples-to-apples.

Primary key scheme. One straightforward approach of indexing is to use primary key indexing. For the TPC-H setup this means that the `LINEITEM-ORDERS` join becomes a merge join as both tables share the major primary index key, also the `PARTSUPP-PART` join becomes a merge join. However, as many attributes that queries select on do not group the primary key, important dimensions cannot be recognized by this scheme and possibilities for selection pushdown are missed. In addition primary key indexing does not result in any co-locality for most of the tables and no optimizations similar to merge or sandwich techniques can be performed.

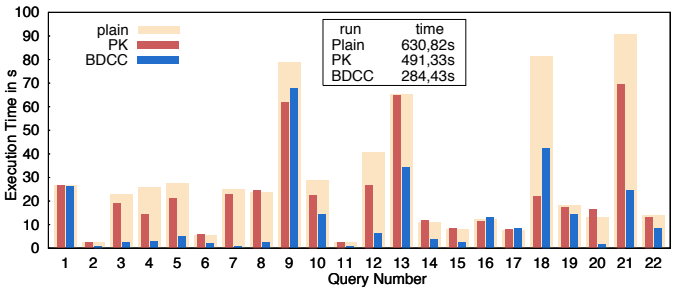

Fig. 2. Execution times of TPC-H SF100 for BDCC, PK and Plain storage schemes.

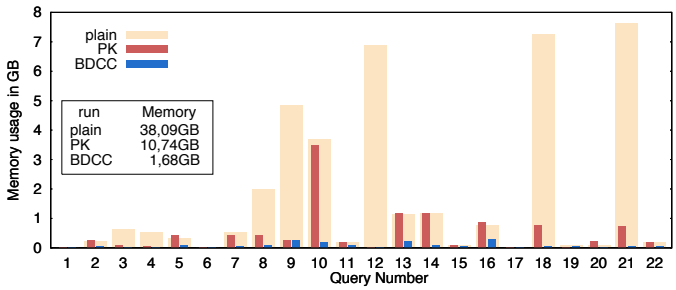

Fig. 3. TPC-H SF100 Memory Usage for BDCC, PK and Plain storage schemes.

BDCC scheme. We used Algorithm 2 to semi-automatically design the physical BDCC schema given as input DDL statements consisting of the usual foreign keys for TPC-H, plus

```
CREATE INDEX date_idx ON ORDERS(o_orderdate),
CREATE INDEX part_idx ON PART(p_partkey),
CREATE INDEX nation_idx ON NATION(n_regionkey, n_nationkey).
```

The latter compound key allows the query rewriter to detect that a region equi-selection determines a consecutive `D_NATION` bin range. Note, that in this setup, we limit ourselves to only key and date columns, which is closer to TPC-H restrictions, significantly simplifies the selection of create index statements, but may miss opportunities.

Using this DDL, Algorithm 2, which treats `CREATE INDEX` just as hints for BDCC, creates the following 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

In addition we declared indices on the foreign key references `o_custkey`, `s_nationkey`, `c_nationkey`, `l_orderkey`, `l_partkey`, `l_suppkey`, `ps_partkey` and `ps_suppkey` that are used to derive the co-clustering of the tables.

Algorithm 2 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). This way, `SUPPLIER` and `CUSTOMER` get clustered on `D_NATION`, and `ORDERS` also gets clustered on `D_NATION` (via `CUSTOMER`), as well as on `D_DATE`, which is a local dimension in `ORDERS`. `PARTSUPP` gets clustered on `D_PART`, and on `D_NATION`, here `D_NATION` is connected over foreign key via the `SUPPLIER` table. `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 – similar to dimension D1 in Figure 1. This yields the following dimension uses per table:

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	1010101010111111111
	D_NATION	FK_PS_S.FK_S_N	10101010100000000
ORDERS	D_DATE	-	1010101010111111111
	D_NATION	FK_O_C.FK_C_N	10101010100000000
LINEITEM	D_DATE	FK_L_O	1000100010001000
	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 550000 pages (using $A_B = 32$KB), Algorithm 1 chose to cluster `LINEITEM` using granularity $\lceil \log_2 550000 \rceil = 20$ bits.

In comparison to the PK scheme this setup looses the two efficient merge joins, but gains sandwiched execution across the whole schema instead, and because of its co-clustered setup, selections are more efficiently pushed down and propagated, again, across the whole schema.

Overall Results. Figure 2 shows cold execution times for all 22 TPC-H queries. The PK setup gains 139 seconds compared to the plain schema setup, mostly through the `LINEITEM-ORDERS` merge join. BDCC by far outperforms both schemes, more than twice as fast as plain and 208 seconds (42 %) faster than the PK setup, because co-clustering across multiple dimensions permits to push down these *and more* selections to *all* tables affected, and in addition accelerates more joins across the whole workload - this way compensating for loosing the merge join for `LINEITEM` and `ORDERS`. Vectorwise automatically creates MinMax indices on each table [8], which in addition to all dimension selections can push down selections correlated with a accelerated dimensions (e.g. shipdate selections are pushed down, as `LINEITEM` and `ORDERS` tables have orderdate locality). We can also see, that the automated BDCC schema accelerates almost all of the 22 queries proving to be suitable for the full query set. In Q01 there is no significant acceleration to be achieved with indexing methods as it is a 95%-97% full scan of the involved columns and in Q16 the Sandwiched aggregation does not pay back enough to cover up for the extra time spent in processing the extra `_bdcc_` column in the joins below.

Memory. Figure 3 shows that our automated BDCC schema not only proves to be fast, but also proves to be very memory efficient. Particularly we can see that with the automated setup we are able to tackle every significant memory intensive join or aggregation, as the setup includes all outgoing foreign key paths. This leads to predictable low memory consumption for all queries. This way, BDCC on average needs much less memory than plain (0.09GB vs. 1.59GB), and peak memory usage drops from 8GB to 275MB for SF100 TPC-H. Also, compared to PK, BDCC still is more memory efficient by a factor 6 (peak 13x), even with the "big" join gone for the PK setup, as BDCC reduces memory for *all* significant joins due to its co-clustering approach across the whole schema. For all experiments, we configured query memory such that

hash-operators would never spill, otherwise the differences to the memory intensive queries would have been much higher.

Detailed Analysis. BDCC accelerates Q02, Q3, Q4, Q5, Q7, Q8, Q10, Q11, Q14, Q15, Q20, Q21 and Q22 by selection pushdown and sandwich operators. In Q6, Q12 and Q20 the automatic BDCC setup benefits from the correlation of `o_orderdate` and `l_shipdate`, allowing MinMax indices to identify pushdown ranges. BDCC acceleration in Q09 and Q13 strictly comes from sandwiched execution of joins. In Q13, the `HashJoin(ORDERS,CUSTOMER)` is sandwiched based on the common customer `D_NATION` dimension, although `NATION` is not even involved in the query, but the join key `c_custkey` implies the nation of a customer. This sandwiching strongly reduces memory usage compared to PK, where a full materialization of the `CUSTOMER` columns is required. Same holds for Q10, Q18 and Q22. In Q14 the join to `PART` is reduced. In Q16 the count of the distinct `s_suppkeys` is sandwiched, shrinking the hash table by a factor of 25 at the cost of a HashJoin instead of MergeJoin between `PART` and `PARTSUPP`. Q18 performs a full aggregation of `LINEITEM` on `l_orderkey`; sandwiching helps here with respect to plain, but the streaming aggregate applied by the PK scheme cannot be beaten.

We acknowledge that a fine tuned setup may be faster for some of the queries, but may loose for others. For example Q16, Q17 or Q19 would benefit from explicitly indexing on `p_brand` or `p_container` or `p_type`. However, it is difficult to choose one column over the other without looking at the query set, also we restricted our index hints to key and date columns, which left us with `p_partkey` in this case.

Other Orderings. In a self comparison we also compared the automatic Z-order setup with a hand created major-minor setup, using the same dimensions and numbers of bits as shown in the tables above, favoring the important time dimension as the major dimension. Again, with buffer setup not requiring the adaptive scan, in order not to penalize the minor part dimension. Both runs are comparable, the automatic setup is slightly faster (284sec vs. 291sec).

V. Related Work

The problem of deriving good partitioning schemes has been studied in physical database design tuning [1], [9], [10] and others. Our focus here is not on partitioning but co-clustering tables in a single primary schema without replication.

Also index recommendation [1], [2] has been widely studied, but these approaches result in multiple index recommendations, where we only aim at a single co-clustered setup. The proposed administration tool in addition also suggest materialized views.

While [11] describes design techniques for multi-dimensional clustering, it does not consider the effects and opportunities based on the co-clustering approach, and is in that sense missing many opportunities to an efficient physical design.

Many approaches use workload monitoring to derive and modify a physical database design, examples are [12], [2], [1]. Here, we do not consider workload monitoring and analysis, but rather use to power of the co-clustered setup to define a robust physical schema, as it is much more costly to re-design the primary schema than adding/dropping additional views or indexes.

Our automatic schema creation algorithms use the concept of Z-ordering introduced in [13] and previously applied in Mistral [7], which explores bit interleaving in Z-order for multi-dimensional clustering.

VI. Conclusions

In this paper we show an easy to use automatic schema design for co-clustered tables that delivers a robust query performance for small to medium sized schemas. Many refinements of the proposed approach are possible, for example to take statistics about data or workload into account to find the initial set of dimensions automatically, or to combine the automated co-clustering with partitioning if the scenario requires it. We leave this for future work.

References

[1] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 Design Advisor: Integrated Automatic Physical Database Design," in *VLDB*, 2004.

[2] S. Agrawal, N. Bruno, S. Chaudhuri, and V. Narasayya, "Autoadmin: Self-tuning database systemstechnology," *IEEE Data Eng. Bull*, vol. 29, no. 3, pp. 7–15, 2006.

[3] S. Baumann, P. Boncz, and K.-U. Sattler, "Query Processing of Pre-Partitioned Data Using Sandwich Operators," in *BIRTE*, 2012.

[4] ——, "Creating Dimensions for BDCC," TU Ilmenau, www.dbis.prakinf.tu-ilmenau.de/publications/files/DBIS:DimEnc.pdf, Tech. Rep., July 2012.

[5] S. Baumann, G. de Nijs, M. Strobel, and K. Sattler, "Flashing Databases: Expectations and Limitations," in *DaMoN*, 2010.

[6] H. Leslie, R. Jain, D. Birdsall, and H. Yaghmai, "Efficient Search of Multi-Dimensional B-Trees," in *VLDB*, 1995.

[7] V. Markl, *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. Institut für Informatik TU München, 1999.

[8] D. Inkster, P. Boncz, and M. Zukowski, "Integration of VectorWise with Ingres," *SIGMOD Record*, vol. 40, no. 3, 2011.

[9] S. Agrawal, V. R. Narasayya, and B. Yang, "Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design," in *SIGMOD*, 2004.

[10] C. Baldwin, T. Eliassi-Rad, G. Abdulla, and T. Critchlow, "The Evolution of a Hierarchical Partitioning Algorithm for Large-scale Scientific Data: Three Steps of Increasing Complexity," in *SSDBM*, 2003.

[11] S. S. Lightstone and B. Bhattacharjee, "Automated design of multidimensional clustering tables for relational databases," in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 1170–1181. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316789

[12] N. Bruno and S. Chaudhuri, "Constrained physical design tuning," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 4–15, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1453856.1453863

[13] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," ser. PODS '84. [Online]. Available: http://doi.acm.org/10.1145/588011.588037