BDCC: Exploiting Fine-Grained Persistent Memories for OLAP

Peter Boncz
NVRAM

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

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

➔ What about OLAP?

Should we re-think warehouse storage for low-latency persistent memories?
BDCC: Bitwise Dimensional Co-Clustering

Stephan Baumann · Peter Boncz · Kai-Uwe Sattler

Received: date / Accepted: date

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

1 Introduction

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

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

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

_bdcc_ column ordering ➔ works in column stores
Bitwise Interleaving = Z-Ordering

space filling curve

Computationally cheaper than eg Hilbert Curve

Almost as nice properties
BDCC - Data Order

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

- **Multi-dimensional** indexing
  - *table* indexing: not multi-media (audio,image) indexing here 😊
  - limited amount of dimensions (up to 5..7)

- **Multi-table** clustering (co-clustering)
  - indexing on dimensions from *other* tables..
  - ..reachable over foreign-key relationships
  - and exploiting common indexing dimensions among tables in operators

- Grouping into **MILLIONS** of very small groups
  - scattered access patterns ➔ **Flash** IO friendly!
  - clustering: because millions not possible with partitioning

- **Column-store** optimized
BDCC - The Idea

How does this help:
• selection?
• order by?
• Aggregation?
• FK join?
What BDCC gives you

Accelerates

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

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

- **No storage overhead**: every tuple stored once
- Bulk **update-friendly**
- Easy to **parallelize** query processing
Two Stages of the Project

- **Bitwise Dimensional Co-Clustering (BDCC)**
  - I/O level clustering and indexing
  - Query processing via PartitionSplit, PartitionRestart
    
    published in VLDBJ 2016

- **Deep Dimensional Co-Clustering (DDC)**
  - additional I/O block clustering
  - Query processing via DDC-Recluster()
    
    unpublished yet.. WIP
BDCC Structures

- **BDCC dimension**
  - mapping to consecutive integers
  - balancing through histograms and Hu-Tucker

- **BDCC table**
  - re-ordered primary copy
  - additional `_bdcc_` order attribute

- **BDCC count table**
  - summary table (`_bdcc_`, `_count_`)
  - cluster access index
BDCC Structures
BDCC Structures

BDCC Structures

“Dimension Use” ➔
“Dimension Use” ➔
“Dimension Use” ➔
BDCC Structures

BDCC Structures
Example

“count total ordered items from Germany per day and supplier”

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

Relational Algebra Plan
**Scans a BDCC table**

**In any desired dimension order**

Here:
1. orderdate
2. customer nation
3. supplier nation

**At a desired granularity using bitmasks**

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

**Pushes down selections:**

- [0,7] = all
- [0,3] = all
- [5,5] = germany
BDCC-scan

- extracts _bdcc_ bits ➔ _gid_ column
  \[d_3s_3c_3d_2s_2c_2d_1s_1c_1 \rightarrow d_3d_2d_1c_3c_2s_3s_2s_1\]
- delivers tuples ordered on _gid_
- performs selection pushdown ([lo-hi])

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

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

• **Partition-wise** operator execution
  – hash based join, grouping/aggregation
  – better cache utilization

• **Sandwich Operators** ➔ PartitionSplit & PartitionRestart
  – sideways information passing: PartitionRestart.cross-partition? (gid change)
    ➔ HashAggr/Join.flush() & PartitionSplit.next-partition()
BDCC - Performance
Sandwich Operators

- Micro-Benchmarks (TPC-H SF10, LINEITEM-ORDERS)
Relational Algebra Plan

Selection Pushdown + Dimension Join Elimination
Relational Algebra Plan

Selection Pushdown + Dimension Join Elimination
Co-Clustering Close-up

Co-Clustering Close-up
Common Path = Co-Clustering
Common Dimension = Accelerated Join
BDCC: All FK Joins Accelerated!
BDCC - Schema Design

- **Semi-automatic**
  - Input: CREATE INDEX() and FOREIGN KEY()

- **Schema traversal along foreign key paths**
  - propagation of „Index“ dimensions
  - weighted according to FK paths

- **automatic** creation of dimensions and tables
  - round robin bit interleaving
  - clustering depth based on column densities, typically 32KB (SSD) and 512KB (HDD) blocks
BDCC - Optimizer

• IDU: Interesting Dimension Uses
  • all dimensions determined by join, sort or aggregation attribute

• IDO: Interesting Dimension Orders
  • all dimension order permutations of each IDU

• MDO: Maximal Dimension Orders
  • Pruning of dominated sort orders of IDOs

• MDOs represent „interesting orders“ for enumeration
BDCC Performance

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

much better power scores with much less memory
BDCC Performance

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

much better power scores with much less memory
BDCC - Updates

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

- TPC-H SF100 update set

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

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

Idea 1: modified hash function

Problem: scattered access to all 4 quadrants -> bad cache utilization

Solution

Idea 2: recluster()
DDC Performance

**time=180,308,802,493 (8.58%)**
**hiMem=280,080,525 (10.56%)**

**time=1,289,344,872,689 (61.32%)**
**hiMem=2,183,884,150 (82.37%)**

**time=57,242,284,283 (2.23%)**
**hiMem=21,478,949,384 (98.40%)**

**time=371,357,339,842 (14.49%)**
**hiMem=10,842,073 (0.05%)**
Conclusion

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

Dimension = set of bins

- Range-Binning of a domain
- Histogram-based approach
  - Needs frequency information
Assigning Bin Numbers: Naïve Way

- Skew/frequent values (single-value bins)

<table>
<thead>
<tr>
<th>value</th>
<th>frequency</th>
<th>code</th>
<th>c2</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(null)</td>
<td>.70</td>
<td>000</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>Polytech</td>
<td>.15</td>
<td>001</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>Bachelor</td>
<td>.08</td>
<td>010</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>Master</td>
<td>.06</td>
<td>011</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>PhD</td>
<td>.01</td>
<td>100</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
Hu-Tucker Binning

- Frequency-based Bin Number Assignment

<table>
<thead>
<tr>
<th>value</th>
<th>frequency</th>
<th>code</th>
<th>c3</th>
<th>c2</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(null)</td>
<td>.70</td>
<td>0000</td>
<td>00</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>Polytech</td>
<td>.15</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bachelor</td>
<td>.08</td>
<td>1100</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Master</td>
<td>.06</td>
<td>1110</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PhD</td>
<td>.01</td>
<td>1111</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hu-Tucker = Order Respecting Huffman Coding
Hu Tucker Dimension Binning

but why is this relevant?
Variety in Data Density of Columns

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

Factor 120 difference

What is the optimal BDC bin size?
- Depends on disk block size
- Depends on column density

What to do if a query accesses multiple columns of very different densities?
Granularity Tuning in BDCC

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

2. Is an issue during **query execution**
   - Table is clustered at some granularity
   - Given a **set of columns** to scan:
     at what granularity to scan the table?
Z-Ordering for Column Stores

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

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

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

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