Dwarfs in the Rearview Mirror: How Big are they Really?

Jens Dittrich Lukas Blunschi Marcos Antonio Vaz Salles

ETH Zurich
8092 Zurich, Switzerland
systems.ethz.ch

ABSTRACT

Online-Analytical Processing (OLAP) has been a field of competing technologies for the past ten years. One of the still unsolved challenges of OLAP is how to provide quick response times on *any* Terabyte-sized business data problem. Recently, a very clever multi-dimensional index structure termed Dwarf [26] has been proposed offering excellent query response times as well as unmatched index compression rates. The proposed index seems to scale well for both large data sets as well as high dimensions. Motivated by these surprisingly excellent results, we take a look into the rearview mirror. We have re-implemented the Dwarf index from scratch and make three contributions. First, we successfully repeat several of the experiments of the original paper. Second, we substantially correct some of the experimental results reported by the inventors. Some of our results differ by orders of magnitude. To better understand these differences, we provide additional experiments that better explain the behavior of the Dwarf index. Third, we provide *missing* experiments comparing Dwarf to baseline query processing strategies. This should give practitioners a better guideline to understand for which cases Dwarf indexes could be useful in practice.

1. INTRODUCTION

Online analytical processing (OLAP) has been an area of interest for the past ten years [31, 5]. In contrast to OLTP-systems, OLAP systems require quite different access patterns to data. For instance, OLAP queries are mostly value-oriented instead of key-oriented. Therefore, OLAP queries typically touch considerable portions of the data. Moreover, OLAP scenarios comprise Terabyte-sized data volumes [32] including data from hundreds of OLTP source databases. In addition, OLAP queries may involve dozens of join operators on complex galaxy schemas[1]. In addition, OLAP queries are navigational [7].

Therefore, it had become clear very early that the relational database engines of the early 90ies were not appropriate to cope with the massive data volumes, large query plans, multi-dimensional schemas, and sheer complexity of OLAP problems.

Since then, vendors have followed three different approaches to implement efficient OLAP engines:

(1.) Relational OLAP (ROLAP). This approach implements multi-dimensional schemas directly on top of an existing relational DBMS; however, it extends DBMSs by appropriate indexing techniques to speed-up query processing. Important extensions were bitmap-indexes which were pioneered by Model 204 [22], Sybase IQ [8, 9], and Oracle. Other important extensions were materialized views [12] and the CUBE-operator [11].

(2.) Multi-dimensional OLAP (MOLAP). This approach implements multi-dimensional schemas on top of a multi-dimensional database system[2]. Systems include Essbase (first acquired by Hyperion and then by Oracle) [15] and Microsoft Analysis Services [17]. Note that it is also possible to combine ROLAP and MOLAP, i.e., parts of the data are kept in the ROLAP, others in MOLAP. This is termed Hybrid OLAP (HOLAP).

(3.) Column-oriented OLAP (VROLAP[3]). This approach implements multi-dimensional schemas on top of a relational DBMS using a column-oriented storage manager based on vertical partitioning [1, 6]. Examples include Sybase IQ [8, 9], KDB [16], MonetDB [4], SAP BI Accelerator [29], and, more recently, Vertica [30]. Column stores have only recently become more popular due to the shift from external to main memory databases.

The three approaches may be judged by three different features: (1) effort for index selection, (2) predictability of the (nightly) index rebuild process, and (3) predictability of query response times. Table 1 gives a brief summary on the three features for the different OLAP approaches.

Approach	Index Selection Effort	Index Rebuild Time Predictability	Query Response Time Predictability
ROLAP	Bad	Good	Bad
MOLAP	Good	Bad	Very Good
VROLAP	Good	Very Good	Good

Table 1: Comparison of competing OLAP approaches

Obviously, VROLAP seems to perform well in all three aspects. This is mainly due to the fact that query plans in a VROLAP system are mostly simple column scans. In addition, maintenance of

[1]Using a star schema is already an optimization: it simply replaces all non-fact tables by materialized views.

[2]Some definitions for MOLAP are restricted to OLAP systems working on array representations. In contrast, we include any approach working on top of a multi-dimensional storage manager/index structure.

[3]There is no accepted acronym for this approach. We term it vertical relational OLAP here.

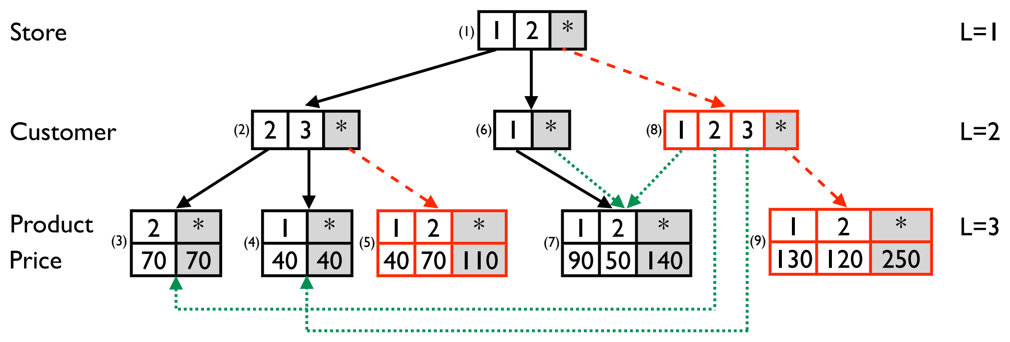


Figure 1: Recap of running example: the Dwarf index for Table 2 (see Figure 1 in [26])

materialized views is rarely required. In contrast, MOLAP has better response times, however, at the cost of possibly unpredictable indexing times.

The goal of this paper is to examine a recently proposed very promising approach for MOLAP: Dwarf [26]. The proposed *Dwarf index* seems to work well for both large data sets as well as high dimensions. Experimental results in [26] indicate that Dwarf indexes work for both uniformly distributed as well as highly skewed data sets. Both index sizes as well as indexing times seem to be controllable.

Motivated by these seemingly excellent results, this paper takes a look into the rearview mirror. We have re-implemented the Dwarf index from scratch and carefully examine its performance characteristics. In this paper we make the following contributions:

(1.) Repeating Experiments. We repeat the experiments of the original paper using a re-implementation of Dwarf. We successfully repeat several of the experiments as presented by the inventors. Our results indicate that results using uniformly distributed data match the ones reported by the inventors.
(2.) Correcting Experiments. We make substantial corrections to the experimental results reported by the inventors. In particular, for skewed data sets we obtain Dwarf index sizes that are by orders of magnitude higher than reported by the inventors – even though Dwarf index sizes are hardware independent. As a consequence, we also obtain Dwarf index construction times that are by orders of magnitude higher than reported by the inventors — even though we are using much better hardware. In summary, our results indicate that Dwarf indexes seem to be hard to control for skewed data sets.
(3.) Additional Experiments. To understand why our results differ, we provide additional experiments that better explain the behavior of Dwarf indexes. This includes experiments using Zipf distributions as well as experiments using different parameters than the ones used by the inventors. Furthermore, we report index sizes in relation to the original fact table size (compression ratio) instead of comparing index sizes to a fully materialized exponentially-sized cube. Moreover, we compare index sizes to a later work of the inventors [28] measuring the number of coalesced tuples.
(4.) Guideline. We provide *missing* experiments comparing Dwarf indexes to baseline query processing strategies. We add query performance experiments comparing Dwarf indexes with simple table scans which are common in OLAP query engines. This should give practitioners a better guideline to understand for which cases Dwarf indexes could be useful in practice.

This paper is structured as follows. The following section recaps Dwarf indexes. Section 3 presents details of our Dwarf implementation. Section 4 details the setup used for our experiments. Section 5 presents results of Dwarf index construction experiments. Section 6 presents results of query processing experiments. Section 7 discusses related work.

2. RECAP: THE DWARF INDEX

In this section we briefly review the Dwarf index. This section is meant to be self-contained. We will try to keep the description concise and refer the interested reader to [26] for details.

We will first explain the Dwarf structure (Section 2.2) using a running example (the same as used by the inventors). After that we recap the Dwarf construction algorithms (Section 2.3).

2.1 Running Example

We explain the Dwarf index by using the running example of [26]. Let's assume we want to create a Dwarf index for a simple three-dimensional fact table as displayed in Table 2.

Store	Customer	Product	Price
1	2	2	70
1	3	1	40
2	1	1	90
2	1	2	50

Table 2: Running example fact table from [26]

The table consists of three dimensions *Store*, *Customer*, and *Product* and single measure *Price*. Dwarf indexes for multiple measures are also possible; however, similar to the inventors, we will assume a single measure for the remainder of this paper. We assume an aggregation function `SUM` on that measure. Other aggregation functions may also be supported.

2.2 Dwarf Structure

The main idea of Dwarf indexes is to precompute all possible views of a given fact table at indexing time and represent all views in a single compact structure: the Dwarf index. This means that at query time no aggregation computation has to be performed anymore. Instead, query processing is restricted to simply looking up values in the Dwarf index.

For our running example this means that all possible views ($2^3 = 8$) are already precomputed. The Dwarf index that will be created based on the Running Example is displayed in Figure 1. We will explain the different aspects of a Dwarf index stepwise:

(1.) Node Layout and Prefix Compression. A Dwarf index for a cube of D dimensions consists of exactly D levels (one for each dimension). The root node corresponds to the first level $L = 1$. Nodes on levels $L < D$ are termed *non-leaf-nodes*. Nodes on the D-th level are termed *leaf-nodes*. Non-leaf-nodes contain mappings from a key to a node at level $L+1$. Leaf-nodes contain mappings from a key to a measure value. A Dwarf index applies prefix-compression on the data. This means that each prefix is stored only once. For instance, rows $(2,1,1;90)$ and $(2,1,2;50)$ have the common prefix $(2,1)$. Therefore, this prefix only has to be stored once. This effect is similar to the one used in search tries [14]. The aforementioned properties of a Dwarf index can be seen in Figure 1 when only considering black framed cells and arrows $\longrightarrow$.

(2.) Aggregation Cells and Suffix Compression. Each node in a Dwarf contains an additional aggregation cell "$*$". In Figure 1 aggregation cells are shaded in grey. For leaf-nodes, an aggregation cell maps to a measure, e.g., consider node (3) mapping "$*$" to value 70. For a non-leaf node at level $L < D$ an aggregation cell points to a node at level $L+1$. It may either point to a newly created node (red framed cells), i.e., a node that would not exist in the prefix-tree. Or it may point to an existing node. For instance, node (2) contains an aggregation cell pointing to a newly created node (5). Aggregation cells pointing to newly created nodes are displayed using arrows $\dashrightarrow$. Alternatively, aggregation cells for non-leaves may also point to existing nodes. For instance, the aggregation cell of node (6) points to the already existing node (7). This effect is termed *suffix compression* (or suffix coalescing) as possibly redundant suffixes are only stored once. This property turns the tree into a DAG. It is displayed using arrows $\dashrightarrow$. For instance, if we follow one of the paths $(2,1,2)$ or $(2,*,2)$, or $(*,1,2)$ we will end up at the same cell of node (7) providing the mapping $2 \mapsto 50$.

2.3 Dwarf Construction Algorithms

In this section we briefly review the core idea of Dwarf index construction algorithms. The Dwarf construction process consists of two Algorithms `CreateDwarfCube` (Algorithm 1, corresponds to Algorithm 1 in [26]) and `SuffixCoalesce` (Algorithm 2, corresponds to Algorithm 2 in [26]). Again, we will describe both algorithms in our own words.

Dwarf index construction is started by invoking `CreateDwarfCube` on the unsorted input fact table F and specifying the maximum number of dimensions D.

`CreateDwarfCube`. This method takes as input an unsorted fact table F and the maximum dimension D. The methods starts by sorting F into a sorted sequence F' (Line 2). Then the method iterates over all tuples in F' (Lines 5–34). The main idea here is to compare the tuple treated in the previous iteration (*lastTuple*) to the tuple treated in the current iteration (*nextTuple*). This will determine a suffix suf_{last} of keys in which *nextTuple* differs from *lastTuple* (Line 6). Based on this information we determine the Nodes $N_{i+1},\ldots,N_D$ for which we need to compute $*$-cells (Lines 7–16). In more detail, for each leaf-level node, we compute its aggregate measure (Line 10) by calling an `aggregate` function. For non-leaf-nodes we call `SuffixCoalesce`. That method takes as its input the children of the current node N_j as well as the current dimension increased by one (Line 14). If all nodes have computed their $*$-cells, we check whether we need to create additional Dwarf nodes (Lines 17–30), i.e., in order to store new prefixes in the structure. If we are already on the last level of the Dwarf (Line 21), we create a leaf-node (Lines 22–25) and insert an appropriate mapping into that node (Lines 23&24). Otherwise we create a non-leaf-node (Lines 26–29) and insert an appropriate mapping into the new node (Line 28). This insertion recursively creates additional nodes on lower levels similar to a recursive insert on any tree-structure (Line 29, omitted). After that the new node is inserted into the existing Dwarf (Line 32). The loop of Lines 5–34 terminates when all input tuples of F' have been treated. After that, the root node of the Dwarf computes its $*$-cell by calling `SuffixCoalesce` and assigning its $*$ entry (Line 36).

`SuffixCoalesce`. This method takes as input a set of Dwarf nodes *nodes*, the maximum dimension D, and the current tree-level

Algorithm 1: CreateDwarfCube

```
   Input: FactTable $F$,
   MaxDimension $D$.
   Output: DwarfIndex $dwarf$.
1  //sort fact table:
2  $F' = \mathrm{sort}(F)$
3  $lastTuple$ = <empty>
4  // iterate over sorted fact table:
5  foreach Tuple $nextTuple \in F'$ do
6      Suffix $suf_{last}$ = suffix of $lastTuple$ differing from $nextTuple$
7      // Determine nodes that need to compute $*$-cell:
8      Let $N_i,\ldots,N_D$ be the nodes corresponding to suffix $suf_{last}$
9      if $i$ != $D$ then
10         $N_D.* = \mathtt{aggregate}(N_D)$
11         //call SuffixCoalesce bottom-up:
12         foreach Node $N_j \in N_{D-1},\ldots,N_{i+1}$ do
13             // call SuffixCoalesce for node $N_j$:
14             $N_j.*$ = SuffixCoalesce($N_j$.children(), $D$, $j+1$)
15         end
16     end
17     // Determine additional Dwarf nodes to be created:
18     Node $currentNode = N_i$
19     Node $newNode$= <empty>
20     Integer $currentDim = i+1$
21     if $currentDim == D$ then
22         $newNode$ = new Leaf()
23         Integer $key$ = nextTuple.getKey().getDim($currentDim$)
24         $newNode$.addMapping($key \mapsto$ nextTuple.getMeasure())
25     else
26         $newNode$ = new NonLeaf()
27         //create $D-i-1$ additional nodes recursively:
28         $newNode$.insert($currentDim$, $D$, $nextTuple$)
29         [...]
30     end
31     //insert newly created sub-Dwarf into existing Dwarf:
32     $currentNode$.addMapping(nextTuple.getKey($i$) $\mapsto$ $newNode$)
33     $lastTuple = nextTuple$
34 end
35 //compute $*$-cell for root node of dwarf:
36 $dwarf$.getRoot().$*$ = SuffixCoalesce($dwarf$.getRoot().children(),
       $D$, 2)
37 // return new dwarf index:
38 return $dwarf$
```

level. The method first checks whether the set *nodes* contains only a single element (Line 1). If that is the case, it simply returns the single node contained in *nodes* (Line 2). This is where suffix coalescing happens. If the node set contains more than one element (Lines 3–23), a new node is created (Lines 4–9). The latter depends on the current level *level*, which will determine whether to create a non-leaf or a leaf-node (Line 5). After that, we loop over all distinct keys available in the set of input nodes (Lines 10–21). For each key we compute the subset $containsKey \subseteq nodes$ having that key (Line 12). For each of those nodes we then unfold the corresponding mapping into a set of values (Line 13). This set may either contain nodes or measures. If we are already on the last level (Line 14), we aggregate the measures contained in *values* into an aggregate measure and append a mapping to *newNode* (Line 16). Otherwise, i.e., if we are on a non-leaf-level, we call `SuffixCoalesce` on the nodes contained in *values* and append a mapping with the result to *newNode* (Line 19). If all distinct keys in *nodes* have been processed, we return *newNode* (Line 22). Note that [26], as well as our implementation, uses a merge-based implementation of Lines 11–21. However, these lines could also be implemented using a hash-based approach.

External Memory. In our description of the algorithms so far we have assumed the entire Dwarf structure to fit into main memory. Obviously, this would not make sense in a real application. Therefore, in sync with the inventors, we assume an external memory

Algorithm 2: SuffixCoalesce

```
   Input: Input: DwarfNodeSet $nodes$,
   MaxDimension $D$,
   Integer $level$.
   Output: Node $subDwarf$.
1  if $nodes.size() == 1$ then
2      return $nodes$.firstEntry()
3  else
4      Node $newNode$= <empty>
5      if $level == D$ then
6          $newNode$ = new Leaf()
7      else
8          $newNode$ = new NonLeaf()
9      end
10     //for each distinct key available in the set of $nodes$:
11     foreach $Integer\ key \in nodes$ do
12         DwarfSet $containsKey$ =
           $\{node \mid node \in nodes \wedge node.\text{hasKey}(key)\}$
13         Set $values$ = $\{node.\text{getValue}(key) \mid node \in containsKey\}$
14         if $level == D$ then
15             //aggregate measures:
16             $newNode$.addMapping($key \mapsto$ aggregate($values$))
17         else
18             //aggregate nodes:
19             $newNode$.addMapping($key \mapsto$ SuffixCoalesce($values$))
20         end
21     end
22     return $newNode$
23 end
```

implementation. The main idea of this implementation is to replace node pointers by offsets. Furthermore, nodes may be written to disk whenever their $*$-cells have been determined. Therefore, in theory, we only have to keep one node per level in main memory, plus the nodes required to compute "$*$" cells in `SuffixCoalesce`. An interesting property of Dwarf is that each input tuple only has to be visited once. Therefore reading the input data is fully sequential. In addition, Dwarf nodes are written to disk fully sequentially. Thus, the `CreateDwarfCube` Algorithm presented above is in fact a *bulk loading* algorithm for Dwarfs. However, `SuffixCoalesce` may implicitly perform *considerable* random I/O as the recursive all node construction (Lines 13&19) may touch large portions of the DAG. In the worst case this method will therefore revisit *all* descendant nodes of all input *nodes*. As `SuffixCoalesce` is called for each non-leaf-node, this may be very expensive.

Coarse-Grained Dwarfs and $\mathbf{G_{min}}$. The inventors have already observed that the size of a Dwarf index as constructed by the algorithms described above may be too high to be useful in practice. Therefore they propose a technique to reduce the size by trading index size for query performance. The idea is to not initialize certain $*$-cells. This works as follows: whenever for a node N_i at any level of the Dwarf the number of tuples that contributed to the subdwarf beneath N_i is less than a given parameter G_{min}, the $*$-cell of N_i will not be computed. As a consequence, aggregation queries that would follow the $*$-cell of N_i have to resort to performing aggregations at query time. This leads to higher query runtimes (see also Section 4.4 in [26]).

Coalesced Tuples. In a follow-up paper [28] the inventors proposed an implementation-independent measure for the size of a Dwarf index termed the number of coalesced tuples. We argue that the number of coalesced tuples is a measure to count the *number of semantically different* measure cells. For instance, in the Running Example of Figure 1 of [26] node (4) contains value \$40 for key P1. The same value will be replicated in node (5) for the same key by the Dwarf construction algorithm. However, in Table 2 of [28] both values \$40 are counted only once. Therefore, the number of coalesced tuples is only a lower bound for the size of the Dwarf index. This is because a Dwarf index *does not* compress to semantically different measure cells only but to a larger number of cells. Also note that this property of a Dwarf index cannot be reduced by introducing additional enhancements like pointers on the leaf-level. For the settings used in [26, 28] and this paper the resulting Dwarf sizes would be exactly the same. Still we will use this lower bound measure to make an additional comparison to the results of [28].

3. OUR IMPLEMENTATION OF DWARF

We have implemented Dwarf strictly following [26]. All code was implemented using Java 5. We avoided object-orientation wherever possible and used native types. We used the Dwarf construction algorithm as described in the original paper and recapitulated above. In addition, in order to make our Dwarf implementation efficient, we used the following optimizations:

(1.) Writable and Read-only Nodes. A Dwarf uses two different node types: *leaves* and *non-leaves*. Implementation-wise, however, it makes sense to keep two different representations: two for each of the leaf and the non-leaf nodes. This results in four different node types. This is because the Dwarf index creation algorithm basically needs to keep only a single writable node for each level — all other nodes may be considered read-only. As a writable node-representation is less space-efficient than a read-optimized version, it makes sense to provide two different implementations. Our writable implementation of nodes is based on linked lists. All writable nodes are pinned in main memory. As soon as a node has computed its $*$-cell, the node will be replaced by a read-optimized version and unpinned.

(2.) Compact Node Representation. The node representation itself is based on integers as in the original paper. Each node contains a header of length 4 Bytes that indicates the node type (for deserialization) and the number of entries. A node may contain at most 2^{30} entries. If the node has N entries, we first store the N keys contiguously. After that, in case of non-leaf nodes we store N offsets pointing to lower-level nodes, and in case of leaf nodes we store the N values. This provides better cache behavior for intra-node searches. Our experiments give some evidence that our node representation is slightly more effective than the one used by the inventors.

(3.) Optional Aggregate Representation. There are some cases when the $*$-cell does not have to be represented: (1) if the $*$-cell is trivial, i.e., the node only has a single entry. Then the $*$-cell would point to the same entry. Therefore, we do not serialize it in our implementation. This implicitly implements the case $G_{min} = 1$. However, in addition, we also drop the $*$-cell if the current node contains a single prefix cell leading to multiple tuples in the sub-Dwarf. (2) if the aggregate node is non-existent due to a coarse-grained Dwarf (see above). In both cases we use the node header to indicate that there is no aggregate node. Again, this saves some storage space — without changing the actual Dwarf algorithms.

(4.) Efficient I/O. As the I/O-classes of `java.io` are rather inefficient, we used the newer I/O-classes of `java.nio`. These classes perform considerably better than the old `java.io`. We implemented the Dwarf index using a memory mapped file which provides all buffering. As each memory-mapped area was limited to 2 billion Bytes, we implemented a segmented buffer working on a set of F memory-mapped files. In our implementation, non-leaf nodes contain integer offsets to nodes in the memory mapped array. Note that we map data to integer arrays rather than byte arrays. Due to this data layout of Dwarfs we do not loose any storage space. This allows us to support a maximum Dwarf index size of up to

16 GB even though we are only using integers for offsets[4]. Also note that we do not use pointers in the tree-structure at any point in time as this would be space-inefficient. Furthermore, our read-optimized node representation does not copy any data from the memory-mapped array but rather operates directly on the memory-mapped array which saves even more memory space (similar to a C++ implementation).

4. EXPERIMENTAL SETUP

4.1 Hardware

All experiments were performed on servers having each two 2.4 GHz Dual Core AMD Opteron 280 processors, i.e., four cores in total, and 6 GB of main memory. The operating system used was Linux 2.6.9. The disk used was a 300 GB ATA/133 hard disk with 16 MB Cache (Maxtor 6L300R0). It has an average seek time of <9 ms and a data transfer rate of 70 MB/sec. In order to have better control on I/O, we turned of swapping. Our implementation does not make use of multiple threads. However, to prevent the JVM from implicitly using more than one computing core we switched off all cores except one. Still, our hardware is of course better than the hardware available in 2002. In [26] the authors used a 700 Mhz Celeron processor running Linux 2.4.12 with 256 MB of RAM. The hard disk used was a 30 GB disk rotating at 7200 rpms, able to write at about 8 MB/sec and read at about 12 MB/sec. The average disk seek time was not specified. Obviously, due to improved hardware we expect our runtime measurements for index construction and query response times to be *always* by a factor better than the results reported by the inventors in [26]. However, note that as the index size is hardware independent, index sizes should match precisely the results obtained by the inventors.

4.2 Data Sets

Synthetic. We used synthetic as well as real data sets. Similarly to [26] we used data where dimension values follow either a uniform or a self-similar 80-20 distribution [10] over a given cardinality. Similarly to the inventors, we did not impose any correlations among the dimensions. Note that [26] does not specify whether the self-similar distribution of [10] or a zipfian distribution was used. These distributions are often confused, e.g., see the discussion in [10]. In this paper we present results of both self-similar and Zipf distributions. Random numbers were generated using FastMersenneTwister [21]. Both self-similar 80-20 and Zipf distributions were generated using the methods described by Gray et al. [10]. Unless specified otherwise, we use a value $\theta = 0.95$ for the Zipf distribution.

Data Set	Description	Raw Size [MB]	#Tuples
`Weather-FULL`	Weather Full Prec	39	1,015,367
`Weather-TRUNC`	Weather Truncated	39	1,015,367
`FOREST`	Forest	24	581,012

Table 3: Raw size and #Tuples of real data sets used

Weather Data Set. In [26], the inventors of Dwarf evaluate their technique with real data sets, among them the Weather data set [13]. A subset of [13] was chosen corresponding to weather conditions at various weather stations on land for September 1985. While the inventors report that the data set had 348,448 tuples, the original data from [13] actually contains 1,015,367 tuples[5]. We believe this discrepancy could stem from the fact that [26] took only a subset of the dimensions from the data set. In fact, [26] reports creating two derived data sets, one with 9 dimensions (Meteo-9) and one with 12 dimensions (Meteo-12), from the original Weather data. We believe that the inventors have pre-aggregated the data according to those dimensions, but that remains unclear in [26]. Furthermore, [26] does not report which dimensions were used. This means that it is unclear how to repeat their experiments for this data set.

Data Set	Dimension	Card.	Min Val	Max Val
`Weather-FULL`	station-id	7037	1001	98851
	longitude	5359	5	36000
	latitude	3809	-8999	8250
	solar-altitude	1782	-891	896
	present-weather	101	-1	99
	day	30	1	30
	weather-change-code	10	0	9
	hour	8	0	21
	brightness	2	0	1
`Weather-TRUNC`	station-id	7037	1001	98851
	longitude	352	0	360
	solar-altitude	179	-89	89
	latitude	152	-89	82
	present-weather	101	-1	99
	day	30	1	30
	weather-change-code	10	0	9
	hour	8	0	21
	brightness	2	0	1
`FOREST`	slope	5827	0	7173
	horz-dist-to-roadways	5785	0	7117
	horz-dist-to-fire-points	1978	1859	3858
	vert-dist-to-hydrology	700	-173	601
	horz-dist-to-hydrology	551	0	1397
	aspect	361	0	360
	hillshade-noon	255	0	254
	hillshade-3pm	207	0	254
	hillshade-9am	185	0	254
	elevation	67	0	66

Table 4: Dimensions, cardinalities, and key ranges of real data

We have, therefore, resorted to the literature to understand similar uses of the same Weather data set. We have found, however, divergent uses of the data set in the literature. In [19], for example, the data set is reported to have 6 dimensions out of the 20 existing. In contrast, all of [20, 18, 2, 23] report using a superset consisting of 9 dimensions. They report these dimensions (with respective cardinalities) to be: station-id (7,037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8), and brightness (2). We have tried to verify the cardinalities of those columns listed in the literature, but we found inconsistent numbers for the longitude (5,359), latitude (3,809), and solar-altitude (1,782) columns. In fact, in the original data set, these columns represent fixed precision floating point numbers. When truncating the numbers on the three columns, we have obtained the same cardinalities reported in the literature. As we believe that by performing truncation we are altering the original features of the data set, we have decided to conduct our experiments with *both* the *full precision* and with the *truncated* Weather data used in literature.

Forest Data Set. The Forest data set [3] includes forest cover type data for areas in the Roosevelt National Forest of northern Colorado. It contains 581,012 tuples and 55 columns of data. Out of these columns, only 10 are quantitative variables and one is a mea-

[4]A minor detail here is that Java does not provide an `unsigned int` type. However, in our implementation we use the full range of 32 bits by letting Integer offsets start at the minimal negative integer value. These offsets are than translated back to `long` typed offsets in the buffer.

[5]We pointed this out to the inventors on Feb 25, 2008, but did not receive any reply. We also asked for the source code used in the original paper. However, the code was not made available by the inventors due to technology licensing issues.

D	F.Tbl. Size [MB]	Uniform Dwarf [MB] ORIG	Uniform Dwarf [MB] NEW	Uniform Time [sec] ORIG	Uniform Time [sec] NEW	80-20 Dwarf [MB] ORIG	80-20 Dwarf [MB] NEW	80-20 Time [sec] ORIG	80-20 Time [sec] NEW	Zipf ($\theta = 0.95$) Dw.[MB] NEW	Zipf ($\theta = 0.95$) T.[sec] NEW
10	4	62	55	26	11	115	152	46	26	118	22
15	6	153	141	68	22	366	1293	147	195	515	85
20	8	300	283	142	42	840	7779	351	1123	1570	234
25	10	516	495	258	66	1788	n/a	866	n/a	3851	548
30	12	812	790	424	102	3063	n/a	1529	n/a	8272	1153

(a) Dwarf index sizes and creation times

Uniform #coalesced Tuples	80-20 #coalesced Tuples	Zipf ($\theta = 0.95$) #coalesced Tuples
0.3 E6	2.3 E6	1.3 E6
0.6 E6	16.4 E6	4.1 E6
1.0 E6	88.6 E6	10.2 E6
1.5 E6	n/a	21.3 E6
2.2 E6	n/a	39.8 E6

(b) #coalesced tuples

Figure 2: Storage size and creation time vs #dimensions. Repeats and complements Table 4 from [26]. Cardinalities for all dimensions equal 1,000. Fact table size = 100,000 tuples.

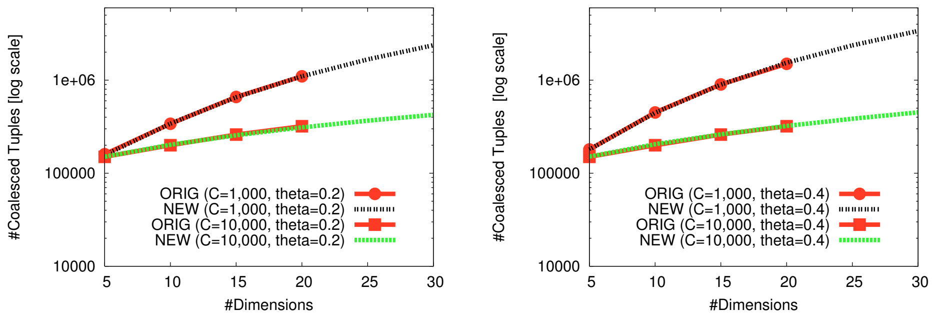
D	Fact Table Size [MB]	Uniform Dwarf [MB]	Uniform Time [sec]	80-20 Dwarf [MB]	80-20 Time [sec]	Zipf ($\theta = 0.95$) Dwarf [MB]	Zipf ($\theta = 0.95$) Time [sec]
10	<1	5	<1	13	3	9	2
15	1	11	2	82	13	30	5
20	1	19	3	410	72	79	12
25	1	31	5	1763	276	170	25
30	1	44	6	6537	1016	325	43

(a) Dwarf index sizes and creation times

D	Uniform #coalesced Tuples	80-20 #coalesced Tuples	Zipf ($\theta = 0.95$) #coalesced Tuples
10	0.02 E6	0.17 E6	0.08 E6
15	0.03 E6	0.95 E6	0.22 E6
20	0.04 E6	4.33 E6	0.45 E6
25	0.05 E6	17.52 E6	0.83 E6
30	0.06 E6	61.44 E6	1.40 E6

(b) #coalesced tuples

Figure 3: Storage size and creation time vs #dimensions. Additional experiment complementing Table 4 from [26] with results for 10,000 tuples. Cardinalities for all dimensions equal 1,000.



(a) zipf-factor $\theta = 0.2$

(b) zipf-factor $\theta = 0.4$

Figure 4: Number of coalesced tuples versus #dimensions. Repeats and complements Figure 12 from [28]. Fact table size 100,000 tuples. Logarithmic scale on vertical axis.

sure. The remaining 44 columns are binary qualitative indicators. Following the methodology of [26], we have taken the 10 quantitative variables as dimensions.

We summarize the features of the real data sets used in our experiments in Tables 3 and 4. Note that we have ordered the dimensions of all data sets in descending order of cardinalities, as this ordering benefits the Dwarf.

4.3 Outline of Experiments

We will examine two principal features of Dwarf indexes: (1) index size and construction time (Section 5) and (2) query performance (Section 6).

We repeat the most important experiments of [26], i.e., those experiments that give the best understanding on how Dwarf indexes scale for certain scenarios. We also repeat some experiments of [28]. We will use the same data sets, query workloads, and parameters as specified by the inventors wherever possible. We measure the same characteristics of the Dwarf index including index creation time, index size, and query response time. In addition, we show the number of coalesced tuples as proposed in [28] which is a hardware- and implementation-independent measure for the index size. As explained in Section 2.3 it is a lower bound on the actual entries in a Dwarf index. Furthermore, we provide additional experiments that help to understand better the performance characteristics of Dwarf indexes. This includes experiments for index creation as well as for query processing. For index creation we display additional observations of Dwarf index creations including: (1) size of a Dwarf index when compared to fact table size, i.e., the actual *index expansion ratio*, and (2) behavior of Dwarf indexes under different skewed distributions (80-20 self-similar and Zipf). For query processing we provide comparisons with realistic baselines, i.e., *scans on row- and column-stores*. This will be explained in more detail below.

In the experiments we will refer to results presented in the original work [26] and [28] as ORIG. Where appropriate, results obtained from our implementation are labeled NEW. For additional experiments we omit to specify NEW.

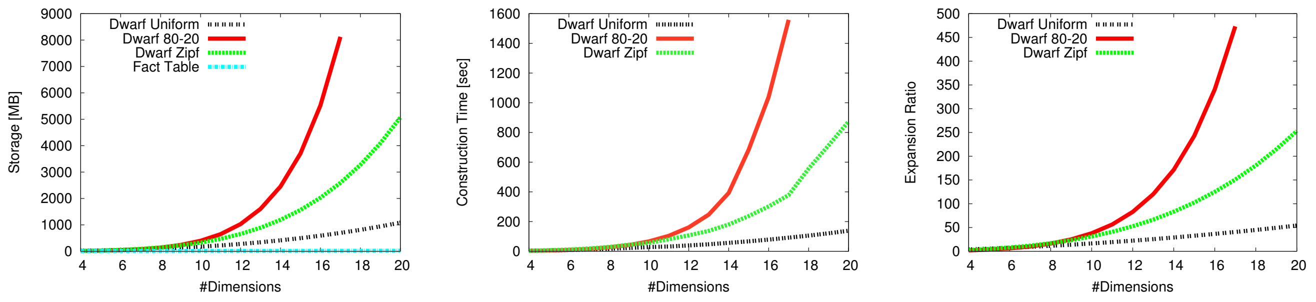
5. INDEX CONSTRUCTION EXPERIMENTS

In this section we examine Dwarf index construction times and index sizes.

5.1 Scaling in the Number of Dimensions

We discuss how Dwarf index sizes and construction times scale with the number of dimensions D.

This experiment corresponds to Table 4 from [26]. Like the inventors, we have constructed Dwarf indexes for increasing numbers of dimensions all of which had cardinalities set to 1,000. The number of tuples in the fact table was always 100,000. Again like the inventors, we have generated the data using uniform and self-similar 80-20 distributions. As described in Section 5.1.1 of [26], dimension values were generated to follow the given distribution



(a) Storage Space vs #Dimensions (b) Construction Time vs #Dimensions (c) Expansion Ratio vs #Dimensions

Figure 5: Storage space, Construction Time, and Expansion Ratio vs #Dimensions. Repeats and complements Figures 3 & 4 from [26] by showing up to 20 dimensions and also a Zipf distribution. In addition, we compare the size of Dwarf indexes to the size of the raw fact table.

over the cardinality of the dimensions. Furthermore, we also did not impose any correlation among the dimensions. In addition to the original evaluation, however, we have also used a Zipf distribution. The results we have obtained are shown in Figure 2(a).

Effect of Uniform Data. For a uniform distribution, our results are very consistent with the ones from [26]. Our index sizes were always slightly smaller than the ones reported in [26] as we have introduced the storage optimizations described in Section 3. Moreover, the index construction times were proportional to the index sizes[6]. We observe that the Dwarf index is always bigger than the fact table. For $D = 10$, Dwarf takes 55 MB while the fact table takes 4 MB, a 13:1 expansion ratio. The ratio becomes even worse for higher dimensionality. For $D = 30$, Dwarf takes 790 MB compared to only 12 MB for the fact table, resulting in an expansion ratio of 66:1.

Effect of Self-Similar Data. For a self-similar 80-20 distribution, however, the Dwarf index sizes differ significantly from the ones reported in [26] for $D \geq 15$. In fact, we observe a strong growth in the Dwarf index size, reaching 1.3 GB for 15 dimensions and 7.8 GB for 20 dimensions (expansion ratios 216:1 and 972:1, resp.). This is by a factor 3 to 9 larger than reported by the inventors. In addition, we could not scale Dwarf beyond 20 dimensions, as the index size was in excess of 16 GB, the maximum index size supported in our implementation (see Section 3). In order to better understand those differences, we have taken three steps: (1) we have also evaluated Dwarf using a Zipf distribution to understand the sensitiveness of Dwarf to skew; (2) we have compared the number of coalesced tuples as obtained from our implementation with the one of the inventors presented in [28], (3) we have repeated the same experiment with only 10,000 tuples in the fact table to understand the scaling behavior of Dwarf with increasing dimensionality.

Effect of Zipf Data. As for the first step, we have chosen $\theta = 0.95$ in order to generate a distribution which presents significant skew while still having less skew than the self-similar 80-20 distribution. The results are displayed in Figure 2(a). The results show that indeed Dwarf produces smaller indexes for less skewed distributions. Nevertheless, as the distribution has significant skew, expansion ratios are still increasing significantly with higher dimensionality. For instance, for $D = 30$, the Dwarf index uses 8.3 GB and exhibits a fact table expansion ratio of 689:1.

Coalesced Tuples. As for our second step, in later work [28] the inventors proposed an implementation-independent lower bound for the Dwarf index size counting the number of coalesced tuples. As [28] also provides results for Zipf data, this allows us to make a direct, implementation-independent comparison. This experiment corresponds to Figure 12 of [28]. Figure 4(a) shows results for very lightly skewed data ($\theta = 0.2$). The figure displays results for two different cardinalities $C = 1,000$ and $C = 10,000$. The results show that the number of coalesced tuples as counted in our implementation (NEW) matches the numbers reported by the inventors (ORIG) perfectly. The same holds for slightly higher skew factors ($\theta = 0.4$). This is displayed in Figure 4(b). In summary, we conclude that the implementations of ORIG and NEW report the same number of coalesced tuples. For completeness we will also display the number of coalesced tuples for other experiments of this paper including Figures 2(b), 3(b), 6(b), 7(b), and Table 10.

Let's look at the number of coalesced tuples in more detail. For instance in Figure 2(b) we observe that for a fact table size of 100,000 tuples and a 20-dimensional 80-20 distribution the number of coalesced tuples is already about 88.6 million tuples. This is by a factor 886 larger than the number of tuples in the fact table. Similar observations may be made for less skewed data, e.g., for the Zipfian ditribution and 20 dimensions we receive a factor of 102. Thus we conclude that the index growth of the Dwarf index is an intrinsic characteristic of the method and not due to a specific implementation.

General Behavior for Skewed Data. As for our third step, Figure 3(a) shows the results we have obtained when repeating the same experiment with only 10,000 tuples in the fact table. Note that the raw data size for $D = 10$ is displayed as <1 MB in the table as the fact table allocates 430 KB only. Again, we have obtained a similar scaling behavior with higher dimensionality for Dwarf indexes. Although the Dwarf index is always bigger than the fact table, the expansion ratios observed with a uniform distribution are not as dramatic as the ones observed with skewed distributions. At $D = 30$, uniform data produces an expansion ratio of 38:1, compared to a 275:1 ratio for Zipf and an impressive 5528:1 ratio for self-similar 80-20 data. We once more observe that the more skewed the distribution the higher the expansion ratio for Dwarf indexes. In addition, confirming the results of Figure 2(a), index creation times are proportional to index sizes.

Graphical Comparison. We have also reproduced the results for scaling Dwarfs as found in Figures 3 and 4 in [26]. The results of this experiment are shown in Figures 5(a), 5(b), and 5(c). Like in the original evaluation, the fact table always contained 250,000 tuples. Unlike the inventors, however, we have not compared Dwarf against Cubetrees, but rather against the fact table itself (see also Sections 6.1 and 6.2 for a comparison of Dwarf query performance with full scan performance under different organizations of the fact table). In addition, we scale Dwarf not only to 10 dimensions as in [26], but rather to 20 dimensions. This is because, as we have

[6]We remind the reader that times from the original paper are not directly comparable with times reported for our implementation due to hardware differences.

#Tuples	Fact Table Size [MB]	Uniform				80-20			
		Dwarf [MB]		Time [sec]		Dwarf [MB]		Time [sec]	
		ORIG	NEW	ORIG	NEW	ORIG	NEW	ORIG	NEW
100,000	4	62	53	27	8	72	94	31	16
200,000	8	133	113	58	19	159	195	69	35
400,000	17	287	239	127	43	351	400	156	70
600,000	25	451	372	202	64	553	608	250	109
800,000	34	622	509	289	90	762	818	357	146
1,000,000	42	798	651	387	114	975	1031	457	188

(a) Dwarf index sizes and creation times

#Tuples	Uniform #coalesced Tuples	80-20 #coalesced Tuples
100,000	0.7 E6	1.9 E6
200,000	1.5 E6	4.1 E6
400,000	3.4 E6	8.6 E6
600,000	5.3 E6	13.2 E6
800,000	7.4 E6	17.9 E6
1,000,000	9.6 E6	22.7 E6

(b) #coalesced tuples

Figure 6: Storage size and creation time requirements vs #Tuples. Repeats and complements Table 5 from [26]. Cardinalities for the dimensions are 30,000, 5,000, 5,000, 2,000, 1,000, 1,000, 100, 100, 100, and 10.

#Tuples	Fact Table Size [MB]	Uniform		80-20	
		Dwarf [MB]	Time [sec]	Dwarf [MB]	Time [sec]
100,000	4	55	8	152	26
200,000	8	130	21	314	54
400,000	17	326	53	653	114
600,000	25	566	88	1002	177
800,000	34	836	127	1358	246
1,000,000	42	1123	172	1720	312

(a) Dwarf index sizes and creation times

#Tuples	Uniform #coalesced Tuples	80-20 #coalesced Tuples
100,000	0.3 E6	2.3 E6
200,000	1.0 E6	4.7 E6
400,000	3.2 E6	9.9 E6
600,000	6.1 E6	15.3 E6
800,000	9.4 E6	21.0 E6
1,000,000	12.9 E6	26.8 E6

(b) #coalesced tuples

Figure 7: Storage size and creation time requirements vs #Tuples. Additional experiment complementing Table 5 from [26] with numbers having dimension cardinalities compatible with Table 4 from [26]. Cardinalities for all dimensions are 1,000.

seen in our previous results, the effect of higher dimensionality can be significant on Dwarf indexes.

Figures 5(a) and 5(c) show that Dwarf indexes exhibit modest expansion ratios when compared to the fact table up to 10 dimensions for all three distributions (Uniform, 80-20, and Zipf). Above 10 dimensions, however, the growth rate on index sizes depends on the distribution, being more acute for distributions with higher skew. For a uniform distribution, we obtain an expansion ratio of 54:1 at 20 dimensions. For Zipf and self-similar 80-20, the ratios are 253:1 at 20 dimensions and 473:1 at 17 dimensions, respectively. We could not scale further than 17 dimensions with a self-similar 80-20 distribution as that would exceed the maximum index size of 16 GB in our implementation. In addition, we confirm our observation that index creation times are proportional to index sizes, as may be observed in Figure 5(b).

In summary, our experiments show that for uniform data the Dwarf index sizes and construction times are in sync with the inventors. For skewed distributions, however, both index sizes and construction times are considerably larger than reported by the inventors. In addition, we observe that the size of a Dwarf index may be up to 5500 times larger than the fact table.

5.2 Scaling in the Number of Tuples

In this section we examine how Dwarf indexes perform when scaling the number of tuples in the fact table. We repeat the experiment presented in Table 5 of [26]. This means that we keep the dimensionality constant at 10 dimensions. In addition, we use the same cardinalities as used by the inventors, i.e., 30,000, 5,000, 5,000, 2,000, 1,000, 1,000, 100, 100, 100, and 10. Figure 6(a) shows the results.

Scaling Changing Cardinalities. Figure 6(a) shows results as reported by the inventors (ORIG) and our results (NEW). The results for uniform data show that the index sizes obtained by NEW are slightly smaller than the ones of ORIG. This is due to the fact that we are using a node representation that is slightly more efficient. In addition, index creation times of NEW are by a factor 2–4 better than ORIG. Again, this is not surprising as we are using newer hardware. However, the results for 80-20 data show that the index sizes of NEW are slightly bigger than ORIG. This is against our expectations as we would assume that a more effective node size should lead to smaller indexes independent of the distribution used. We do not have an explanation for these results. However, recall that for the experiments in Section 5.1 we have already observed larger index sizes for 80-20. In terms of index creation times we observe that the results of NEW for 80-20 are better than ORIG.

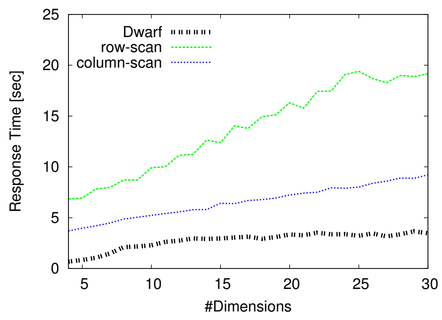
Scaling Keeping Cardinalities. Another interesting aspect of this experiment is that the inventors change at the same time the scaling dimension as well as the cardinalities of the dimensions. In more detail, in Section 5.1.1 of [26] the inventors scale the dimension while using a cardinality of 1000 for each dimension (repeated above in Section 5.1). Then, in the same section they perform a similar experiment scaling the number of tuples (Table 5 of [26]). However, at the same time while changing the scaling parameter the inventors also change the cardinalities of the dimensions. The differing cardinalities are reported in the last paragraph of Section 5.1.1 of [26], however, without giving any motivation for this.

Due to this observation we decided to provide an additional experiment that *does not* change the cardinalities but only the scaling parameter. It is displayed in Figure 7(a). The results show that if we keep the cardinalities at 1,000 each, Dwarf index sizes as well as index construction times are considerably larger. For instance, for uniform data and 1,000,000 tuples the Dwarf index has a size of 1,123 MB instead of 651 MB. For 80-20 data the Dwarf index has a size of 1,720 MB instead of 1,031 MB. The same increase may be observed for index construction times.

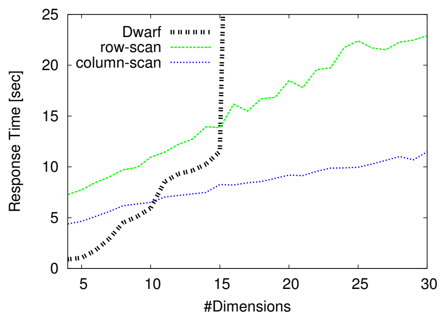
We conclude that the cardinalities of the dimensions have a strong impact on both Dwarf index sizes and construction times.

Index Expansion Ratios. Figure 6 also displays the sizes of the original fact tables. Again, this is done to give a better feeling on the relative space requirements of Dwarfs when compared to the original fact table. For the cardinalities used in Figure 6(a), we observe that Dwarf indexes are about 10–25 times larger than the fact table. For cardinalities equal to 1,000 in each dimension (see Figure 7(a)) we observe that the Dwarf indexes are even larger and may grow up to 41 times larger than the fact table. We argue that it is important to display such a huge expansion ratio in order to judge the costs of an indexing method.

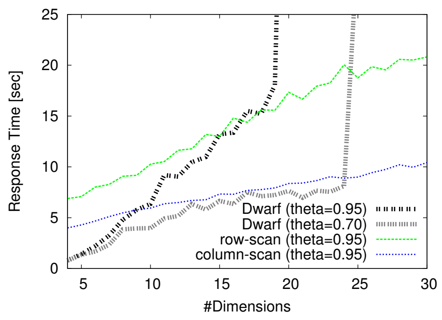
In summary, our experiments show that for uniform data our re-



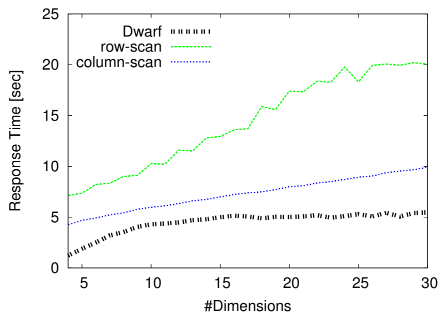
(a) Uniform data, query workload A



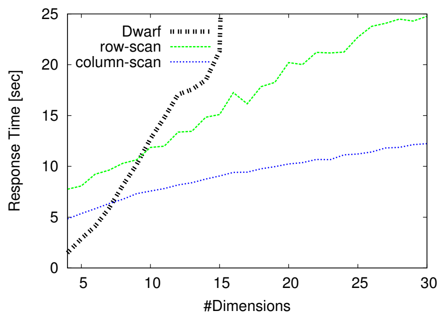
(b) 80-20 data, query workload A



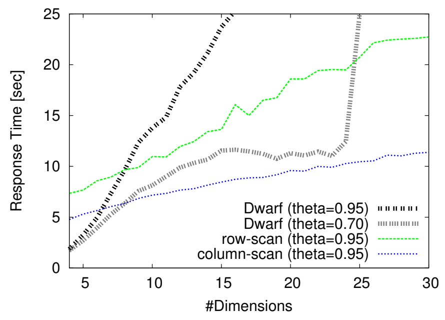
(c) Zipf data, query workload A



(d) Uniform data, query workload B



(e) 80-20 data, query workload B



(f) Zipf data, query workload B

Figure 8: Query performance on synthetic data: Dwarf versus row-scan and column-scan (complements Figures 5&6 from [26] by showing up to 30 dimensions and by comparing with full table scan methods)

sults are in sync with the inventors. For skewed data, however, we observe larger index sizes. The index sizes increase even more if we keep the original cardinality distribution.

5.3 Results with Real Data Sets

In this section we examine how Dwarf indexes perform on real data sets. We repeat the experiment presented in Table 6 of [26]. As mentioned above, only one of the real data sets used in ORIG could be reproduced. Table 5 shows the results.

Dataset	Fact Table Size [MB]	D	Dwarf [MB]		Time [sec]	
			ORIG	NEW	ORIG	NEW
`Weather-FULL`	39	9	n/a	212	n/a	35
`Weather-TRUNC`	39	9	n/a	294	n/a	48
`Forest`	24	10	594	524	350	74

Table 5: Storage and creation time for real datasets (repeats and complements see Table 6 in [26])

The results for `Forest` show a slightly smaller index size for NEW than ORIG. The Dwarf index is 22 times bigger than the fact table. The index construction time for NEW is by a factor five better than ORIG. For `Weather-FULL` and `Weather-TRUNC` we obtain index sizes of 212 MB and 294 MB and index construction times below a minute. The results are in sync with the results observed above for 10-dimensional data. It would, however, be interesting to try higher-dimensionality data sets as we have seen that the Dwarf is sensitive to high dimensional data. We leave this to future work.

6. QUERY PROCESSING EXPERIMENTS

In this section we examine the query performance of Dwarf indexes. In addition to the experiments shown by the inventors we also show comparisons to scans on a row- and a column-store.

6.1 Query Performance on Synthetic Data

In this section we evaluate the query performance of Dwarf indexes on synthetic data. This corresponds to the results shown in Figures 5&6 in [26]. Our results are shown in Figure 8. Similarly to the inventors we scale the dimensionality and show query performance for two different query workloads as specified by the inventors. The fact table size is 250,000 tuples. In contrast to the inventors we scale up to 30 dimensions and not only 10. Moreover, we do not only use uniform and 80-20 distributed data but also Zipf distributed data. The query workloads are summarized in Table 6.

		Probabilities			Range	
Workload	#Queries	P_{newQ}	P_{dim}	P_{pointQ}	Max	Min
A	1,000	0.34	0.4	0.2	20%	1
B	1,000	1.00	0.4	0.2	20%	1

Table 6: Workload Characteristics for "Dwarfs vs Full Table Scans" Query Experiment (see Table 7 in [26])

For these query workloads P_{newQ} describes the probability that a new query will not be related to the previous query. $1 - P_{newQ}$ is the probability that the query is a drill-down or a roll-up query. P_{dim} describes the probability that a single dimension will be selected to participate in a query. P_{pointQ} describes the probability that the entire query specifies only a point for each dimension. A range query covers at most 20% of the possible values and has at least one value (see Section 5.2.1 of [26] for details).

In contrast to the inventors we also show two additional aggregation methods: (1) `row-scan` which uses a full-table scan on the fact table to obtain tuples that qualify for the aggregation, and (2) `column-scan` which corresponds to a dimension-wise scan on a column-store. It uses a bitmap to mark tuples qualifying for a query. All three methods Dwarf, `row-scan`, and `column-scan` return the same result.

WL	Weather-FULL			Weather-TRUNC			FOREST			
	Dwarf NEW	row-scan	column-scan	Dwarf NEW	row-scan	column-scan	Dwarf ORIG	Dwarf NEW	row-scan	column-scan
A'	4.0	77.5	35.2	1.1	76.6	35.2	150	21.5	49.0	24.8
B'	3.1	74.5	34.2	1.0	74.4	34.3	176	14.8	47.1	21.8
C'	6.0	75.8	35.0	2.2	76.1	35.1	208	33.8	50.8	23.0
D'	2.0	76.0	30.3	0.6	77.6	30.3	217	11.1	49.3	19.4
E'	4.7	81.4	32.0	1.6	80.3	31.8	262	23.5	48.8	20.6

Table 7: Query performance for query workload A'–E' on three real data sets [times in sec]. Repeats and complements Tab. 9 of [26].

Effect of Uniform Data. Figures 8(a) and 8(d) show the results for uniform data for query workload A (resp. B). The results show that Dwarf performs pretty well: it only needs up to 3 seconds for the entire workload A and for workload B it takes up to 5 seconds to process 1000 queries. This is because workload B does not contain any drill-down or roll-up queries ($P_{newQ} = 1$), which slightly affect the query performance of Dwarf. Figures 8(a) and 8(d) also show `row-scan` which takes from 7 seconds for 4 dimensions up to 20 seconds for 30 dimensions. In contrast, `column-scan` only requires 4 seconds for 4 dimensions up to 9 seconds for 30 dimensions. In summary, the Dwarf index performs by a factor 1.5 to 3 better than the best scan method on uniform data.

Effect of Self-Similar Data. Figures 8(b) and 8(e) show the results for the same experiment for 80-20 self-similar data. Here, the performance characteristics of the different methods are quite different. The runtimes for `row-scan` are similar to the ones for uniform data. For `column-scan` the runtimes slightly increase to 4 seconds for 4 dimensions up to 11.5 seconds for 30 dimensions. In contrast, the runtimes for Dwarf increase dramatically: for 11 dimensions Dwarf is already slower than `column-scan`; for 15 dimensions it is comparable to `row-scan` and increases sharply for higher dimensions to multiples of the runtimes needed for `row-scan`. This increase is due to the fact that for more than 15 dimensions the Dwarf index built for the 16 MB size fact table does not fit into the 5 GB available main memory anymore and has to perform considerable I/O. Also note that this strong increase may not have been observed by the inventors as the corresponding Figure (Figure 6 in [26]) only scales up to 10 dimensions. Our index sizes for uniform data, which closely match the ones reported by the inventors, were always below 256 MB up to 11 dimensions. With these index sizes the Dwarf would have remained in main memory even with the hardware used in [26]. The increase in query response time, however, only happens for higher dimensions.

Effect of Zipf Data. To better understand these results we also show results for Zipf data (not shown by the inventors) which is less skewed than the self-similar data. We have run Dwarf with two different values of the θ parameter for the Zipf distribution: $\theta = 0.95$ (as used in other experiments) and $\theta = 0.7$ (less skewed). Figures 8(c) and 8(f) show the results for the same experiment using Zipf data. For `row-scan` and `column-scan` we see the same tendencies: runtimes are somewhere in-between the ones for uniform and 80-20 data. The change in runtime is minimal. Therefore, we present only the scan results for the more skewed distribution with $\theta = 0.95$. For Dwarf, however, we see a similar strong increase in runtimes as for self-similar data when $\theta = 0.95$. This increase is due to the fact that for 19 dimensions the Dwarf index does not fit into the 5GB available main memory anymore. For query workload A that increase does not seem to be as pronounced. For query workload B, however, that increase matches the one observed for 80-20 data. This means that for $\theta = 0.95$ `column-scan` will perform better than Dwarf for dimensions higher than 6 (see Figure 8(f)). For a less skewed Zipf distribution, with $\theta = 0.7$, the growth in the runtime of Dwarf is smoother up to 24 dimensions. Above that threshold, Dwarf performance again deteriorates due to insufficient main memory. Between 10 and 24 dimensions, the difference in performance between Dwarf and `column-scan` is not significant, with Dwarf performing slightly better on workload A and slightly worse on workload B. This means that for less skewed data Dwarf will only outperform `column-scan` for less than 10 dimensions for workload A and less than 7 dimensions for workload B.

6.2 Query Performance on Real Data

In this section we evaluate the query performance of Dwarf indexes on real data sets. This corresponds to the results shown in Table 9 of [26]. Our results are shown in Table 7.

Workload	#Queries	P_{newQ}	P_{dim}	P_{pointQ}	$Range_{max}$
A'	2,000	0.34	0.4	0.1	15%
B'	2,000	0.34	0.4	0.5	25%
C'	2,000	1.00	0.4	0.5	25%
D'	2,000	0.34	0.3	0.5	25%
E'	2,000	1.00	0.3	0.5	25%

Table 8: Workload Characteristics for "Dwarfs vs Full Table Scans" Query Experiment. Corresponds to Table 8 of [26].

Similarly to the inventors we use three different real data sets using five different query workloads. The query workloads are specified in Table 8. The features of the data sets are summarized in Table 3. As stated above (see Section 4.2) only one of the data sets `FOREST` used by the inventors was sufficiently specified such that it could be exactly reproduced. Again, in addition to the inventors we also show results for `row-scan` and `column-scan`. As before, we may not compare our query response times with the ones reported by the inventors for `FOREST` as we are using different hardware. However, we may compare the performance of the three different aggregation methods `Dwarf`, `row-scan`, and `column-scan`. For `Weather-FULL` Dwarf is consistently faster than `column-scan` by about a factor 10. For `Weather-TRUNC` this advantage is even more pronounced. The reason is that for the artificially created data set `Weather-TRUNC` the number of nodes and leaves touched to process a query is by a factor three smaller than for `Weather-FULL`. For `FOREST`, however, the performance of Dwarf is comparable to the one of `column-scan`. For workload A', B', and D', Dwarf performs better. For workload C' and E' `column-scan` performs better. This again supports the observation made by the inventors that Dwarf indexes profit from drill-down and roll-up queries, but do not perform as well on workloads of independent queries.

In summary, Dwarf seems to perform well for *some* real data sets. For other real data sets, however, the performance of Dwarf matches the one of a simple column scan on unindexed data. However, it is difficult to estimate in advance for which data sets a Dwarf will be advantageous over a column scan.

6.3 Coarse-Grained Dwarfs

In this section we examine how storage requirements, construction time, and query performance are affected by coarse-grained Dwarfs. As described in Section 2.3, the index size may be traded for query performance by setting the minimum *granularity* (G_{min}) parameter. Like that we obtain a coarse-grained Dwarf. This experiment repeats and complements Table 10 of [26]. Our results

		Uniform						80-20						Zipf ($\theta = 0.95$)		
	G_{min}	Space [MB]		Creation [sec]		Queries [sec]		Space [MB]		Creation [sec]		Queries [sec]		Space [MB]	Creation [sec]	Queries [sec]
		ORIG	NEW	ORIG	NEW	ORIG	NEW	ORIG	NEW	ORIG	NEW	ORIG	NEW			
`Dwarf`	1	490	418	202	64	154	9	482	242	218	43	199	24	433	70	29
	10	n/a	237	n/a	38	n/a	10	n/a	147	n/a	30	n/a	39	253	48	42
	50	n/a	168	n/a	26	n/a	11	n/a	105	n/a	21	n/a	59	164	32	67
	100	400	137	74	19	110	15	376	85	81	18	262	77	128	25	84
	500	n/a	82	n/a	10	n/a	90	n/a	59	n/a	13	n/a	162	75	16	184
	1,000	312	66	59	7	317	254	343	52	62	12	295	280	65	13	319
	5,000	166	66	29	7	408	254	288	50	53	11	1094	297	63	13	332
	20,000	151	66	25	7	476	256	160	49	30	11	1434	305	61	13	341
`row-scan`						n/a	222					n/a	251			237
`column-scan`						n/a	138					n/a	192			167

Table 9: Storage size, Creation Time, and Query Performance for synthetic data set for coarse-granular Dwarfs, row-scan, and column-scan. Repeats and complements Table 10 of [26].

are shown in Tables 9 and 10. We use the same synthetic dataset as described in [26], having 8 dimensions and 800,000 tuples. The cardinalities of the dimensions are: 1250, 625, 300, 150, 80, 40, 20 and 10. Similarly to the inventors we use a uniform as well as a self-similar 80-20 distribution. In addition, we use a Zipf distribution ($\theta = 0.95$). Moreover, we show additional values for G_{min} to better understand the trade-offs of this tuning knob.

Query Workload. The query workload for this experiment is again different from the ones in the previous experiments. However, we precisely follow the query workload description given by the inventors. The query workload contains 8,000 queries using rotated dimensions. The first 1,000 queries perform a range query in dimensions 1 to 3. In the remaining 5 dimensions (4 to 8) a point query is issued with probability 30%, otherwise ALL semantics are used. For the next 1,000 queries, the range queries are executed in dimensions 2 to 4. The remaining five dimensions (1 and 5 to 8) again contain point queries or ALL semantics as described above. Creating the next 1,000 queries continues in the same fashion. When the range query dimensions reach the last dimension 8, they wrap around. See the technical report [27] for details.

Increasing G_{min}. The results show that for increasing G_{min} the Dwarf index benefits in terms of index size and index creation time. The price that has to be paid, however, is increased query runtime.

For uniform data, we observe that index sizes of NEW drop from 418 MB to 66 MB whereas query response times rise from 9 sec to 256 sec. Note that the index sizes of ORIG and NEW again do not match. The results of NEW are in fact by a factor better which cannot be explained solely based on our more compact node representation. For 80-20 data and Zipf data we see similar effects for index sizes, which go down for increasing G_{min}. Once more, creation times are correlated with index sizes over all distributions. Similarly, for query response times, we observe that runtimes grow for increasing G_{min}. Another interesting observation is that most of the reduction in index size is already obtained for small values of G_{min}, e.g., $G_{min} = 10$. At these values query performance is still very close to the one of a fine-grained Dwarf for a uniform distribution. For skewed distributions, however, query response times rise much quicker even for small values of G_{min}.

Comparison with Scan-Methods. Table 9 also shows query execution times for both scan methods `row-scan` and `column-scan` (not shown by the inventors). For uniform data we observe that a Dwarf for $G_{min} = 100$ requires 15 seconds to process the query workload. For $G_{min} = 1,000$, however, the Dwarf requires 254 seconds. This is slower than both scan methods `row-scan` and `column-scan` which require 222 sec (resp. 138 sec). Therefore we observe that in fact a coarse-granular Dwarf is smaller than a fine-granular Dwarf, i.e., for $G_{min} \geq 1,000$ it was 'only' about three times bigger than the fact table which had 28.8 MB. However, at

G_{min}	Uniform	80-20	zipf ($\theta = 0.95$)
	#coalesced Tuples	#coalesced Tuples	#coalesced Tuples
1	8.7 E6	7.2 E6	11.5 E6
10	3.6 E6	3.0 E6	4.5 E6
50	1.4 E6	1.6 E6	1.9 E6
100	1.0 E6	1.2 E6	1.2 E6
500	0.8 E6	0.9 E6	0.8 E6
1,000	0.8 E6	0.8 E6	0.8 E6
5,000	0.8 E6	0.8 E6	0.8 E6
20,000	0.8 E6	0.8 E6	0.8 E6

Table 10: Storage size, Creation Time, and Query Performance for synthetic data set for coarse-granular Dwarfs. Complements Table 10 of [26] showing number of coalesced tuples.

the same time the query response times on such a coarse-granular Dwarf are higher than the ones for both scan-methods. This obviates the need to use a Dwarf index. Similar observations hold for 80-20 and uniform data. Only for a small G_{min} it makes sense to keep a coarse-granular Dwarf. For higher values of G_{min} the scan methods perform better.

In summary, coarse-granular Dwarfs may have lower index sizes and creation times at the cost of increases in query response times. The G_{min} tuning knob has to be tuned carefully to avoid response times worse than the ones obtained by simple scan methods.

7. RELATED WORK

As mentioned in the Introduction, much work has been done in the area of OLAP. Here we focus on work related to Dwarf indexes.

Dwarf. The Dwarf index was proposed on SIGMOD 2002 [26, 27] and also patented [24]. We have already detailed its main ideas in Section 2. An in-depth discussion of differences among performance results reported by the inventors and our implementation is provided in Sections 5 and 6. Here, we want to point out a few additional observations. In [26] a point that is particularly misleading is that the inventors compare the size of Dwarf indexes to the size of the fully materialized cube. For instance, in Table 4 of [26] the size of the cube is counted using a binary storage footprint (BSF) which is equal to storing all possible views of the cube in unindexed relations. Obviously, the size of BSF is exponential in the number of dimensions. Throughout their work the inventors compare Dwarf indexes to BSF but never to the size of the original fact table. Thus, they claim compression ratios of up to 1:400,000. We think that this kind of comparison is highly unrealistic as the fully materialized cube would never be computed and, in fact, *was never* materialized by the inventors. On the contrary, the original fact table was simply *expanded* to a Dwarf index. Let's pick one example, e.g., from Table 4 in [26] we pick a 25-dimensional cube that when fully materialized would require a claimed 173 TB of storage. We may calculate the size of the original fact table as $4 \cdot 100,000 \cdot (d+1) = 9.9$ *Megabytes*. Using [26] we can then compute the size ratio of a Dwarf compared to the original fact table as 52 times bigger (Uniform) and 181 times bigger (80-20) than the original fact table. We argue that an index size two orders of magnitude bigger than the original data set is an important number to understand the performance characteristics of a method.

Dwarf Extensions. The idea of Dwarf indexes was extended in more recent work. In [25] the idea was extended to also support hierarchies which are common in OLAP. However, that paper did not change the underlying principal structure and algorithms of the Dwarf. In [28] a complexity analysis for Dwarfs was presented. It makes the same comparisons to fully materialized cubes as discussed above. The main result of that work is that the expected size and computation time complexity is polynomial, i.e, for a uniform cube the expected size and computation time complexity is shown to be $O(D \cdot |F|^{1+1/log_D C}) = O(D \cdot |F| \cdot |F|^{1/log_D C})$ where $|F|$ is the number of rows in the fact table, D the number of dimensions, and C the cardinality in each dimension. Although Dwarf index sizes grow polynomially for uniform data, even a modest exponent for the polynomial may lead to very large index sizes in practice, several times larger than the original data set. Therefore we agree with the inventors that a polynomial complexity is a nice result [28]. However, we think that this theoretical result is of limited practical value — as evidenced by our experiments.

8. CONCLUSIONS

This paper has taken a look into the rearview mirror to better understand a promising index structure for multi-dimensional OLAP: Dwarf indexes. Our experiments for uniform data indicate that the behavior of Dwarf indexes seem to be in sync with the observations of the inventors. For skewed data, however, our results show that the size of a Dwarf index may be orders of magnitude larger than reported by the inventors.

In fact, Dwarfs seem to be very sensitive to skewed data which makes these structures hard to control in practical situations. In particular, Dwarf indexes do not scale well for more than 10 dimensions on these data sets. Moreover, Dwarf indexes require considerable disk (or main memory) space ranging from 10 up to 5,500 times larger than the original fact table size. This makes the technique hard to apply to large scenarios – unless one is willing to pay the price for huge storage and index construction times.

Furthermore, given the current hardware trends that allow system architects to keep the entire data set in main memory, we conclude that for many realistic scenarios (at least the ones presented by the inventors of Dwarf) it makes sense to keep the raw fact table entirely in main memory. Consequently, we showed experiments comparing Dwarf indexes to simple main memory table scans on both a row-store and a column-store which are widely used in ROLAP and VROLAP implementations. Our results indicate that in several situations the cost of a Dwarf index will not be competitive. However, there are some situations where Dwarf outperforms even a column scan, i.e., for uniform data one may gain a factor two over a column-scan. The same holds for some of the real data sets we tried where one may gain an even higher factor. However these gains seem to be unpredictable. On the downside, the price to pay for a Dwarf index are: considerable storage space, long index construction times as well as possibly unpredictable increase in index rebuild time. Therefore, Dwarf does match very well the behavior of a typical MOLAP method.

As a general guideline we recommend to use Dwarf indexes only for cases where the query load is so demanding that it cannot be satisfied by a column-store. For instance for uniform data and 10 dimensions we observed that a throughput of 200 queries per second may be served by a simple column-scan. Only if this query throughput is not sufficient or cannot be coped with by scaling out (i.e., adding more computing nodes), Dwarf should be considered. In addition, the data sets indexed by Dwarf should only be lightly skewed and should not have too many dimensions.

Acknowledgements. We would like to thank Cristian Duda for comments on earlier versions of this paper. Work partially supported by SNF under contract 200021-112115.

9. REFERENCES

[1] D. S. Batory. On Searching Transposed Files. *ACM TODS*, 4(4):531–544, 1979.

[2] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBES. In *SIGMOD*, 1999.

[3] J. A. Blackard. The Forest CoverType dataset. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype.

[4] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.D. Thesis, May 2002.

[5] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[6] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[7] J.-P. Dittrich, D. Kossmann, and A. Kreutz. Bridging the Gap between OLAP and SQL. In *VLDB*, 2005.

[8] C. D. French. "One Size Fits All" Database Architectures Do Not Work For DSS. In *SIGMOD*, 1995.

[9] C. D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *ICDE*, 1997.

[10] J. Gray et al. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*, 1994.

[11] J. Gray et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *ICDE*, 1996.

[12] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. In *ICDE*, 1997.

[13] C. Hahn, S. Warren, and J. London. Edited Synoptic Cloud Reports from Ships and Land Stations over the Globe. http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html.

[14] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[15] Essbase. `download.oracle.com/docs/cd/e10530_01/doc/index.htm`.

[16] Kx Systems. `http://www.kx.com`.

[17] Microsoft Analysis Services. `www.microsoft.com/sql/technologies/analysis/`.

[18] L. V. S. Lakshmanan, J. Pei, and Y. Zhao. QC-trees: An Efficient Summary Structure for Semantic OLAP. In *SIGMOD*, 2003.

[19] H. Li, H. Huang, and S. Liu. PMC: Select Materialized Cells in Data Cubes. In *DaWaK*, 2005.

[20] X. Longgang and F. Yucai. Fast Computation of Iceberg Dwarf. In *SSDBM*, 2004.

[21] Mersenne Twister. `http://www.cs.umd.edu/users/seanl/gp`.

[22] Model 204. `http://www.cca-int.com/prodinfo/m204.html`.

[23] K. A. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. In *VLDB*, 1997.

[24] N. Roussopoulos, J. Sismanis, and A. Deligiannakis. Dwarf cube architecture for reducing storage sizes of multidimensional data. US Patent 7,133,876, 2002.

[25] Y. Sismanis, A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical dwarfs for the rollup cube. In *ACM DOLAP*, 2003.

[26] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. In *SIGMOD*, 2002.

[27] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. CS-TR 4342, U of Maryland, 2002.

[28] Y. Sismanis and N. Roussopoulos. The Polynomial Complexity of Fully Materialized Coalesced Cubes. In *VLDB*, 2004.

[29] A. R. Thomas Legler, Wolfgang Lehner. Data Mining with the SAP Netweaver BI Accelerator. In *VLDB*, 2006.

[30] Vertica. `http://www.vertica.com`.

[31] J. Widom. Research Problems in Data Warehousing. In *CIKM*, 1995.

[32] Wintercorp 2005 Top Ten Program. `http://www.wintercorp.com/vldb/2005_topten_survey/`.