This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

Fusion OLAP: Fusing the Pros of MOLAP and ROLAP Together for In-memory OLAP

Yansong Zhang, Yu Zhang, Shan Wang and Jiaheng Lu

Abstract— OLAP models can be categorized with two types: MOLAP (multidimensional OLAP) and ROLAP (relational OLAP). In particular, MOLAP is efficient in multidimensional computing at the cost of cube maintenance, while ROLAP reduces the data storage size at the cost of expensive multidimensional join operations. In this paper, we propose a novel Fusion OLAP model to fuse the multidimensional computing model and relational storage model together to make the best aspects of both MOLAP and ROLAP worlds. This is achieved by mapping the relation tables into virtual multidimensional model and binding the multidimensional operations into a set of vector indexes to enable multidimensional computing on relation tables. The Fusion OLAP model can be integrated into the state-of-the-art in-memory databases with additional surrogate key indexes and vector indexes. We compared the Fusion OLAP implementations with three leading analytical in-memory databases. Our comprehensive experimental results show that Fusion OLAP implementation can achieve up to 35%, 365% and 169% performance improvements based on the Hyper, Vectorwise and MonetDB databases respectively, for the Star Schema Benchmark (SSB) with scale factor 100.

Index Terms—MOLAP, ROLAP, vector index, surrogate key index, vector referencing

1 Introduction

In recent years, the real-time OLAP has changed the requirements of traditional data warehouses. The sophisticated techniques in traditional OLAP domain such as indexed view, aggregation table, and cube materialization are not adaptive for the "*Velocity*" requirement of Big Data. Therefore, the low latency analytical data processing on raw data turns to be the mainstream technique.

The emerging approach for the real-time OLAP is to employ in-memory databases as real-time OLAP engines. For example, MonetDB [2], Vectorwise [3], SAP HANA [4], Hyper [5], MemSQL [6] are the most representative in-memory databases, and the traditional database vendors also push forward the new in-memory database products including Oracle Database In-Memory [7], IBM DB2 BLU Acceleration [8], SQL Server in-memory Columnstore [9]. The SQL Server 2016 Analysis Service adds InMemory as new storage model to the traditional OLAP models. For In-memory OLAP, the performance is the most critical issue because OLAP queries are on-the-fly executed with billions of tuples within seconds. Therefore, the in-memory OLAP should keep improving performance for the increasing data volume and real-time analysis requirements. On the other hand, the in-memory OLAP should upgrade from in-memory computing to hybrid coprocessor computing to achieve the massive parallel computing power from the emerging hardware accelerators like GPU, Phi, FPGA etc. The algorithms of in-memory databases employ CPU architecture centric designs, and the hardware-conscious algorithms deeply optimize the cache and pipeline mechanisms, but the same optimizations are not adaptive to coprocessors comprised of many computing cores and simple control circuits with different programming models. Moreover, the main stream coprocessors such as GPU, Phi, FPGA etc. have different hardware features. Therefore, the OLAP model and algorithm designs should be simple enough to be adaptive for different types of coprocessors.

Since OLAP queries join big fact tables with multiple dimension tables for aggregation, the join performance dominates the whole OLAP performance. In the academic community, there are a plethora of works on high performance and efficient joins, such as radix partitioned hash join [10], vectorized processing [11], JIT (just-in-time) compilation [12], hardware-conscious and hardware-oblivious hash join [13], etc. Furthermore, the developments of coprocessors such as GPGPU, Xeon Phi, and FPGA encourage the hardware acceleration techniques [14-17]. Note that join optimizations involve complex techniques [22][24], which make implementations on coprocessors difficult to be optimized, and the algorithm complexity and platform heterogeneity prevent in-memory databases from smoothly upgrading to the emerging heterogeneous computing platforms. The complexity of equi-join operation lies in that one tuple is unware of the exact position of the matched tuple, and thus the complex data structure and algorithms have to be invented to boost the efficiency of key probing. Nevertheless, such complexity could be significantly re-

- *Yansong Zhang is with the DEKE Lab, Renmin University of China, School of Information in Renmin University, and the National Survey Research Centre at Renmin University of China, Beijing 100872, China. E-mail: zhangys_ruc@ hotmail.com.*
- *Yu Zhang is with the National Satellite Meteorological Centre, Beijing 100081, China. E-mail: yuzhang@cma.gov.cn.*
- *Shan Wang is with the DEKE Lab, Renmin University of China, Beijing 100872, China. E-mail: swang@ruc.edu.cn.*
- *Jiaheng Lu is with Department of Computer Science, University of Helsinki, Finland. E-mail: jiahenglu@gmail.com*

*Please note that all acknowledgments should be placed at the end of the paper, before the bibliography(**note that corresponding authorship is not noted in affiliation box, but in acknowledgment section**).*

xxxx-xxxx/0x/$xx.00 © 200x IEEE Published by the IEEE Computer Society

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

duced with the multidimensional model perspective, because the multidimensional model maps each fact data to the exact position on each dimension. Therefore, the simple vector (acting as dimension) and positional addressing can take the place of hash table and hash probing for equijoin in OLAP, and the simple data structure may speedup the processing on coprocessors platforms.

In this paper, we present the *Fusion OLAP model*, in which we combine the relational storage model and multidimensional computing model together to enable multidimensional cube operation on relational storage without materialized cube or costly relational joins. In a nutshell, the salient feature of the Fusion OLAP model is to seamlessly integrate the multidimensional computation feature of MOLAP with the relational storage of ROLAP as a Multidimension-Relational OLAP model (MROLAP), by running multidimensional access operations (e.g. dicing, slicing) (rather than the traditional relational queries) on relational data. This feature not only enables Fusion OLAP to outperform state-of-the-art representative analytical in-memory databases but also to be adaptive for many core coprocessors.

To implement Fusion OLAP model, we develop a set of indexes to span multidimensional OLAP model and relational model. We use vector indexes as mappers between the multidimensional model and the relational model. The dimension tables are mapped to dimension vector indexes, and the foreign key columns in fact table are mapped to multidimensional index. We use a fact vector index to map relational fact table to virtual multidimensional cube. With these vector indexes, we can simplify OLAP operations through efficient multidimensional computing and vector index oriented aggregation.

To evaluate Fusion OLAP, we divide OLAP operations into three independent phases: (1) The first phase is to generate dimension vector indexes to construct the virtual cube; and (2) the second phase is to compute the fact vector index to enable multidimensional retrieval, and (3) the last phase is to retrieval fact data with vector index for aggregation. The first and third phases can be implemented by existing in-memory databases with SQL statements, and the second phase is implemented with independent module on multicore CPU, Xeon Phi and GPGPU platforms. We use the state-of-the-art Hyper, Vectorwise, and MonetDB databases as testbeds to evaluate how these high performance in-memory databases improve performance with a light-weighted vector index mechanism and an external multidimensional computing module. For the Star Schema Benchmark (SSB) with scale factor 100, the Fusion OLAP implementation can achieve up to 35%, 365% and 169% improvements upon Hyper, Vectorwise and MonetDB databases, respectively.

Therefore, the main contributions of this paper are summarized as follows:

1) We present the Fusion OLAP model for real-time OLAP that simplifies the OLAP model with higher performance. The Fusion OLAP model is adaptive to the emerging heterogeneous architecture with coprocessors like Xeon Phi and GPGPU.
2) We present the vector index as Fusion OLAP implementation mechanism for in-memory databases. Vector index is a computation-oriented index and shares fixed size columns for various queries. The *vector referencing* and vector index oriented aggregation achieve both better simplicity and efficiency than traditional relational operations.
3) We implemented the simulated Fusion OLAP model with three leading analytical in-memory databases. By accelerating vector index computing with coprocessors, Fusion OLAP model can achieve the average 150% performance improvement. Moreover, the experimental results show that the loose coupling framework accelerates OLAP performance with negligible side-effects on database systems.

Organization: The rest of paper is organized as follows. We describe the Fusion OLAP model and the major characteristics in Section 2. Section 3 discusses how to fuse multidimensional operation with relational database. Section 4 discusses the implementation of Fusion OLAP model on relational databases. In Section 5, we evaluate the Fusion OLAP with Hyper, Vectorwise and MonetDB databases. Related work is discussed in Section 6. Finally, we conclude this paper in Section 7.

2 Preliminaries

Multidimensional model is the fundamental data model of data warehouse, where the data are organized by dimensions which describe the diverse angles for comprehensive analysis and exploration. Conceptually, OLAP performs operations, e.g. rollup, drilldown, slicing, dicing and pivot, on multidimensional dataset. In this section, we briefly discuss three OLAP models (i.e. MOLAP, ROLAP and HOLAP) and the virtual cube mechanism to efficiently answer OLAP queries.

2.1 OLAP models

The major OLAP models include MOLAP (multidimensional OLAP), ROLAP (relational OLAP) and HOLAP (hybrid OLAP).

(1) MOLAP model stores data with multidimensional dataset organized as data cube, and the OLAP operation is multidimensional retrieval in data cube. The major disadvantage of multidimensional model is the high overhead of storage and precalculating. Multidimensional array model is not efficient for the sparse data. Further, when the dimensions are changed, the big data cube needs to be reconstructed which is not adaptive for real-time OLAP.

(2) ROLAP model employs relational database as OLAP

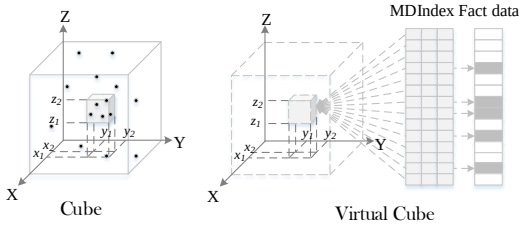


Fig. 1. Virtual cube.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

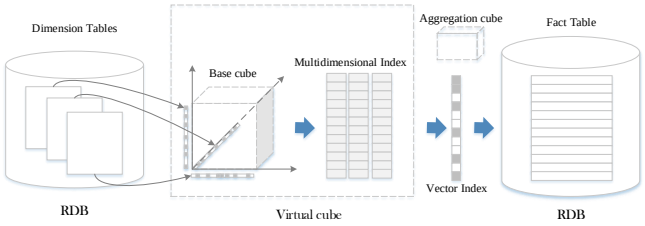


Fig. 2. Fusion OLAP framework.

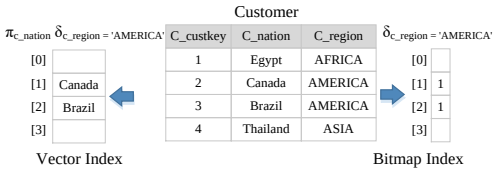


Fig. 3. Dimension Mapping.

engine, and the multidimensional operation is transformed as SQL queries. The data cube is materialized as aggregation tables, and the multidimensional retrieval is implemented as table scan. For ad-hoc OLAP queries, the relational join performance dominates the ROLAP performance. The essential difference between MOLAP and ROLAP model is multidimensional addressing mechanism. The MOLAP model is designed with double mapping between dimensions and fact data by organizing fact data with the multidimensional array (even for NULL cells), while ROLAP model simplifies the dimensional addressing mechanism with primary key and foreign key constraints.

(3) Finally, HOLAP model is a tradeoff between MOLAP and ROLAP models, in which the detailed data are stored as relational tables and frequently accessed aggregate tables are stored in multidimensional arrays.

2.2 Virtual Cube mechanism

Virtual cubes are based on multidimensional data model, where the dimensions are used as coordinate axises, and the fact data are labelled with multidimensional coordinate values instead of allocating real space in multidimensional array.

Figure 1 gives an illustration of cube and virtual cube. A multidimensional query mq defines the dataset of a subcube inside the cube, and the query is executed as multidimensional addressing in the cube according to coordinate values of dimensions.

$mq=\{A[x][y][z] \mid x \in [x_1,x_2] \wedge y \in [y_1,y_2] \wedge z \in [z_1,z_2]\}$, where A denotes a multidimensional array.

For virtual cube, the multidimensional query mq defines a dimension addressing and relational operation.

$mq=\{t[F] \mid t[X] \rightarrowtail [x_1,x_2] \wedge t[Y] \rightarrowtail [y_1,y_2] \wedge t[Z] \rightarrowtail [z_1,z_2]\}$, where "$\rightarrowtail$" denotes dimensional addressing operation from multidimensional index to the dimension.

The dimensions and fact data can be stored as relational tables for efficiency, and the virtual cube acts as a multidimensional filter for multidimensional index in ROLAP.

3 Fusion OLAP Model

In this section, we present Fusion OLAP model, which combines high performance multidimensional computing model in MOLAP and efficient relational storage model in ROLAP.

3.1 Fusion OLAP framework

Fusion OLAP model inherits the virtual cube mechanism, where the data are organized with logical multidimensional storage and labelled with multidimensional index as relational store, and the dimensions are used as coordinate axises for the cube. With the column-store, the fact data are divided into two parts, multidimensional index columns and fact columns.

Figure 2 shows the Fusion OLAP framework. The dimensions and fact data are stored as relational tables, the switcher between ROLAP and MOLAP is the vector index, which is an extension of bitmap index with specified bits to store more information. For dimension tables, each dimension table uses one vector index. The vector indexes are used as dimensions to map data in virtual multidimensional space, and the results of selection and projection operations on dimension table are mapped to the vector index to enable the dimension vector index to be dimension filters. All the the dimension vector indexes generated by queries construct the multidimensional filter of a virtual subcube space. Multidimensional index columns are mapped to virtual cubes as multidimensional addressing operation, and the query results are mapped to the fact vector index to retrieve relational fact data for aggregation. The fact vector index acts as a bitmap index on relational fact data. Hence, the Fusion OLAP framework comprises with two components, (i) the relational store for dimension and fact data, and (ii) the multidimensional computing.

3.2 Fusion OLAP operations

We first define the basic Fusion OLAP operations and will discuss the implementations in Section 4.

3.2.1 Dimension Mapping

A dimension mapping operation is to map each dimensional tuple to distinct vector index cell. The vector index plays the role of dimension like MOLAP, the vector offset address represents dimension coordinate value.

The dimension mapping operation is defined as:

$R \searrow VI=\{VI[x] \mid VI[x]=DimMap(\ \pi_p\ \delta_{key,s} t[R])\}$, where $\searrow$ operator denotes mapping dimension table R to vector index VI, $DimMap$ function processes each dimensional tuple $t[R]$ as single value by select (s) and projection (p) clauses on dimension table and then writes the value to unique vector index cell according to dimension key (key).

Figure 3 gives the examples of dimension mapping. The key column *c_custkey* is mapped to vector index offset address. When query projects *c_nation* under the condition *c_region*='AMERICA', the dimension is mapped to a vector index with filtered *c_nation* values. When the query only has predicate expressions, the vector index is a bitmap index.

3.2.2 Cube Aggregating

The multidimensional dataset is comprised of dimensions

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

and fact data, and the dimension comprises with hierarchies of different analytical angles. With the definitions of dimensions and hierarchies, the OLAP query can be normalized as aggregation with specified dimensional or hierarchical attributes, the aggregation can be modeled as subcube which is also named as cuboid.

It is efficient to retrieve subcube for OLAP queries, but the overhead of creating and maintaining the materialized data cube is very high. For a real-time OLAP requirement, we define the cube aggregating operation as creating aggregate cube and mapping it to dimensions.

$C \rightarrowtail VI=\{VI[x] \mid VI[x]=CubeMap(d[C])\}$, where *CubeMap* function denotes mapping from aggregating cube(C) to vector index for specified dimension (d).

When each vector index includes one aggregating cube dimension, the vector indexes are directly refreshed with aggregating cube index as shown in Figure 4(a). When one vector index includes multiple aggregating cube dimensions like Figure 4 (b), *part* dimension vector index has *color* and *size* aggregating dimensions, the composite attributes are normalized as linear address of the 2-dimension slice (*color* and *size*).

3.2.3 Multidimensional filtering

Dimension mapping operation defines a virtual cube for base data, the cube aggregating operation further defines a virtual or physical cube for aggregation by mapping aggregating cube to dimension vector indexes. Therefore, a fact data F logically locates its position by multidimensional index (e.g., (x,y,z)), each index value is mapped to position in dimension vector index to verify whether the current fact data satisfies the query condition. If the fact data satisfies all the conditions, the aggregating cube address can be used as aggregating index for fact data.

We define this process as multidimensional filtering.

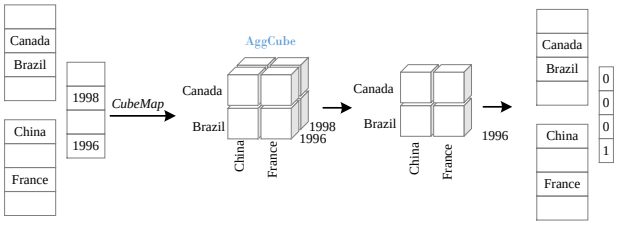
$MD \rightrightarrows \text{Dim\_}VI=$

$\{VI[x]=getAddress\ (MD_1[x] \rightarrowtail VI_1,\ MD_2[x] \rightarrowtail VI_2,\ \ldots,\ MD_n[x] \rightarrowtail VI_n) \mid MD_1[x] \rightarrowtail VI_1 \wedge MD_2[x] \rightarrowtail VI_2 \wedge \cdots \wedge MD_n[x] \rightarrowtail VI_n\}$, where MD denotes multidimensional index columns, *Dim_VI* denotes dimension vector indexes, *getAddress* function translates the multidimensional address of aggregating cube to single value address, "$\rightrightarrows$" denotes the operation of multidimensional filtering from multidimensional index to dimension vector indexes, and the result set is another vector index which is used as fact vector index to identify which fact data should be accessed and where the fact data should be aggregated in aggregating cube.

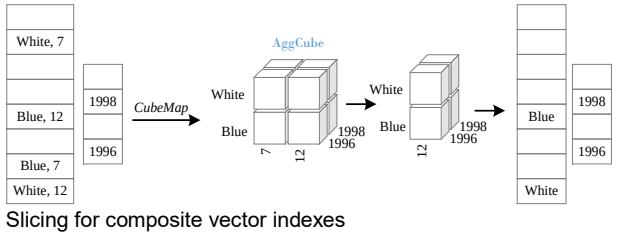
The multidimensional filtering operation is described in Figure 2, which is an inverted multidimensional addressing operation. The aggregating cube and query clauses are decomposed to dimension vector indexes, the multidimensional index columns map the multidimensional address to dimension vector indexes as an inverted multidimensional addressing operation, and the vector index oriented fact data retrieval acts as multidimensional accessing.

3.2.4 Slicing

In Fusion OLAP model, slicing is a reduction operation on aggregating cube, which changes the vector index. Figure 5(a) illustrates the slicing operation on vector indexes. For slicing condition "*year=1996*", the *date* dimension vector index is transformed from vector index to bitmap to identify dimensional tuple position for condition "*year*=1996", and the bitmap is only used for filtering multidimensional index. For vector index with composite attribute like Figure 5(b), the slicing condition "*size*=12" filters the vector index and removes *size* attribute from the original vector index.

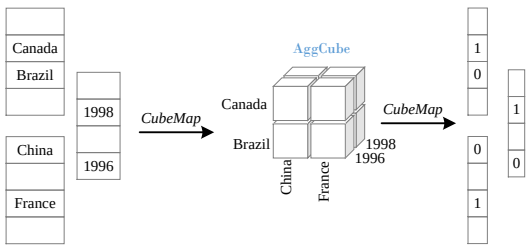


(a) Slicing for vector indexes

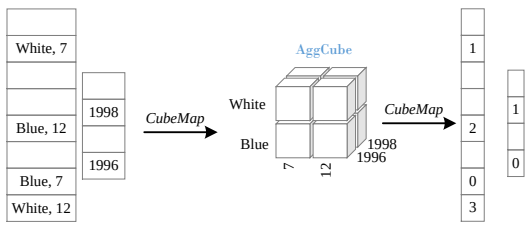


(b) Slicing for composite vector indexes

Fig. 5. Slicing on Aggregating cube.



(a) Mapping vector indexes to aggregating cube



(b) Mapping composite vector indexes to aggregating cube

Fig. 4. Aggregating cube mapping.

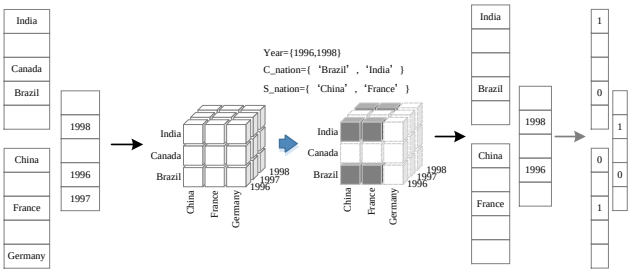


Fig. 6. Dicing on Aggregating cube.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

3.2.5 Dicing

A dicing operation produces a subcube by assigning specified values of multiple dimensions. In Fusion OLAP model, dicing operation is reduction operation on multiple dimension vector indexes. For example, in figure 6, dicing operation specifies values on aggregating cube dimensions, the subcube is reconstructed and the dimension vector indexes are updated by removing values not including in dicing conditions. By cube mapping, the dimension vector indexes are updated with new aggregating cube addresses.

3.2.6 Rollup

A rollup operation summarizes the data along a dimension of aggregating cube. In Fusion OLAP model, rollup operation updates the aggregating cube and vector indexes. Figure 7 illustrates how rollup operation works. In this example, aggregating cube comprises with 3 dimensions, *year*, *c_nation* and *s_nation*, 3 dimension vector indexes give the dimension mapping from 3 dimension tables to 3 vector indexes. By cube mapping, 3 dimension vector indexes update with aggregating cube addresses. The multidimensional filtering operation produces a fact vector index to retrieve fact data.

3.2.7 Drilldown

A drilldown operation shrinks the data view from high hierarchy level to lower hierarchy level to give user a detailed view. Figure 8 illustrates the drilldown operation. For a high-level aggregating hierarchy like *Region*, hierarchy member "EUROPE" has two lower level hierarchy members "Italy" and "Spain", the drilldown operation on "EUROPE" produces a new cube with other dimensions and new dimension with lower hierarchy members. For Fusion OLAP model, the drilldown operation produces updates on vector indexes. For example, drilldown operation on hierarchy member "EUROPE" produces filtering and update operation on corresponding vector index, "EUROPE" is used as additional filtering expression for vector index, and the filtered vector index cells are updated with corresponding nation hierarchy values. By cube mapping operation, the vector indexes are refreshed with new aggregating cube addresses. The current fact vector index needs two steps to refresh the fact vector index, the first step is to further filter the fact vector index by filtering the *non-NULL* values with drilldown value address constraint (e.g., (1,0,0) is removed because the 2nd coordinate value 0 is not the address of "EUROPE"(1)), the second step is to update the fact vector index values with new aggregating cube addresses(the 2nd coordinate value is set to new *nation* dimension address of aggregating cube).

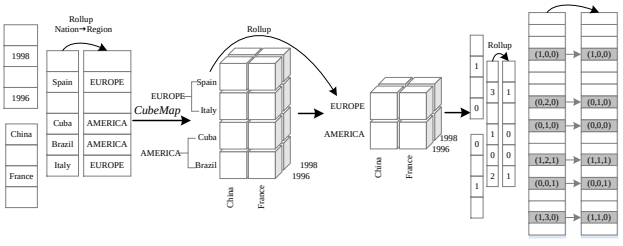


Fig. 7. Rollup operation.

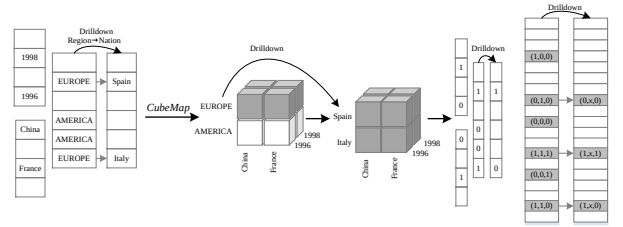


Fig. 8. Drilldown operation.

3.2.8 Pivot

A pivot operation gives various views of data by rotating the cube on specified dimension. A pivot operation only changes the multidimensional address in fact vector index, which can be easily implemented by multidimensional address transformation. Figure 9 illustrates the pivot operation. The original aggregating cube is defined by *nation*, *region* and *year* dimensions, and we mark the three dimensions as X, Y and Z. The multidimensional address in fact vector index can be presented as (x,y,z). For example, we use (1,0,1):5 denote 3-D address and 1-D address in Figure 9. If we rotate the cube by X axis, the new cube changes Y and X axis. Then non-*NULL* value in fact vector index represents which fact data satisfies the multidimensional filtering and the address of aggregating cube for summarizing.

4 Fusion OLAP Implementations

This section discusses how to implement Fusion OLAP model with state-of-the-art analytical in-memory databases to accelerate OLAP performance.

4.1 Relational storage implementation

To make MOLAP model efficient in storage model, Fusion OLAP model normalizes the multidimensional dataset as relational data. The dimensions are stored as dimension tables, and the fact data are mapping to fact tables with multidimensional addresses as additional fact attributes, and the storage model only contains relational dimensions and fact data as base cube.

ROLAP model also uses relational database as OLAP engine, and the dimensions are stored as dimension tables with primary key constraints, and the fact data are stored as fact table with foreign key constraints to guarantee each fact data can uniquely locate one dimensional tuple in dimension table. The primary key constraint relaxes the constraint of multidimensional space, and it no longer strictly represents multidimensional address, so that fact tuple relies on relational join operations to explore the dimensional

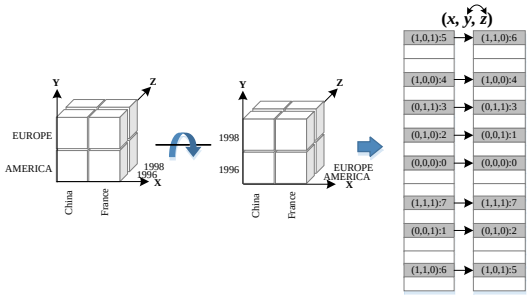


Fig.9. Pivot operation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

tuple with the same key as fact tuple's foreign key value. The ROLAP performance is dominated by relational database engine's performance, and aggregate tables are commonly used to precalculating data cube to reduce relational operation overhead.

As a major difference, Fusion OLAP model maintains the address mapping from MOLAP model, where the dimensions are strictly stored as relational tables with dimension coordinate system, and the original dimension coordinate is used as primary key of dimensional table. This dimension coordinate constraint guarantees directly mapping to vector index as dimension of virtual cube. Note that the foreign key in fact tuple can directly map to virtual cube instead of key probing oriented relational join operation.

4.2 Update mechanism

To store dimensions in relational database, we should employ additional mechanism to guarantee dimension coordinate for updates.

Most databases support auto-increment constraint in defining table statement, the "AUTO_INCREMENT" constraint defines a sequence for column with default step 1, and the key is automatically set for new tuple. The following SQL statement is an example of defining *supplier* dimension table, the column *S_Key* is primary key with auto-increment constraint which is used as *supplier* dimension.

```
CREATE TABLE Supplier (
  S_Key int NOT NULL AUTO_INCREMENT,
  Name varchar(50) NOT NULL,
  Nation varchar(50) NOT NULL,
  Region varchar(50) NOT NULL,
  PRIMARY KEY (S_Key) );
```

When creating clustered index for *S_Key* column, the tuples in dimension table have the same order with *supplier* dimension in MOLAP model.

Data warehouses usually employ surrogate key as primary key with consecutive sequence like 0,1,2,…, which is adaptive to be used as dimension coordinate for Fusion OLAP.

MOLAP uses pre-calculated cube for OLAP operation, where the updates on dimensions and fact data produce re-constructing data cube overhead. ROLAP has no constraints to maintain multidimensional space or cube, and the OLAP operation is transformed to SQL statements to be executed by database engine, where the updates only influence the base tables. Fusion OLAP model is a trade-off between MOLAP model and ROLAP model. For insert operation, Fusion OLAP model performs like ROLAP for dimension and fact tables' tuple appending. The extra constraint is auto-increment mechanism automatically assigns new primary key for inserted dimensional tuple. For delete operation, Fusion OLAP model follows the similar way as ROLAP model, the difference is that Fusion OLAP maps dimensional tuples to dimension so that the deleted dimensional tuples leave "holes" in dimension coordinate axis. The deleted dimensional tuples can be managed with three strategies: 1) deleting tuples for small and slowly increasing dimension table, mapping dimension primary key to dimension vector index, mapping the deleted tuples to corresponding *NULL* cells; 2) reusing the primary key of deleted tuple for new inserted tuple to change the mapping from dimension table to dimension vector index; and 3) reorganizing dimension table by assigning new auto-increment column as new primary key and updates the foreign key column in fact table when deleted tuples exceed threshold (by user configuration or by vector size relative to cache size). Figure 10 illustrates reorganizing dimension table by reusing deleted keys for existing tuples, identifying the changes in vector index and updating the relative multidimensional index column by vector index. For update operation, because the primary key of dimension table has no semantic information, we can add an extra constraint of not allowing change the value of existing primary key.

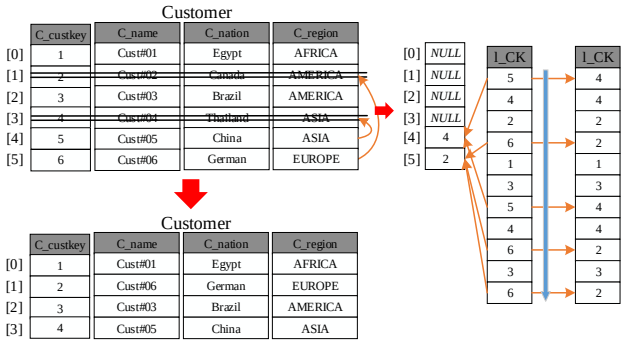


Fig.10. Batched consolidation of dimension table

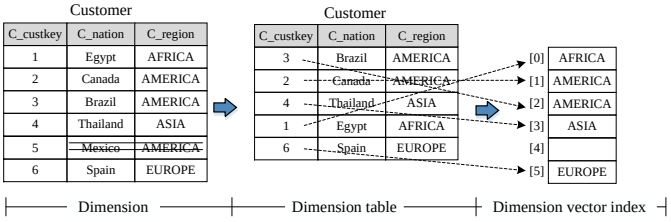


Fig.11. Logical dimension coordinate.

Since the dimension table may use other semantic attribute as clustered index column and the deleted dimensional tuple may make primary key of dimension table inconsecutive, we can relax the primary key of dimension table as virtual dimension coordinate by using auto-increment constraint and allowing deleting tuples or out-of-place update. In Figure 11, the dimension is strictly stored by dimension coordinate order, the deleted item leaves a *NULL* slice in the cube. The dimension table stores tuples by logical dimension coordinate i.e. auto-increment column, so that the clustered index, out-of-place update, delete mechanism may make dimension table out of order. The dimension vector index is an intermediator between dimension and dimension table, the vector size equals to the maximum of auto-increment primary key of dimension table, and the primary key is used as dimension vector index address for mapping dimension tuple. The deleted tuple maps to corresponding *NULL* cell in dimension vector.

4.3 Vector index for Fusion OLAP

Vector index is the intermediator between MOLAP and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

Algorithm 1 Algorithm for Creating Dimension Vector Index

```
1:  Vector Vec; //defining vector data structure
2:  HashTable HT; //defining hash table
3:  AttributeSet GP; //defining grouping attribute set
4:  Sequence ID; //defining auto-increment various value
5:  InitVec(Vec,max(R[PK])); //initiate vector by maximum PK value
6:  InitHT(HT(key,ID)); //initiate hash table with key and sequence
7:  GP=extractGP(Q,R); //get grouping attributes for R and query Q
8:  for each R[PK] do
9:    ID=HashProbing(R[GP]); //probe grouping attributes for ID
10:   Map(ID,R[PK],Vec); //store ID value to vector position R[PK]
11: end for
```

ROLAP. From MOLAP perspective, dimension vector index is a materialized dimension coordinate, the results of slicing and dicing operations project multidimensional addresses to corresponding dimension vector indexes and the aggregating cube further projects aggregating cube addresses to corresponding dimension vector indexes. From ROLAP perspective, dimension vector index is a bitmap index to filter foreign key columns and keys for grouping clause. The dimension vector index seems like a wide bitmap index, but the major differences are described as:

- Vector length. The length of bitmap index equals to the rows of relative table, while dimension vector index length may exceed the rows of relative dimension table for keeping deleted tuple positions.
- Vector map. Bitmap index maps bit to physical tuple position, while vector index maps each cell to logical dimension coordinate (auto-increment primary key), the key oriented mapping is adaptive to the update mechanism of relational database.
- Vector dependency. Bitmap index is used for filtering the corresponding table, while vector index is used for filtering the referencing table, the foreign key maps to the primary key position in vector index.
- Vector value. The value in bitmap denotes whether the corresponding tuple satisfies the selection condition. The value in vector index denotes the key for grouping.

Algorithm 1, the implementations for OLAP operations, describes the algorithm of creating dimension vector index. A hash table can be used to assign a unique sequential ID for each unique grouping attributes set, and mark the ID in dimension vector index according to primary key.

In relational database, Algorithm 1 can be simulated by a set of SQL statements. For example, *part* table has predicate *p_category* = 'MFGR#12' and outputs *p_brand1* as grouping attribute, the dimension vector index of *part* table can be implemented by:

```
CREATE TABLE vect(groups CHAR(30),id INTEGER AUTO_INCREMENT);

CREATE TABLE dimvec(vecid INTEGER, id INTEGER);

INSERT INTO vect(groups) SELECT DISTINCT p_brand1 FROM part WHERE p_category = 'MFGR#12';

INSERT INTO dimvec SELECT p_partkey,id FROM vect,part WHERE p_category = 'MFGR#12' AND groups=p_brand1;
```

Table *vect* represents aggregating cube dimension for each distinct grouping attribute with auto-increment ID, Table *dimvec* represents the compressed dimension vector index with sequence of primary key and aggregating cube dimension ID.

Algorithm 2 Algorithm for Multidimensional Filtering

```
1:  InitVec(FVec,RowNumber(F); //initiate fact vector by fact data rows
2:  for i=0 to DimNum do //loop for multidimensional index columns
3:  for j=0 to RowNumber(F) do //loop for fact data rows
4:    if (i>0 && FVec[j] is not NULL) //fact vector index as filter
5:      if (DimVec[i][MI[i][j]] is NULL) //block by dimension filter
6:        FVec[j]=NULL //label fact vector index cell as NULL
7:    else
8:          FVec[j]+=DimVec[i][MI[i][j]]*Card[i];
      //incrementally calculate aggregate cube address for fact vector index
9:      end if
10:   end if
11: end for
12: end for
```

If Fusion OLAP model is integrated into in-memory database, a customized creating dimension vector index API should be implemented to make this process more efficient than using SQL statements with scan and join cost.

4.4 Multidimensional filtering

Multidimensional filtering acts as an inverted multidimensional addressing of MOLAP, the multidimensional index value is mapped to corresponding dimension vector index position for *non-NULL* filtering, and the filtered multidimensional index calculates the aggregating cube address by dimension vector index values. The multidimensional filtering can be described as Algorithm 2.

In Algorithm 2, dimension vector index (*DimVec*[]) is used as dimension filter for multidimensional index (*MI*[][]), fact vector index (*FVec*[]) is used as both multidimensional bitmap index and aggregate cube index for fact data. The mapping from multidimensional index to dimension vector index inherits the inverted multidimensional addressing feature of MOLAP, it is similar to the primary key – foreign key reference in relational database with the essential difference that multidimensional index references the dimension vector index by addressing instead of key probing. This inverted multidimensional addressing mechanism is named as *Vector Referencing* opposite to foreign key referencing in relational databases, and the *vector referencing* process plays the same role as foreign key join in relational databases.

The performance of *vector referencing* is dominated by the latency of accessing the vector. The vector access latency can be improved by two roadmaps, by cache locality or by simultaneous multi-threading. State-of-the-art multicore processors equip with large LLC (Last Level Cache, e.g., 60M Cache in Intel® Xeon® Processor E7-8891 v4 processor), the LLC size is large enough for most of dimensions in data warehouses. For big vectors that exceed the LLC size, the *vector referencing* mechanism enables directly accessing the vector cell with at most one cache line miss. The cache locality feature is adaptive to multicore processor architecture and the emerging MIC (Many Integrated Core) Phi coprocessor architecture with hierarchical cache. The simultaneous multi-threading technique also helps to improve *vector referencing* performance by overlapping memory access latency. The multicore processor supports two threads for each core, the many-core Phi coprocessor supports four threads for each core which can further improve memory access latency. The multidimensional filtering algorithm uses shared dimension vector indexes, the equal length multidimensional index columns and fact

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

Algorithm 3 Algorithm for Vector Index oriented Aggregating

```
1: for i=0 to RowNumber(F) do //loop for fact table
2: if (FVec[i] is not NULL ) //filter fact table
3:     Res=AggPro(extractAggExpression(Q), FVec[i]);
   //aggregate by query defined expression and group address
4: end if
5: end for
6: for each t in Res do
7:     Gexpr=addToCube(t.key,AggCube);
   //get grouping attributes by mapping key to Aggregating Cube
8: updateRes(t.key, Gexpr);
9: end for
```

vector index column, the thread for multidimensional index row reads the shared dimension vector indexes and writes the result to the same position in fact vector index column with no writing conflicts. This design is also adaptive to GPU architecture with thousands of cores and SIMT (Single Instruction Multiple Threads) technique. In section 5, we will show how to improve multidimensional filtering performance with MIC Phi coprocessor and NVIDIA K80 GPU.

4.5 Vector index oriented aggregating

From MOLAP perspective, fact vector index acts as linear multidimensional addresses for compact fact data store. From ROLAP perspective, fact vector index acts as combination of bitmap index and aggregating index, the *non-NULL* cell can be used as either aggregating cube address or compressed hash key for grouping. With fact vector index, relational database can efficiently execute aggregations on fact data just as MOLAP engine does.

Algorithm 3 describes the vector index oriented aggregating. For an OLAP query, the grouping expression is integrated into fact vector index as single integer value, the aggregation can be performed with aggregating expression extracted from query Q and grouping value from fact vector index for each *non-NULL* position in fact vector index. The aggregating *AggPro* can be performed with either multidimensional array (as aggregating cube) or hash table. The result tuple is labeled by integer key, by mapping from key to aggregating cube, the result key is refreshed by group expression, and the result set is output as OLAP query results.

The fact vector index can be further optimized as binary table with row ID and value for highly selective queries, the relational database needs additional modification to support vector index oriented table scan. In Section 5, we simulate the fact vector index mechanism by using an additional column as fact vector index, the multidimensional filtering results are refreshed to the column, and the aggregation can be normalized as:

```
SELECT VecIdx,< AggExp > FROM F WHERE VecIdx IS NOT NULL GROUP
BY VecIdx;
```

5 Experiments

In this section, we evaluate the performance of Fusion OLAP model with benchmarks to simulate how Fusion OLAP model can accelerate state-of-the-art in-memory analytical database engines.

5.1 Experiment design and configuration

Fusion OLAP model targets at accelerating the performance of ROLAP model by integrating virtual multidimensional addressing features into relational database engine. Therefore, the experiment design majorly focuses on the updating maintenance cost, the core operator of Fusion OLAP model on different computing platforms and simulated benchmark performance.

The core operators of Fusion OLAP model are *vector referencing* and multidimensional filtering operators, which define the mapping from fact data address to dimension and multidimensional addressing from fact data to query concerned dimensions. These two operators are majorly implemented as hash join and star join in relational database engines. The previous research provides an open-source hash join algorithms in [13], and we adopt these open-source codes as testbed for evaluating Fusion OLAP operator performance. We add modules for *vector referencing* and multidimensional filtering operators inside the open-source codes to obtain the code efficiency and performance measurements. The compiled join performance is close to the JIT-compilation Hyper's join performance.

For a fair comparison, we install the most representative in-memory analytical databases of Hyper, Vectorwise and MonetDB with SSB (star schema benchmark) at Scale Factor of 100 with 600,000,000 fact rows and 4 dimension tables. SSB is normalized star schema benchmark opposite to 3NF oriented TPC-H benchmark, the 13 testing queries are divided into 4 groups with 1, 3 and 4 dimension tables joining with fact table, and each query group represents a drill-down operation in which there are 3 or 4 queries with selectivities from high to low for fine-grained analysis. SSB queries can be used as standard OLAP operations while most TPC-H queries are difficult to be used as OLAP operations with sub-query or cross dimension clauses. Fusion OLAP is designed as OLAP accelerator rather than general purpose query processing accelerator, so we limit our research scope in OLAP performance evaluation with SSB. Hyper is downloaded from [5], the Vectorwise version is 4.2.0 which is download from the action website (*http://esd.actian.com/*), and MonetDB is downloaded from MonetDB website (*https://www.monetdb.org/downloads/*) of the latest version MonetDB-11.21.19.

The experiments were performed on a DELL Power Edge R730 server with 2 Intel Xeon E5-2650 v3 @ 2.30GHz CPUs and 512 GB DDR3 main memory. Each CPU has 10 cores and 20 physical threads. The OS is CentOS, and the Linux kernel version is 2.6.32-431.el6.x86_64. The gcc compiler version is 4.4.7. The server equips with two Intel Xeon Phi 5110P coprocessors, each coprocessor has 60 cores and 240 threads. Moreover, the server also equips with a NVIDIA K80 GPU with two GK210 GPUs.

5.2 Maintenance cost evaluation

We evaluated Fusion OLAP maintenance cost by two perspectives, i.e. the storage efficiency and update overhead.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

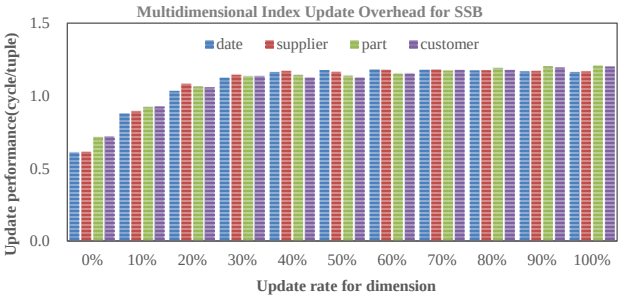


Fig. 12. Multidimensional index update performance for SSB.

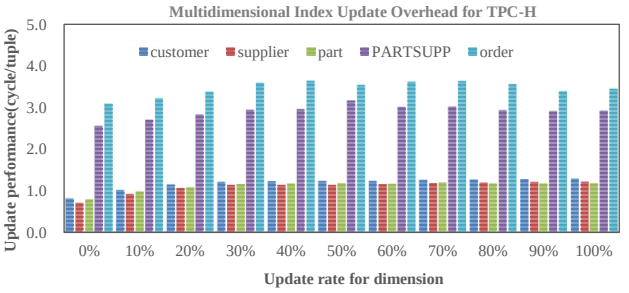


Fig. 13. Multidimensional index update performance for TPC-H.

The updates for dimension can be labelled in dimension vector, as shown in Figure 10, where the *NULL* cell represents non-update key, and the *non-NULL* cell represents the new assigned dimension key for original dimension key. The refreshing on multidimensional index produces a *vector referencing* for multidimensional index column and dimension column. Figure 12 gives the update performance for 4 dimensions in SSB, the fact table has 600,000,000 rows, the dimension row numbers of *date*, *supplier*, *part* and *customer* are 2,555, 200,000, 1,528,771, and 3,000,000. The x-axis represents update ratios, 0% represents a baseline *vector referencing* join performance. The overhead of update increases as the update rate rises, and the maximal update overhead are 90.42%, 91.09%, 64.81%, and 61.74% higher than original *vector referencing* operator.

In TPC-H, the updates on *customer* table involves multidimensional index column with 150,000,000 rows in orders table, other *vector referencing* operations all involve multidimensional index columns with 600,000,000 rows in *lineitem* table, the *supplier* and *part* dimensions have 1,000,000 and 20,000,000 rows, the *PARTSUPP* (80,000,000 rows) and *order* (150,000,000 rows) tables can also use *vector referencing* operation to accelerate traditional joins. Figure 13 gives the update overhead for TPC-H, for small dimension vectors (smaller than LLC size) from *supplier*, *part* and *customer* tables, the update overhead increases slowly as update rate rises, while large vectors for *PARTSUPP* and *order* tables, the middle update rates produce higher update overhead, the lower and higher update rates produces lower update overhead for auto branch prediction optimization improving the memory access for big vectors. The maximal update overhead are 52.95%, 66.73%, 46.43%, 20.42% and 15.37% higher than original vector referencing operator, the bigger the vector size is, the smaller the update overhead increases.

Logical surrogate key index mechanism is described in Section 4.2 Figure 11. We implemented this method by writing *payload* to vector cell mapped by *key*. Table 1 gives the increment rate of logical surrogate key index oriented *vector referencing* algorithm on TPC-DS(SF=100).

TABLE 1
LOGICAL SURROGATE KEY INDEX ORIENTED VECTOR REFERENCING OPERATION PERFORMANCE ON TPC-DS

TPC-DS Cycles Increment %	AIR Execution Time (cycles)			BUILD in TOTAL%
	BUILD %	PROBE %	TOTAL %	
reason	36.69%	-0.57%	-0.17%	1.06%
store	42.48%	-0.12%	-0.07%	0.13%
promotion	37.83%	0.41%	0.46%	0.13%
household_demographics	16.99%	-0.03%	-0.00003%	0.18%
date_dim	286.70%	-0.13%	0.08%	0.07%
time_dim	298.47%	-0.04%	0.20%	0.08%
item	112.67%	0.06%	0.29%	0.20%
customer_address	190.71%	0.29%	1.08%	0.41%
customer_demographics	171.94%	-0.04%	1.35%	0.81%
customer	207.83%	0.08%	1.64%	0.75%
store_returns	246.21%	-5.28%	7.76%	5.19%

For small dimension table that the corresponding vector size is smaller than LLC, logical surrogate key index method random writes to vector in cache, the increased build vector overhead is low. For big dimension tables with large vector, logical surrogate key index method involves more cache misses for random writing to vector, the increased build vector overhead is 2-3 times of physical surrogate key index method. But the proportion of build vector phase in total phases is very low, so that logical surrogate key index oriented *vector referencing* operation only pays limited overhead for vector mapping.

5.3 Core operator performance evaluation

For ROLAP, the core operator is foreign key join from fact table to dimension table, and the core operator of Fusion OLAP is *vector referencing* from multidimensional index to dimensions. The performance of foreign key join is dominated by how fast the referenced tuple is probed by *key*, while *vector referencing* is dominated by the speed of the vector cell read.

We first evaluate the performance of NPO and PRO algorithm in our platform to verify whether the performance status is consistence with conclusions of [13]. We configure NUM_PASSES 2 and NUM_RADIX_BITS 14 parameters for PRO which yield the best cache residency for our CPU through experiments. We use the open source code for Phi platform from [15] as Phi version NPO and PRO algorithms. Our *vector referencing* algorithm is compiled with icc compiler for Phi version. As we can not get available open source GPU hash join algorithm, we develop the cuda version *vector referencing* algorithm. In addition, we verify different configurations for block number and thread number to set the optimal parameters to compare the performance in multicore CPU, many-core MIC Phi and GPU platforms.

For SSB, the dimension vector indexes are smaller than LLC size (2.5KB for *date* dimension, 200KB for *supplier* di-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

mension, 1.5MB for *part* dimension, 3 MB for *customer* dimension and 25MB for LLC), the performance is very high for cache resident *vector referencing*. Figure 14 shows the performance of NPO, PRO and *vector referencing* algorithms for SSB. On the multicore CPU platform, *vector referencing* achieves higher performance for large L3 cache slice (2.5MB per core) and small vector size. Hash table organizes tuples in hash bucket, and each hash bucket has 8-byte *header* and 8-byte *next* pointer. While Fusion OLAP model only employs single vector for *payload* attribute without *key* attribute and other additional space overhead, the vector size can be further reduced by compression on low cardinality grouping attributes. The small vector size achieves performance gains by reducing cache misses from both TLB and LLC.

On the MIC Phi platform, *vector referencing* proves higher performance than on CPU platform for *date* and *supplier* dimensions and lower performance for *part* and *customer* dimensions. The Phi coprocessor has 512KB shared L2 cache, when vector size is lower than 512 KB, the massive parallel threading accelerates the performance, when vector size is higher than 512 KB, the *vector referencing* from share L2 cache ring produces higher latency than that from LLC on CPU. On the GPU platform, the *vector referencing* is accelerated by SIMT mechanism rather than cache, and the performance is lower than CPU and Phi for small vectors and higher than CPU and Phi for larger vector that exceeds the LLC size. NPO also achieves high performance for small dimension table by caching, and the performance drops remarkably as dimension table size increases. For Phi with smaller LLC size per core, the NPO performance on Phi is usually lower than on CPU for moderate dimension tables. PRO proves constant performance on CPU and Phi for different size of dimension tables by the normalized radix partitioning and in cache probing at the cost of 2x memory consumption for partitioning two tables.

Figure 15 shows the results for TPC-H. The sizes of five dimension vectors are 15MB, 1MB, 20MB, 80MB and 150MB, and the corresponding hash table is much larger than dimension vector. For cache fit joins, *vector referencing* and NPO all performs well on CPU platform, for large hash table, PRO outperforms NPO. For *vector referencing* algorithm, small vector size achieves the highest performance on CPU platform with large LLC, big vector achieves higher performance on GPU with SIMT mechanism than on CPU and Phi.

TPC-DS schema has multiple small dimension tables, whose size increase much slower than that of the fact tables. In addition, all the dimension vector sizes are smaller than LLC size. the *Vector referencing* algorithm performs well on CPU, Phi and GPU platforms, and NPO outperforms PRO for most cases on CPU and Phi platforms. For big tables, PRO outperforms NPO on both CPU and Phi platforms, and GPU also outperforms Phi for big tables.

As summary, *vector referencing* performance are majorly influenced by caching and memory accessing latency. When vector size is smaller than 512 KB (L2 cache size of Phi), Phi wins for its massive parallel threads; when vector is smaller than 25 MB (LLC size of CPU), CPU wins for vector caching; When vector is larger than LLC size, GPU wins for SIMT mechanism to overlap memory access latency.

Multi-table join is the core function of OLAP with multidimensional accesses. We use SSB and TPC-H as typical cases to evaluate multi-table join performance of state-of-the-art in-memory databases Hyper, Vectorwise and MonetDB with SQL statements. We also evaluate the *Vector Referencing* performance on CPU, Phi and GPU. We use the fixed length fact vector index, *vector referencing* and column-at-a-time processing as unified OLAP framework for different queries. For CPU and Phi platforms, the performance is dominated by caching dimension vector indexes, for GPU platform, and by memory accessing with SIMT mechanism. Table 2 shows the results of each multi-table join performance. GPU achieves the best performance for join with big tables, and the SIMT mechanism can efficiently overlaps the memory latency. For in-memory databases, multi-table join is still costly especially for joining with big table cases.

For Fusion OLAP model, the schema defines the vector index computing framework, and each query refreshes the index vectors with different value and distributions. To measure how different selectivities and amounts of tables affect performance, we use SSB to evaluate multidimensional filtering performance with groups of queries (with 1,3,4 dimension tables, different selectivities for roll-up and drill-down OLAP operators).

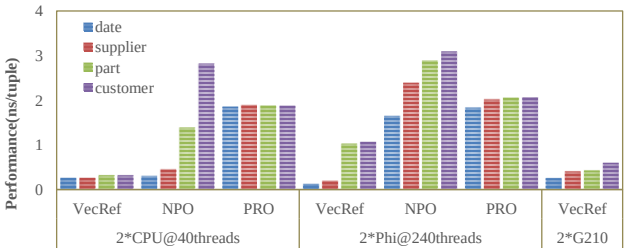


Fig. 14. Foreign key join performance for SSB.

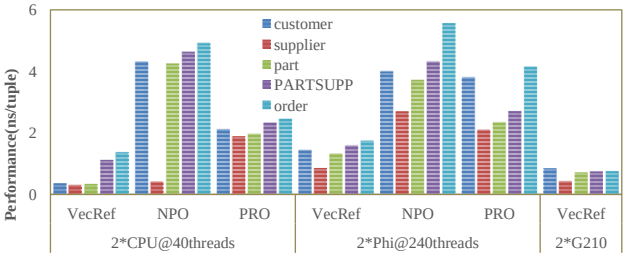


Fig. 15. Foreign key join performance for TPC-H.

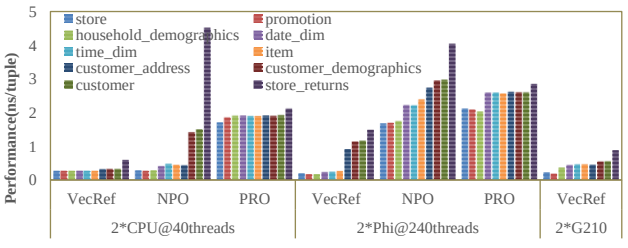


Fig. 16. Foreign key join performance for TPC-DS.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

TABLE 2
MULTI-TABLE JOIN PERFORMANCE(MS).

Bench.	Multi-Table Joins	VecRef@ CPU	VecRef@ Phi	VecRef@ GPU	MonetDB	Vectorwise	Hyper
SSB	lineorder⋈date	46	83	47	719	412	141
	lineorder⋈date⋈supplier	81	175	133	1400	1005	519
	lineorder⋈date⋈supplier⋈part	357	459	242	2100	1598	1161
	lineorder⋈date⋈supplier⋈part⋈customer	491	769	359	3100	2280	2295
TPC-H	lineitem⋈supplier	161	254	74	2300	851	399
	lineitem⋈supplier⋈part	307	611	269	8700	4024	2430
	lineitem⋈supplier⋈part⋈orders	864	1090	470	63000	7028	4133
	lineitem⋈supplier⋈part⋈orders⋈customer	913	1204	530	66000	8151	5252

The predicate expressions are mapped to dimension vectors as dimension filters, and the fact vector is used to incrementally calculate aggregating cube address and filter for next *vector referencing* operation.

We manually execute the algorithm with different selectivity and vector size orders on CPU and Phi platforms. We choose the minimal executing time as multidimensional filtering time. For GPU platform, we simply use selectivity prior strategy as GPU doesn't rely on cache.

Figure 17 illustrates the multidimensional filtering execution time for SSB with Scale Factor 100. For queries with relative high selectivity such as Q2.1, Q3.1, Q4.1, GPU works well for thousands of parallel working threads. For low selectivity queries, both CPU and Phi work well with sparse vector oriented random scan for the branch prediction and auto pre-fetch mechanisms. For the average multidimensional filtering execution time, GPU outperforms Phi and Phi outperforms CPU.

5.4 Benchmark performance evaluation

Fusion OLAP model targets at fusing the relational OLAP model and multidimensional OLAP model together, in which the dimensions and fact data are stored as relational tables. OLAP queries are transformed to multidimensional computing. The ideal roadmap is to add new modules inside the in-memory database such as vector index management module for dimension vector index and fact vector index, the multidimensional filtering module and vector oriented aggregating module. We have verified that the core Multidimensional Filtering algorithm can be implemented inside BAT module of open-source MonetDB with *fetchjoin*() function, and the vector index oriented aggregation can also be implemented as SQL processing with vector index BAT as grouping column, but entirely integrating Fusion OLAP model is a complex work involving adding new index and modifying system modules such as MAL-parser, MALoptimizer, MALscheduler, MALengine. Moreover, the MAL engine is quite different to other non-open-source in-memory databases.

Therefore, in our experiments, we use SQL statements to simulate creating dimension vector indexes. The fact vector index is simulated by adding a column, and vector index aggregating is simulated by SQL based aggregating with vector index column as WHERE and GROUP BY

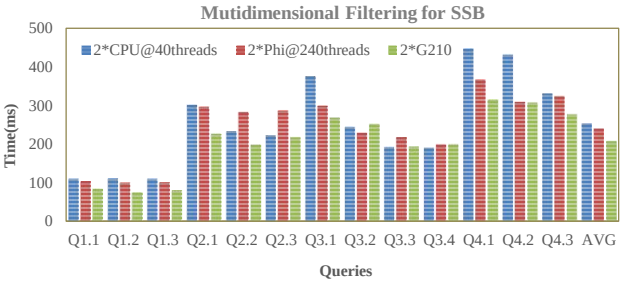


Fig. 17. Multidimensional filtering performance for SSB.

clause. The multidimensional filtering operation is designed as external module with normalized vectors as input and output, and the module can be adaptive to migrate to MIC Phi, GPU platforms or database engines. For example, Q4.1 of SSB is a typical OLAP query with four dimension tables, the dimension tables act as bitmap index or dimension vector index. The dimension with predicate expression (yet without grouping attribute) is used as bitmap index for filtering, while the dimension (with or without predicate expression) which appears in GROUP BY clause acts as vector index.

Q4.1:

```
SELECT d_year, c_nation, SUM(lo_revenue - lo_supplycost) AS profit
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey AND lo_partkey = p_partkey
  AND lo_orderdate = d_datekey AND c_region = 'AMERICA'
  AND s_region = 'AMERICA' AND (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
GROUP BY d_year, c_nation;
```

For Fusion OLAP model, the OLAP processing is divided into three stages. In first stage, we simulate the dimension vector index generating with SQL statements.

(1) creating dimension vector index for *customer* table

```
CREATE TABLE vect(groups CHAR(30),ID INTEGER AUTO_INCREMENT);
CREATE TABLE dimvec(key INTEGER, vec INTEGER);
INSERT INTO vect(groups) SELECT DISTINCT c_nation FROM customer WHERE c_region = 'AMERICA';
INSERT INTO dimvec SELECT c_custkey, ID FROM vect,customer WHERE c_region = 'AMERICA' AND groups=c_nation;
```

(2) creating dimension bitmap index for *supplier* table

```
CREATE TABLE bitmap(ID INTEGER);
INSERT INTO bitmap SELECT s_suppkey FROM supplier WHERE s_region = 'AMERICA';
```

(3) creating dimension bitmap index for *part* table

```
CREATE TABLE bitmap(ID INTEGER);
INSERT INTO bitmap SELECT p_partkey FROM part WHERE p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2';
```

(4) creating dimension vector index for *date* table

```
CREATE TABLE vect(groups INTEGER,ID INTEGER AUTO_INCREMENT);
CREATE TABLE dimvec(key INTEGER, vec INTEGER);
INSERT INTO vect(groups) SELECT DISTINCT d_year FROM date;
INSERT INTO dimvec SELECT d_datekey, ID FROM vect, date WHERE groups=d_year;
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering

TABLE 3
CREATING DIMENSION VECTOR INDEXES BY HYPER(S).

Query	GeDic1	GeVec1	GeDic2	GeVec2	GeDic3	GeVec3	GeDic4	GeVec4	ToTime
Q1.1		0.0001							0.0001
Q1.2		0.0001							0.0001
Q1.3		0.0001							0.0001
Q2.1	0.0023	0.0421		0.0086	2.05E-05	0.0007			0.0536
Q2.2	0.0050	0.0443		0.0088	1.20E-05	0.0007			0.0589
Q2.3	0.0019	0.0130		0.0108					0.0257
Q3.1	0.0045	0.0685	0.0016	0.0093	1.23E-05	0.0006			0.0845
Q3.2	0.0036	0.0798	0.0015	0.0056	1.11E-05	0.0007			0.0912
Q3.3	0.0047	0.0611	0.0028	0.0040	1.07E-05	0.0007			0.0733
Q3.4	0.0048	0.0611	0.0029	0.0041	1.70E-05	0.0002			0.0731
Q4.1	0.0040	0.0237		0.0079		0.0437	1.78E-05	0.0013	0.0807
Q4.2		0.0449	0.0020	0.0138	0.0035	0.0000	1.25E-05	0.0004	0.0647
Q4.3		0.0449	0.0013	0.0071	0.0023	0.0135	1.25E-05	0.0004	0.0694

TABLE **4**
CREATING DIMENSION VECTOR INDEXES BY VECTORWISE(S).

Query	GeDic1	GeVec1	GeDic2	GeVec2	GeDic3	GeVec3	GeDic4	GeVec4	ToTime
Q1.1		0.005							0.005
Q1.2		0.004							0.004
Q1.3		0.005							0.005
Q2.1	0.015	0.021		0.006	4.00E-03	1.20E-02			0.058
Q2.2	0.013	0.02		0.006	3.00E-03	0.016			0.058
Q2.3	0.009	0.018		0.006	0.004	0.012			0.049
Q3.1	0.017	0.032	0.007	0.018	4.00E-03	0.012			0.09
Q3.2	0.015	0.022	0.006	0.015	4.00E-03	0.012			0.074
Q3.3	0.016	0.02	0.005	0.013	3.00E-03	0.011			0.068
Q3.4	0.017	0.02	0.005	0.013	4.00E-03	0.011			0.07
Q4.1	0.017	0.027		0.007		0.013	4.00E-03	0.01	0.078
Q4.2		0.012	0.007	0.016	1.50E-02	0.031	3.00E-03	0.01	0.094
Q4.3		0.013	0.006	0.015	0.014	0.002	4.00E-03	0.001	0.055

TABLE 5
CREATING DIMENSION VECTOR INDEXES BY MONETDB(S).

Query	GeDic1	GeVec1	GeDic2	GeVec2	GeDic3	GeVec3	GeDic4	GeVec4	ToTime
Q1.1		0.002							0.002
Q1.2		0.001							0.001
Q1.3		0.001							0.001
Q2.1	0.016	0.029		0.019	0.003	0.004			0.071
Q2.2	0.018	0.019		0.018	0.001	0.003			0.058
Q2.3	0.015	0.012		0.018	0.002	0.002			0.048
Q3.1	0.010	0.279	0.017	0.024	0.003	0.002			0.336
Q3.2	0.015	0.069	0.016	0.013	0.002	0.003			0.118
Q3.3	0.017	0.044	0.016	0.010	0.002	0.003			0.092
Q3.4	0.015	0.034	0.017	0.012	0.003	0.003			0.082
Q4.1	0.020	0.282		0.021		0.095	0.002	0.002	0.421
Q4.2		0.095	0.016	0.026	0.023	0.286	0.003	0.003	0.452
Q4.3		0.098	0.017	0.018	0.034	0.041	0.002	0.003	0.213

By these SQL statements, we obtain the relational table as dimension vector indexes. For the build-in dimension vector index generator, creating group dictionary table and creating dimension vector index by join can be optimized within the single scan as described in Algorithm 1.

Table 3-5 give the results of SQL oriented dimension vector index creating time of Hyper, Vectorwise and MonetDB.

Hyper cannot supports AUTO_INCREMENT constraint, and thus we simply use a NULL integer value to simulate dimension key. Each query has different amount of dimension vector indexes or bitmap indexes. We summarize all the time for generating group dictionary table and dimension vector index table as total time (ToTime) for generating dimension vector indexes in the first stage.

In the second stage, the multidimensional filtering stage uses our compiled program to evaluate execution time for CPU, Phi and GPU platforms. The execution time is shown in figure 17.

In the third stage, we get a fact vector index to identify which fact tuple belongs to the multidimensional subset of query and the address in aggregating cube. The OLAP query is simplified as single table aggregation. In experiments, we add a *vector* column in fact table, update the column with -1 (denotes *NULL*) or group ID (or 1 for bitmap index) according to selectivity and group cardinality. For examples, for Q1.1, we update *vector* column with 1 and -1 according to selectivity, and we rewrite the SQL statement with additional predicate expression "vector=1".

Q1.1: UPDATE lineorder SET vector=(CASE WHEN lo_orderkey <= 0.142857 * 600000000 THEN 1 ELSE -1 END);

SELECT SUM(lo_extendedprice*lo_discount) AS revenue FROM lineorder WHERE vector=1 AND lo_discount BETWEEN 1 AND 3 AND lo_quantity < 25;

For Q4.1, we update the *vector* column with group cardinality and selectivity to simulate the fact vector index generated by multidimensional filtering stage. The query is rewritten with additional predicate expression "vector>=0" and group by clause with *vector* column.

Q4.1: UPDATE lineorder SET vector=(CASE WHEN lo_orderkey <= 0.016000 * 600000000 THEN (lo_orderkey%35) ELSE -1 END);

SELECT vector, SUM(lo_revenue - lo_supplycost) AS profit FROM lineorder WHERE vector>=0 GROUP BY vector;

Figure 18 shows the vector index oriented aggregation performance for Hyper, Vectorwise and MonetDB. For high selectivity like Q1.1(14.29%), Q3.1(3.4%), Q4.1(1.6%), MonetDB commonly spends more time for aggregating, for low selectivity queries, MonetDB has similar aggregating performance as Hyper. Vectorwise proves to be higher performance than the other two for the efficient vectorized processing in vector index filtering.

We also evaluate the OLAP performance accelerated by Fusion OLAP with CPU, Phi and GPU. Figure 19(a)(b)(c) show the breakdown query processing time of Fusion OLAP with different processors for multidimensional filtering and database oriented dimension vector index processing and aggregating. For queries with high selectivity like Qx.1, the major execution time is spent on aggregation,

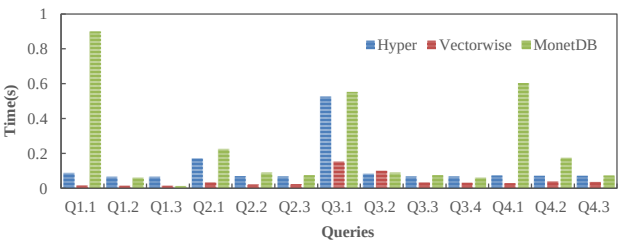
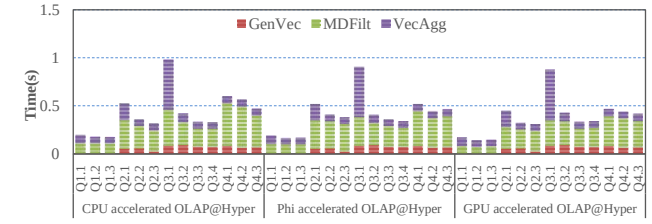
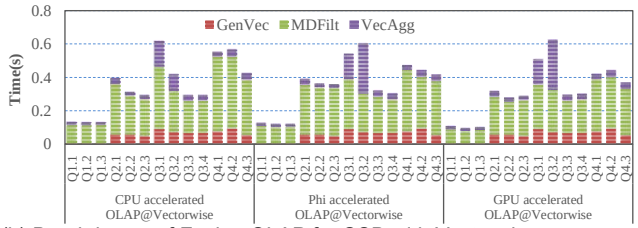


Fig. 18. Aggregation performance for SSB.

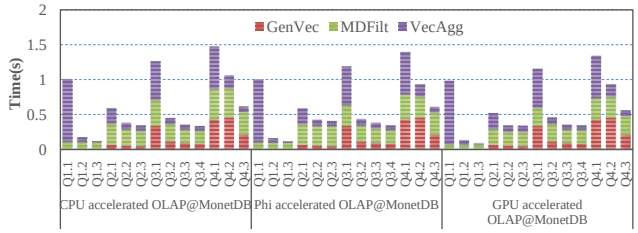
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2018.2867522, IEEE Transactions on Knowledge and Data Engineering



(a) Breakdowns of Fusion OLAP for SSB with Hyper.



(b) Breakdowns of Fusion OLAP for SSB with Vectorwise.



(c) Breakdowns of Fusion OLAP for SSB with MonetDB.

Fig. 19. Breakdowns of Fusion OLAP for SSB.

for queries with low selectivity, the multidimensional filtering dominates the whole performance which can be accelerated by CPU, Phi and GPU. Compared with hash join algorithms, *vector referencing* algorithm is adaptive for both caching and massive parallel processing, it achieves high performance on CPU, Phi and GPU platforms.

Join is the most complex and costly operation in relational database, which dominates the performance of relational database. Figure 20 gives the results of average query execution time of 13 OLAP queries in SSB, with Fusion OLAP accelerating techniques, and each in-memory databases achieves better performance than ever. Hyper achieves up to 35% improvement by GPU accelerated Fusion OLAP; Vectorwise outperforms Hyper and MonetDB in dimension vector index processing and aggregation. Vectorwise achieves maximal 365% improvement by GPU accelerated Fusion OLAP; MonetDB takes more time in aggregating than Hyper and Vectorwise, the maximal improvement achieves 169%. Besides performance improvement, the core module of multidimensional filtering can be deployed on multicore CPU platform, many-core Phi platform and GPU platform, and the multidimensional filtering module can be adaptive to be embedded in relational database engines with simple vectors as input and output data.

Summary. We summarize the main findings in our experiments belows.

(1) The maintenance cost of Fusion OLAP model is acceptable. The index update is performed with efficient *vector referencing* operations for big dimension update in TPC-H, and the update overhead is only 15.37% higher than original vector referencing operator which is invoked for a long time. The logical surrogate key indexes make *vector referencing* adaptive to update operations, and thus the additional overhead is below 2% for small and moderate dimensions, while the maximal overhead is about 6% for the big referenced fact table in TPC-DS. Therefore, the overall update maintenance overhead is small enough for updates scenarios.

(2) For the core star-join operations of OLAP, *vector referencing* outperforms state-of-the-art in-memory analytical databases with CPU, Phi and GPU platforms. We achieve the significant speedup of up to 7~9x for SSB and TPC-H star-join tests.

(3) Finally, we verify how to accelerate in-memory analytical databases with vector index mechanism. The simulation results show that vector indexes can futhre achieve 35%~365% improvement on in-memory databases.

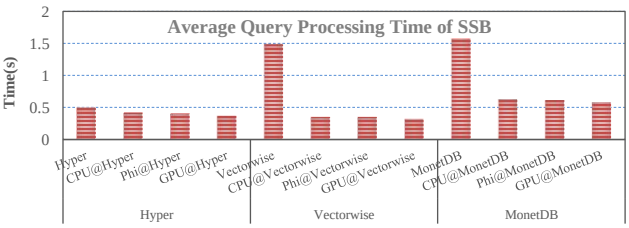


Fig. 20. Average Query Execution Time of SSB.

6 Related Work

In the era of in-memory computing, main-memory becomes a new disk to bridge the big performance gap between DRAM and processor. One recent research topic is to improve in-memory join performance with hardware-conscious optimization techniques. The no-partitioning hash table join represents hardware-oblivious join algorithm [18] without considering hardware features of TLB size, page size, SIMD etc., which is simple for implementation but is proved to be less efficient than hardware-conscious join algorithms of radix partitioning hash joins and sort-merge joins [13] [19]. Hardware-conscious joins achieve better performance, but they are difficult to tune for maximal performance [22] [24]. A neglected point is that hardware-conscious joins commonly need double space consumption in the partitioning or sorting phases, which may significantly reduce the rate of memory utilisation.

The new trend is to accelerate query processing performance with hardware accelerators. As MIC Phi equips with 512-bit vector processing unit inside core, [15] further developed the source code of [13] with 512-bit SIMD optimizations on MIC Phi platform. Their experimental results show that no-partitioning hash join benefits more than radix partitioning hash join on Phi opposite to CPU platform. The latest research [16] used entirely vector codes for fundamental operations such as selection scan, hash table and partitioning. The hash join optimization is also implemented on APU platform [20], GPU platform [14] [21] and FPGA platform [17]. The platform-conscious designs keep improving performance by tuning algorithm with new hardware parameters [25], the

hardware feature variety and complexity make these optimization approaches hardly benefits for different coprocessors.

Another roadmap is to exploit new OLAP operation, instead of the relational join operation. Paper [26] proposed a native join index to accelerate join performance, and [23] further proposed an array oriented storage and computing model for data warehouse. These methods simplify hash joins with array addressing, and rely less in hardware features which can be considered as hardware-oblivious join algorithms for both CPU and coprocessors platforms. Unlike the hash joins, the join performance of the array addressing is majorly dominated by the array size which is pre-determined by data warehouse schemas and the LLC size of multicore. HBM size of KNL Phi and SIMT mechanism of GPU can automatically make array addressing efficient. This paper further exploits how to develop a multidimensional processing oriented OLAP model to make OLAP faster and more adaptive to hybrid coprocessors.

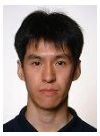
7 Conclusions

In this paper, we have proposed Fusion OLAP model which fuses the benefits of high performance multidimensional addressing from MOLAP with the advantage of efficient storage from ROLAP together. The multidimensional address module can be efficiently implemented by multicore CPU, MIC Phi and GPU processors for its simple vector structure and *vector referencing* operations. This feature improves in-memory analytical database performance by hardware oriented multidimensional addressing accelerator with the minimal changes for databases. We have evaluated Fusion OLAP performance by simulating Fusion OLAP operations with staged processing, which combine customized vector index computing with vector index oriented aggregating phases. In our future work, we will further explore how to integrate the Fusion OLAP techniques inside in-memory databases to make them adaptive to the emerging hybrid coprocessor platforms.

Acknowledgements. This work was supported by the National Natural Science Foundation of China (61732014, 61772533) and Academy of Finland (310321). Yu Zhang is the corresponding author.

References

[1] http://go.sap.com/documents/2016/03/ba1aa531-647c-0010-82c7-eda71af511fa.html
[2] https://www.monetdb.org/Home
[3] http://www.actian.com/products/big-data-analytics-platforms-with-hadoop/vector-smp-analytics-database/
[4] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, M. Grund, "High-Performance Transaction Processing in SAP HANA". IEEE Data Eng. Bull. 36(2): 28-33 (2013)
[5] http://hyper-db.com/
[6] http://www.memsql.com/
[7] http://www.oracle.com/us/corporate/features/database-in-memory-option/index.html
[8] http://www.ibmbluhub.com/
[9] P. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, V. Papadimos, "Real-Time Analytical Processing with SQL Server". *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1740-1751, 2015.
[10] P. A. Boncz, S. Manegold, and M. L. Kersten. "Database architecture optimized for the new bottleneck: Memory access". *Proc. VLDB Endowment*, pages 54–65, 1999.
[11] P. A. Boncz, M. Zukowski, N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution". *Proc. CIDR*, 2005: 225-237
[12] T. Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". *Proc. VLDB Endowment*, vol. 4, no. 9, pp. 539-550, 2011.
[13] C.Balkesen, J. Teubner, G. Alonso, M. T.Özsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware". *Proc. ICDE*, pages 362-373, 2013.
[14] Y. Yuan, R. Lee, X. Zhang, "The Yin and Yang of Processing Data Warehousing Queries on GPU Devices". PVLDB 6(10): 817-828 (2013)
[15] S. Jha, B. He, M. Lu, X. Cheng, H. P. Huyn, "Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach". PVLDB 8(6): 642-653, 2015.
[16] O. Polychroniou, A. Raghavan, K. A. Ross, "Rethinking SIMD Vectorization for In-Memory Databases". *Proc. SIGMOD*, pp. 1493-1508, 2015.
[17] R. J. Halstead, I. Absalyamov, W. A. Najjar, V. J. Tsotras, "FPGA-based Multithreading for In-Memory Hash Joins". *Proc. CIDR*, 2015.
[18] S.Blanas, Y. Li, J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs". *Proc. SIGMOD*, pp. 37-48, 2011.
[19] C. Balkesen, G. Alonso, J. Teubner, M. T. Özsu, "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited". PVLDB 7(1): 85-96, 2013.
[20] J. He, M. Lu, B. He, "Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture". PVLDB 6(10): 889-900, 2013.
[21] S. Breß, M. Heimel, M. Saecker, B. Kocher, V. Markl, G. Saake, "Ocelot/HyPE: Optimized Data Processing on Heterogeneous Hardware". PVLDB 7(13): 1609-1612, 2014.
[22] S. Richter, V. Alvarez, J. Dittrich, "A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing". PVLDB 9(3): 96-107, 2015.
[23] Y Zhang, X Zhou, Y Zhang, Y Zhang, M Su, S Wang. Virtual Denormalization via Array Index Reference for Main Memory OLAP. IEEE Trans. Knowl. Data Eng. 28(4): 1061-1074 (2016).
[24] S Schuh, X Chen, J Dittrich, "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory". *Proc. SIGMOD*, pp. 1961-1976,2016.
[25] R Rui, Y Tu, "Fast Equi-Join Algorithms on GPUs: Design and Implementation". *Proc.SSDBM*, pages 1-12, 2017.
[26] Y Zhang, S Wang, J Lu, "Improving performance by creating a native join-index for OLAP". Frontiers Comput. Sci. China 5(2): 236-249, 2011.



Yansong Zhang is an associate professor in the School of Information at the Renmin University of China. His research interests include Main-Memory Database, Data Warehouse, OLAP and Coprocessor processing.

Yu Zhang is an associate professor in National Satellite Meteorological Center. Her research interests include GPU based OLAP and database optimization.

Shan Wang is a professor in the School of Information at the Renmin University of China. Her current research focuses on high performance database, data warehouse, knowledge engineering and information retrieval.

Jiaheng Lu is an associate professor of the Department of Computer Science at the University of Helsinki, Finland. His recent research interests include multi-model database management systems, and job optimization for big data platform.