Virtual Denormalization via Array Index Reference for Main Memory OLAP

Yansong Zhang, Xuan Zhou, Ying Zhang, Yu Zhang, Mingchuan Su, and Shan Wang

Abstract—Denormalization is a common tactic for enhancing performance of data warehouses, though its side-effect is quite obvious. Besides being confronted with update abnormality, denormalization has to consume additional storage space. As a result, this tactic is rarely used in main memory databases, which regards storage space, i.e., RAM, as scarce resource. Nevertheless, our research reveals that main memory database can benefit enormously from denormalization, as it is able to remarkably simplify the query processing plans and reduce the computation cost. In this paper, we present A-Store, a main memory OLAP engine *customized* for star/snowflake schemas. Instead of generating fully materialized denormalization, A-Store resorts to *virtual denormalization* by treating array indexes as primary keys. This design allows us to harvest the benefit of denormalization without sacrificing additional RAM space. A-Store uses a generic query processing model for all SPJGA queries. It applies a number of state-of-the-art optimization methods, such as vectorized scan and aggregation, to achieve superior performance. Our experiments show that A-Store outperforms the most prestigious MMDB systems significantly in star/snowflake schema based query processing.

Index Terms—Main-memory, OLAP, denormalization, A-Store, array index.

1 Introduction

The purpose of database normalization is to eliminate data redundancy, so as to save storage space and avoid update abnormality. It is usually achieved by decomposing a large relation into several small ones, connected by foreign keys. Star and snowflake schemas are typical forms of normalization, in which data is decomposed into fact tables and dimension tables. Despite its benefits, normalization introduces performance penalty to a DBMS, as it has to process multiple relations separately and perform expensive joins to integrate the intermediate results. Therefore, denormalization is sometimes applied to optimize the performance of a DBMS. Denormalization reverses the process of normalization by joining multiple relations back into one, such that query processing can be conducted on a single table. It simplifies query execution plans and eliminates expensive join operations.

Recently, there has been an increasing interest in the technology of main memory database (MMDB), thanks to the advance of hardware technology which yields increasingly powerful CPUs and large RAMs. As RAM is much more expensive than disk, MMDB rarely considers denormalization as an optimization approach. However, this does not necessarily mean that MMDB cannot benefit from the strategy of denormalization. Recent development of MMDB [1], [2], [3] has shown that simplicity of data processing algorithms is an important aspect of good performance. Simple program can utilize modern CPU architectures, such as the features of prefetching and SIMD, more efficiently. It can also help to achieve better data locality and thus cache efficiency. Denormalization, in its very nature, offers such simplicity.

To assess the potential of denomalization in accelerating in-memory data processing, we measured the performance of a fully denormalized in-memory database implemented with C++. We compared it against three prestigious MMDBs, MonetDB, Vectorwise and Hyper on the Star Schema Benchmark (SSB, SF = 100). We also evaluated the denormalized versions of the MMDBs. As shown in Fig. 1, the denormalized MMDBs (with "_D" suffix) commonly outperform the original MMDBs, with the exception of MonetDB (explained in section 7). Our hand-code denormalized implementation exhibits the best performance among all the candidate MMDBs. The benefit of denormalization seems attractive. It encourages us to conduct further research in this direction.

Inspired by the aforementioned observation, we created A-Store, a prototype main-memory database system that applies the strategy of denormalization to achieve highly efficient OLAP over typical star and snowflake schemas. Instead of exercising fully materialized denormalization, A-Store applies a method called *virtual denomalization*, which allows query processing to be performed in a denormalized way, while *without incurring additional space consumption*. Fig. 1 shows that virtual denormalization, i.e., A-store, performs similarly as the hand-code denormalization, and outperforms the other MMDBs.

Specifically, A-Store adopts a column oriented storage model [5], [6], in which a table is organized as an array family composed of a set of arrays of equal length, each representing

- *Y. Zhang is with the DEKE Lab and the National Survey Research Centre, Renmin University of China, Beijing 100872, China. E-mail: zhangys_ruc@ hotmail.com.*
- *X. Zhou, Y. Zhang, M. Su, and S. Wang are with the DEKE Lab, Renmin University of China, Beijing 100872, China. E-mail: zhou.xuan@outlook.com, {zyszy, minchuan, swang}@ruc.edu.cn.*
- *Y. Zhang is with the Centrum Wiskunde & Informatica, Amsterdam, Netherlands. E-mail: y.zhang@cwi.nl.*

Manuscript received 16 Jan. 2015; revised 11 Oct. 2015; accepted 26 Oct. 2015. Date of publication 0 . 0000; date of current version 0 . 0000. Recommended for acceptance by C. Chan. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2015.2499199

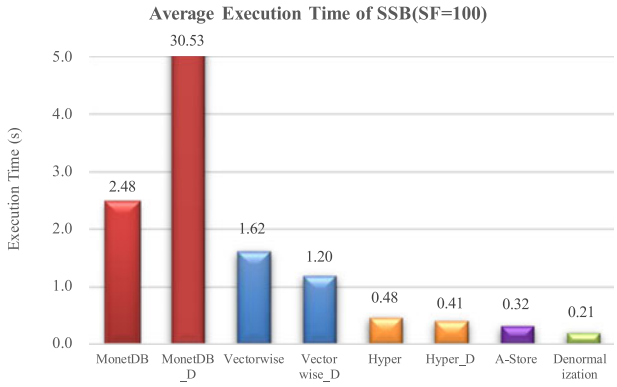


Fig. 1. Denormalization versus normal MMDBs on SSB [4].

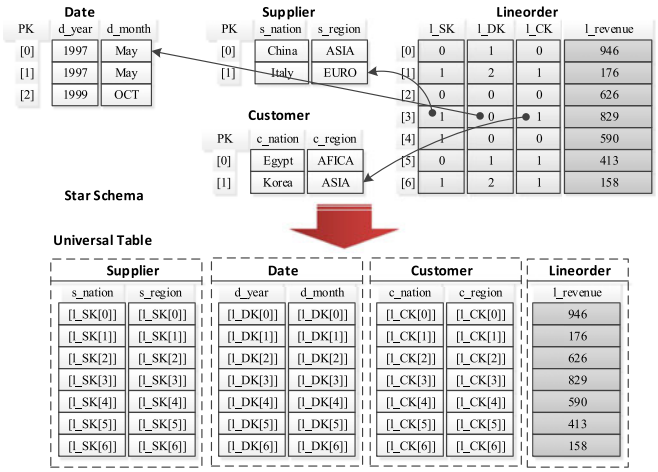


Fig. 2. Array families and universal table.

a column of the table. The core mechanism of virtual denormalization is to treat array indexes as primary keys, such that each tuple is directly addressable by its key. This turns foreign keys into the array indexes of the referenced tables too, known as array index reference (AIR), through which join operations can be accomplished by direct positional tuple access. This design integrates the tables of an entire star/snowflake schema into a *virtually denormalized relation*. An arbitrary SPJGA OLAP query can be processed by simply scanning the relation. Based on the storage model of A-Store, we propose a generic query processing model and a set of optimization methods specialized for SPJGA OLAP queries. We conducted extensive experiments. The results show that virtual denormalization enables A-Store to outperform the most prestigious MMDBs significantly in star/snowflake-schema based query processing.

By virtual denormalization, we refer to the design of A-Store's complete OLAP framework, which consists of a storage model, a query processing model and a set of optimization techniques. We present the storage model of A-Store in Section 2 and its query processing model in Section 3. The optimization techniques of A-Store are introduced in Section 4. Section 5 discusses the parallel implementation of A-store. In Section 6, we evaluate A-Store experimentally and compare it against other main memory databases. Related work is discussed in Section 6. Finally, we conclude this paper in Section 7.

2 The Storage Model

In A-Store, we store a relational table in an *array family*, which is composed of a set of arrays of equal length, each representing a column of the table. The arrays in an array family are completely aligned with one another, such that all the *i*th elements in the arrays constitute the *i*th tuple of the table. As to a column of variable length, e.g., *varchar*, we do not store the contents of the column in its array directly. Instead, we store its contents in a dynamically allocated memory space and keep their addresses in the array. As array indexes can be used to directly locate the tuples in a table, A-Store treats the array index as the primary key of a table. The storage model of A-Store closely resembles that of MonetDB, except that MonetDB uses Binary Associated Table (BAT), while A-Store uses array, bitmap and vector as basis storage objects.

Fig. 2 provides an example of the storage layout of A-Store. The database consists of four tables—*Lineorder*, *Date*, *Supplier* and *Customer*. Each table is stored in an array family. No explicit primary key is created for the tables, as we can use the array index to locate their tuples. For instance, the fourth tuple of *Lineorder* is composed of *l_SK*[1], *l_DK*[0], *l_CK*[1] and *l_revenue* [3].

Accordingly, the column of a foreign key directly refers to the array index of its reference table. This is named AIR. As illustrated in Fig. 1, the foreign keys of the *lineorder* table refer to the array indexes of the other three tables. This design allows us to use foreign keys to directly locate the tuples in the reference tables. Thus, conventional join algorithms, such as nest-loop join and hash join [7], [8], are usually not required. For instance, in a typical data warehouse based on a star or snowflake schema, data is organized in a fact table and a set of dimension tables. Multi-way join is commonly performed to integrate the data from these two types of tables. In A-Store, a multi-way join is not necessary. We can get the entire universal relation by scanning the fact table and fetching referred tuples in the dimension tables using array indexes. In effect, A-Store creates a virtual denormalization of its data.

For columns with low cardinality, such as *c_region* and *c_nation* in the *customer* table, A-Store uses dictionary compression to reduce their space consumption. A-Store uses arrays to store dictionaries and uses array indexes as compression codes [9]. Thus, decompression can be performed by simple array lookup too. In fact, a dictionary can be regarded as a reference table in A-Store. The compressed column can be regarded as a foreign key to the reference table.

3 The Query Processing Model

The current version of A-Store is not designed for general purpose relational algebra. It is customized for the multidimensional model, the data model considered by MOLAP [35]. Therefore, A-Store only deals with Selection-Projection-Join-Grouping-Aggregation (SPJGA) queries on star/snowflake schemas, which are actually the most common cases for OLAP applications. A-Store does not consider relational operators that are not supported by MOLAP, such as non-PK-FK join and self-join. When applied, A-store can be used as an auxiliary OLAP engine of a general purpose MMDB, which specializes in SPJGA queries or sub-queries.

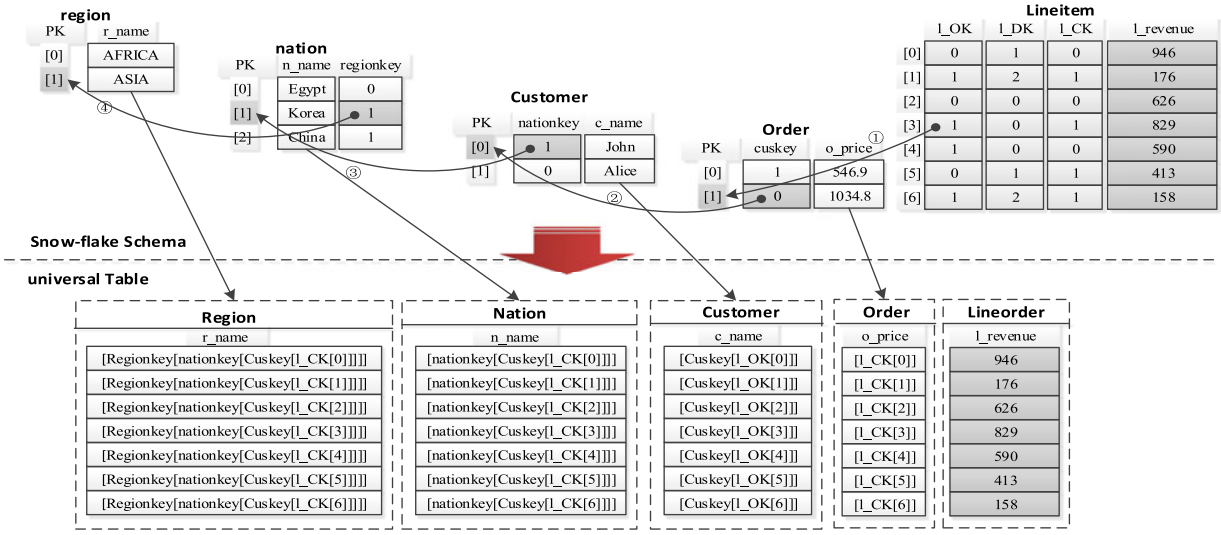


Fig. 3. A universal table for a snowflake schema.

To execute a SPJGA query, we can first denormalize the concerned relations into a single table and then apply the various operations, such as selection, projection, grouping and aggregation on the table. We call the result of denormalization *universal table*, which is the key concept of our query processing model.

Universal Table. Given a SPJGA query Q, we reserve only the join operations of Q and truncate all the other operations. The result of the remaining query, which contains only joins, is the universal table of Q.

Consider the following SPJGA query based on the star schema in Fig. 2.

```
Q1: SELECT c_nation, s_nation, d_year,
          sum(lo_revenue) as revenue
    FROM customer, lineorder, supplier, date
    WHERE lo_custkey = c_custkey
          AND lo_suppkey = s_suppkey
          AND lo_orderdate = d_datekey
          AND c_region = 'ASIA'
          AND s_region = 'ASIA'
          AND d_year> = 1992
          AND d_year< = 1997
    GROUP BY c_nation, s_nation, d_year
    ORDER BY y d_yearasc, revenue desc;
```

Its universal table is exactly the universal relation shown at the bottom of Fig. 2, which is the join result of the subquery in the black rectangle of Q1. (Note that in Fig. 2 some fields are not filled with real values but references to the original arrays, which indicate how we obtain the values through array address references.)

A-Store executes a SPJGA query by transforming it into a SPGA query on its universal table. During the execution, it simply scans the universal table, filters out the tuples that do not satisfy the selection criteria, and feeds the resulting records to grouping and aggregation operators. For instance, Q1 will be transformed into the following Q2 upon its execution.

```
Q2: SELECT c_nation, s_nation, d_year,
          sum(lo_revenue) as revenue
    FROM universal_table WHERE c_region = 'ASIA'
    AND s_region = 'ASIA'
    AND d_year> = 1992
    AND d_year< = 1997
    GROUP BY c_nation, s_nation, d_year
    ORDER BY d_yearasc, revenue desc;
```

As another example, consider the following query based on the snowflake schema shown at the top of Fig. 3 (an adaption of the TPC-H schema). Its universal table is the relation shown at the bottom of Fig. 3.

```
Q3: SELECT n_name, sum(l_extendedprice *
          (1 - l_discount)) as revenue
    FROM customer, lineitem, order, nation, region
    WHERE c_custkey = o_custkey
          AND o_orderkey = l_orderkey
          AND c_nationkey = n_nationkey
          AND n_regionkey = r_regionkey
          AND r_name = 'ASIA'
          AND o_price> = 800
    GROUP BY n_name
    ORDER BY revenue desc;
```

In A-Store, every query is executed through a scan of its universal table. Most importantly, A-Store never materializes the universal table before the scan. As mentioned previously, the array index references have already linked all the tables together, forming a virtual denormalization. A-Store can perform the scan directly on the virtual universal table.

A universal table is formed by joining multiple tables using their foreign keys. The structure of join can be modeled as a directed graph, where the vertexes represent the tables and the edges represent the array index references, i.e., foreign-key-to-primary-key relationships. We call it a *join graph*. A vertex without incoming edges is known as a root of the join graph. We call its corresponding table a *root table*. For a typical OLAP query on a star/snowflake schema, there is normally only one *root table*, which is exactly the fact table. We call the other tables *leaf tables*. In a star or snowflake schema, *leaf tables* are dimension tables.

It is obvious that each leaf table can be reached from the root table through a chain of array index references. For instance, the leaf tables in Fig. 3 can be reached from the root table through the following reference paths:

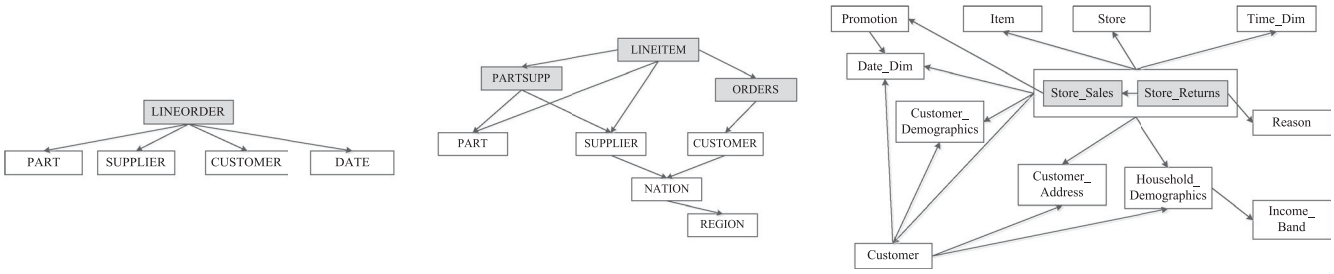


Fig. 4. Join graphs of widely used benchmarks.

- *lineitem → order*
- *lineitem → order → customer*
- *lineitem → order → customer → nation*
- *lineitem → order → customer → nation → region*

To scan a *universal table*, A-Store just needs to scan the root table and simultaneously follow the reference paths to fetch the tuples in the leaf tables (as illustrated by the solid edges in Fig. 3). As the array index references allow us to locate the tuples in leaf tables through simple array lookups, the whole scan process is efficient. During the scan, each tuple is evaluated against the selection criteria. Only the tuples satisfying the criteria are selected and fed to the grouping and aggregation operators.

Thus, an arbitrary SPJGA query can always be processed through the following three phases:

(1) **Scan and Filter**: Scan the universal table and identify the tuples satisfying the selection predicates;
(2) **Grouping**: Add each tuple identified by the first phase to a group based on the grouping conditions;
(3) **Aggregation**: Perform incremental aggregation on each group. Sort is performed in the end to handle the order-by clause.

For a fully materialized denormalization, the scan and filtering process will be fast, as sequential scan is cache efficient. However, this may not apply to *virtual denormalization*. As data accesses based on array index references are randomized, it may incur a large number of cache misses. This problem can be alleviated by the optimization techniques introduced in Section 4.

The query processing model of A-Store can be applied to most of the queries in the commonly used benchmarks, such as SSB, TPC-H, TPC-DS, whose schema graphs aremostly single rooted, as illustrated in Fig. 4. These benchmarks include a number of nested queries, which need to process the root table recursively. To deal with such nested queries, we can decompose the join graph into multiple single rooted subgraphs; then the subgraphs can be pipelined and processed seperately.

4 Optimizations

This section introduces three important optimization techniques adopted by A-Store. Two of them are intended to accelerate the scan of universal table, which is the most costly operation in A-Store. The other is intended to optimize the performance of aggregation.

4.1 Vector Based Column Scan

There are two basic approaches to scan a universal table—row-wise scan and column-wise scan. In a row-wise scan, the tuples of a universal table are fetched one after another and evaluated against the selection predicates. As A-Store uses an array-oriented storage model, row-wise scan is not the optimal solution, as it involves a large number random memory accesses across arrays.

In a column-wise scan, we only need to access the columns required in query processing. There are usually three types of columns to be accessed. The first type is the columns required by predicate processing. By scanning the column, we obtain the tuples that satisfy the selection criteria of a query. We call them selection columns. The second type is the columns used for grouping (following the group-by clause). We call them grouping columns. The third type is the columns participating in aggregation. We call them measure columns. As mentioned earlier, the query processing model of A-Store is composed of a scan-and-filter phase, a grouping phase and an aggregation phase. Therefore, the scans of the three types of columns should be conducted in the three phases respectively.

A column-wise scan of the selection columns can be performed in several ways. Some systems choose to scan and evaluate each column independently. The result of each scan is a bitmap [10], where each bit indicates whether the corresponding field in the column satisfies the selection predicates. Then the scan results of all the columns are combined through bitwise AND to determine the final tuples participating in the grouping and aggregation phases. As it requires each column to be completely scanned, it would usually consume a lot of memory bandwidth.

A-Store applies vector based column-wise scan, which aims to minimize the consumption of memory bandwidth. A *selection vector* is used to record the ids of the tuples satisfying the selection predicates. The vector is updated after the evaluation of each column. A tuple that has not passed any predicate evaluation is immediately removed from the *selection vector*, and will not be evaluated again.

Using vector based column-wise scan, a significant part of the universal table can be safely skipped in predicate evaluation. It minimizes the consumption of memory bandwidth as well as the cost for materializing intermediate bitmaps. As the most selective predicates can be evaluated first, this effect is maximized. Selection vector based predicate processing is a widely adopted mechanism in many column stores, such as MonetDB. In A-Store, this technique

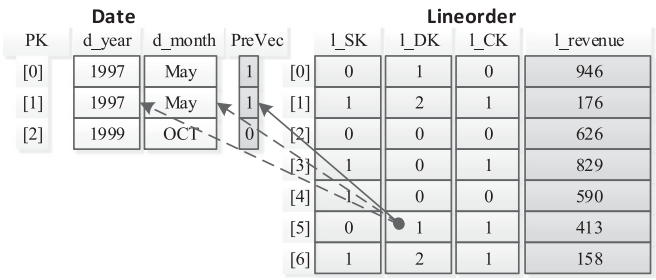


Fig. 5. Predicate vector optimization.

can not only save memory bandwidth, but also reduce the random memory accesses based on AIR.

4.2 Predicate Filter

As described in Section 3, query processing in A-Store does not require a fully materialized denormalization. As a price, a scan of a virtual universal table usually involves a lot of repeated random array lookups. For instance, Fig. 5 depicts a virtual universal table consisting of a root table *Lineorder* and a leaf table *Date*. To scan an original column of the root table, such as *l_revenue*, we only need to perform a simple scan without incurring any random data access. In contrast, to scan a column originated from a leaf table, such as *d_month* in the *Date* table, we need to perform a scan of the foreign key column *l_DK* and lookup the leaf column *d_month* repeatedly using array index references. Such repetition is a waste of CPU cycles, especially for complex predicate evaluation, such as *strcmp*(). When the predicates on a dimension table involve multiple attributes, the lookups will incur a large number of random memory accesses too.

A-Store applies predicate filter to eliminate repeated evaluation of leaf tables. It first conducts predicate evaluation directly on the leaf tables and generates a bit vector for each leaf table. In a bit vector, "1" indicates that the corresponding tuple satisfies the selection predicates, and "0" the other way round. Such a bit vector is called a predicate vector. When scanning the universal table, we do not lookup the leaf tables, but probe the predicate vectors to determine if a tuple satisfies the selection criteria. The PreVec in Fig. 5 is an example of a predicate vector. Suppose the predicate on Date is "*d_year* = 1997 and *d_month* = 'May'". In predicate evaluation, we find that only the first two tuples satisfy the predicates. Thus, we obtain the predicate filter <1,1,0>. As a result, the scan of the *d_year* and *d_month* columns in the universal table can be reduced to a single scan of the foreign key *l_DK* and several random accesses on *PreVec*. As a predicate vector is usually small enough to fit in CPU caches(for instance, Intel Xeon Processor E7–8890 v3 has a LLC of 45 MB, which can accommodate a predicate vector of 377 million bits, which is sufficient for most real world dimension tables), this approach can help to improve cache efficiency significantly.

For a snowflake schema (see Fig. 3), a reference path can involve a chain of leaf tables. Then, predicate filters can be generated recursively for the leaf tables on the chain. In the end, a single predicate filter can be generated for the entire chain—the length of a predicate filter is determined by the number of rows of the first level dimension. If the predicate filter of a leaf table is too big to fit in CPU Caches, the effect of using predicate filter will diminish. In this case, A-Store can choose not to use predicate filters, but resort to the original probing approach. For instance, in Fig. 3, the *order* table of TPC-H can be big. When scanning the universal table in Fig. 3, A-Store can choose to generate a predicate filter only for the dimension tables *customer*, *nation* and *region*, and probe the *order* table directly during the scan. An optimizer is used to decide whether to use predicate vectors, according to the row number of each table.

4.3 Array Based Column-Wise Aggregation

Traditional OLAP engines usually perform hash based grouping and aggregation. Basically, a hash table is used for storing aggregation results. The grouping attributes are used as the hash key. Given a tuple that has passed the predicate evaluation, the system identifies its group using the hash table and integrates the tuple's measure attributes into the aggregation result of its group. Such a hash based method is particularly suitable for row-wise query processing, as a pipeline can be established to connect the scan, grouping and aggregation stages. By contrast, A-Store applies column-wise aggregation. It chooses to use a multidimensional array instead of a hash table to collect aggregation results. Furthermore, aggregation is processed in two independent phases—a grouping phase and an aggregation phase. For selective queries, the grouping phase can filter out most tuples to reduce useless memory accesses on the measure columns.

In query processing, A-Store first scans the selection columns to identify the tuples that should participate in aggregation. Then, it proceeds to scan the grouping columns to create a multidimensional array for storing aggregation results. Finally, the measure columns are scanned to finish the aggregation. Each element of the multidimensional array corresponds to a group in aggregation. During the scan of the grouping columns, the array index (i.e., a multidimensional array indexes like $agg[x][y][z]$) of each tuple's group will be identified and stored in a *Measure Index*. A *Measure Index* is a vector associated with the fact table (as shown in Fig. 6) to identify the multidimensional array index of each fact tuple (–1 for fact tuples which failed the predicate evaluation).

A *Measure Index* can also be used as a selection vector for the measure columns. As the addressing mechanism of arrays is faster than that of hash tables, our array based aggregation can outperform hash based aggregation remarkably.

The procedure for scanning grouping columns is illustrated in Fig. 6. In most cases, grouping columns are located in leaf tables. Thus, when we use the leaf tables to generate the predicate filters, we generate a set of group vectors as well. A group vector is used to determine the group each tuple belongs to. As shown in Fig. 6, during the scan of the leaf table customer, a group vector is created based on the grouping column *c_nation*. As we can see, the 2nd and 3rd tuples belongs to the "Canada" and "Brazil" groups respectively. Note that if a tuple does not pass the predicate evaluation, such as the 1st and the 4th tuples, its value in the group vector is marked as *null*.

Afterwards, dictionary compression is applied to encode each group vector. Basically, a dictionary array is used to store the group IDs. In the compressed vector, the *null* value

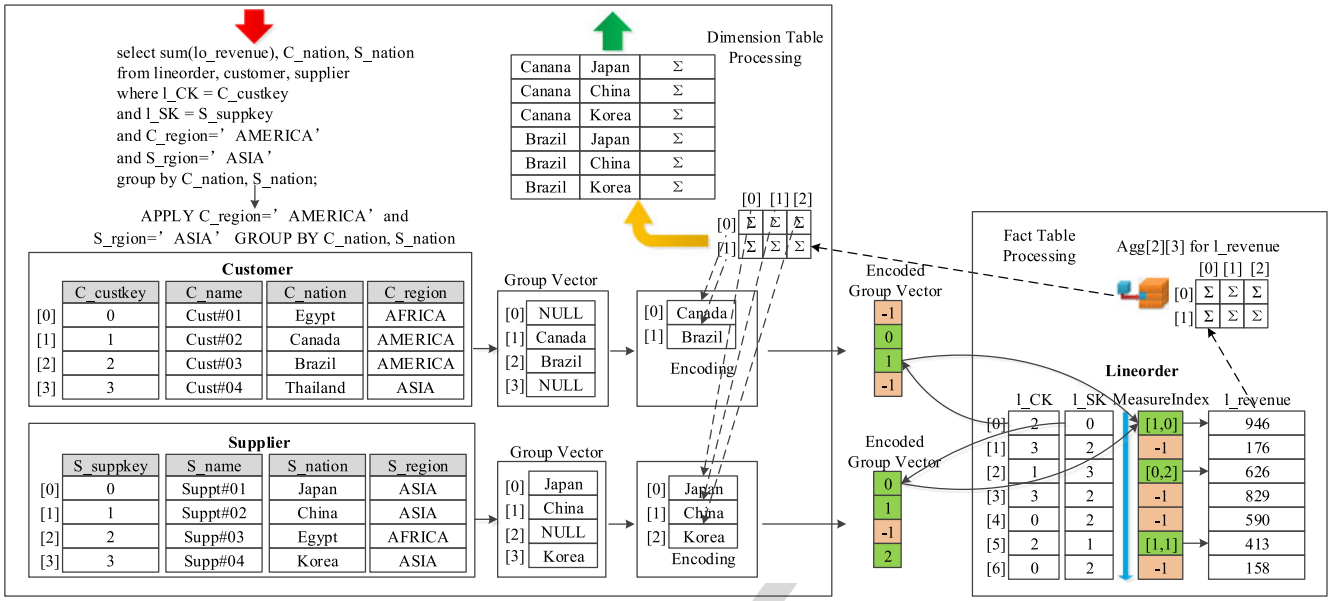


Fig. 6. Multidimensional array oriented aggregation.

is encoded as "–1" and the group IDs are encoded as the array indexes of the dictionary. Please note that the size of the dictionary array is proportional to the number of the distinct values in the group column. It is usually small in size. When multiple grouping columns from different leaf tables are involved in a query, multiple group vectors will be generated. Then, a multidimensional array is created for storing the aggregation results. As shown in the top-right of Fig. 6, each dimension of the array corresponds to the dictionary of a group vector.

After processing the group columns in the leaf tables, the system proceeds to scan their reference columns in the root table. During the scan, the system probes the group vectors to identify the group each root tuple belongs to, and then obtains the index of that group in the aggregation array. This procedure is illustrated in Fig. 6. A *Measure Index* is used as the *selection vector* for fact columns. For instance, we retrieve the first field of the foreign-key column *l_CK* and get the value 2, which indicates that the third element in customer's group vector contains the group ID of the tuple. This group ID turns out to be 1. Then, we know that the index of the group's aggregation value in the aggregation array must be [1,x]. By probing the foreign-key column *l_SK*, the same procedure leads us to the group ID 0. Thus, we obtain the complete index of its group's aggregation value, which is [1,0].

After the scan of the grouping columns, we obtain a complete array index that links the tuple of the universal table to the aggregation values in the aggregation array. In the aggregation phase, we simply scan the measure columns, and add their values to the aggregation array using the *Measure Index*. Again, as a *Measure Index* is used, only the parts of the measure columns referred by the *Measure Index* need to be accessed. Please note that the dimensionality of the aggregation array can be further reduced if there are functional dependencies among the grouping columns. Due to limited space, we do not go into the details of such scenarios.

Although positional lookup has been widely used by modern query processors, such as MonetDB [2] and invisible-join [10], there is a major difference between A-Store and the existing approaches—A-Store pre-constructs a multidimensional array according to the *Group By* clause, instead of dynamiccally constructing a hash table through repeated grouping attributes probing.

Array based column-wise aggregation utilizes array index reference to reduce the cost of data location. Through column-wise scan, it can also achieve better locality of memory access. However, such an array based approach cannot be applied to every query. When a query involves a large number of grouping columns, the resulting aggregation array can be too sparse and consumes overly large RAM space. As an option, we can use *Measure Index* as hash keys and perform hash based aggregation instead of array based aggregation. For instance, some queries in TPC-H, such as Q3, fall in this category. In such an extreme case, a hash table instead of an array will be used to store the aggregation results. The optimizer of A-Store is responsible for estimating the sparsity of aggregation arrays and deciding whether to use array based or hash based aggregation.

The LLC of a modern CPU is already quite big (which can store millions of array cells). It usually can accommodate the entire aggregation array. According to our evaluation, the multidimensional array of most OLAP queries in SSB, TPC-H and TPC-DS is smaller than an ordinary LLC. While array based aggregation is not a one-size-fits-all solution, it is especially suitable for interactive OLAP, whose resultsets should be small enough for end users to consume.

4.4 Handling Updates

A-Store is highly optimized for OLAP workload and read intensive applications. It is not necessary or realistic to expect A-Store to be as fast as a conventional DBMS in processing OLTP workloads. Nevertheless, the ability to handle updates is mandatory, especially when deployed for real-time analytics, which imposes strict requirements on not only response

TABLE 1
Update Mechanism Comparison

	Insertion	Deleteion	Update
A-store	Append	Deletion vector and slot reuse	In-place updating
MonetDB	Append	Deletion vector	Out-of-place updating
Vectorwise	Append	Deletion vector	Out-of-place updating
Hyper	Append	Deletion vector	Copy-on-write updating

time but also freshness of results. In this section, we sketch a mechanism A-Store can employ to handle updates efficiently.

To enable A-Store to handle both updates and OLAP queries simultaneously, multi-version concurrency control (MVCC) can be applied. The simplest MVCC mechanism is the one adopted by Hyper [11], which utilizes a *copy-on-write* mechanism of the operating system to isolate OLTP and OLAP workloads. A-Store can adopt the same mechanism. As virtual denormalization is used, modification of a table may result in cascade modification of its referenced tables. To minimize such cascade effects, A-Store applies a lazy-deletion approach. Specifically, an additional bit vector is maintained for each table, to record whether each tuple is up-to-date. When a tuple is deleted, the corresponding bit is simply set to out-of-date, without incurring further modification. When dealing with OLAP queries, A-Store uses the bit vector to filter out the tuples that are out of date.

Insertion. Insertion in A-Store is conducted through appending—new fields are normally added to the end of the column. A-Store preserves a certain proportion of free space at the end of each array, such that insertion does not always require allocation of new memory space. When the space of an array is used up, new space needs to be allocated and concatenated to the end of the array. To ensure efficiency, the page table of the virtual memory can be directly modified to accomplish the concatenation.

Deletion. When a tuple is deleted from a table, the corresponding bit in the bit vector is simply marked as out-of-date. The position of a deleted tuple will later be reused by a newly inserted tuple. Sometimes, intensive deletion may leave too many holes in the arrays, so as to affect the system performance. Therefore, A-Store allows users to consolidate a table by resorting its primary keys. Consolidation is an expensive operation, as it has to update all the references to the table. Therefore, consolidation should be performed only when the system is idle.

Update. A-Store applies in-place updating, so it can avoid modifying foreign keys. As A-Store uses dynamic space to store columns of variable length, e.g., *varchar*, it makes in-place updating possible.

Therefore, virtual denormalization does not incur much extra cost when handling updates. The main extra cost occurs in the phase of consolidation, caused by intensive deletion. In a typical data warehousing scenario, we normally do not perform deletion frequently. We rarely delete fact tuples, except periodically migrating the data to backend storage. We normally do not delete dimensional tuples either, due to the reference constraint on foreign keys. Therefore, the cost of deletion is not a major concern of a data warehouse.

Comparison with other MMDBs: The insertion mechanism of A-Store is similar to that of MonetDB and Vectorwise. The deletion mechanism of A-Store is an extention of that of MonetDB and Vectorwise, which uses deletion vectors to mark deleted tuples. In contrast to MonetDB, A-Store allows reuse of the slots of deleted tuples to make dimension tables compact. The reuse mechanism is enabled by the use of surrogate key (array index), which has no semantic meaning. As to consolidation, A-Store is less efficient than MonetDB and A-Store, as it needs to update the foreign keys too. Fortunately, consolidation is rarely used in common practice. The following table summaries the difference.

5 Multicore Parallelization

Our implementation of A-Store fully considered modern multicore processors, for which parallelization is an important issue. To parallelize query processing, we logically partition the universal table (i.e., the fact table) horizontally.

As shown in Fig. 7, each partition is assigned to a worker thread. As the predicate filters are shared by all the partitions during the query processing, we centralize the

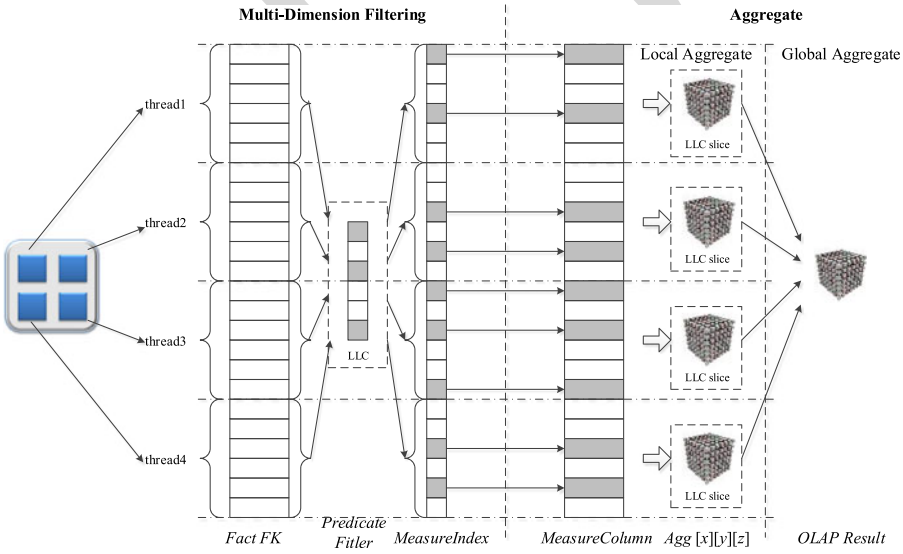


Fig. 7. A-store implementation of multicore parallel OLAP.

TABLE 2
Comparison of AIR Against NPO and PRO Hash Join Algorithms

Join tables and table sizes	NPO	PRO	AIR
	Cycles/tuple		
SSB(SF = 100)			
Lineorder⋈date 600000000:2555	1.02	5.13	0.62
Lineorder⋈part 600000000:1528771	6.27	5.27	1.00
Lineorder⋈supplier 600000000:200000	1.87	5.42	0.63
Lineorder⋈customer 600000000:3000000	9.94	5.41	1.06
TPC-H(SF = 100)			
Lineitem⋈part 600000000:20000000	14.28	5.38	2.04
lineitem⋈supplier 600000000:1000000	2.04	5.31	0.94
orders ⋈ customer 150000000:15000000	15.67	6.43	1.83
lineitem⋈ order 600000000:150000000	24.03	7.05	3.76
TPC-DS(SF = 100)			
Store_sales⋈store 287997024:402	0.88	4.95	0.64
Store_sales⋈date_dim 287997024:73094	1.39	5.45	0.62
Store_sales⋈time_dim 287997024:86400	1.68	5.25	0.63
Store_sales⋈household_demigraphics 287997024:7200	1.03	5.29	0.63
Store_sales⋈customer_demigraphics 287997024:1920800	6.64	5.51	1.03
Store_sales⋈customer 287997024:2000000	6.75	5.63	1.04
Store_sales⋈item 287997024:204000	1.86	5.31	0.64
Store_sales⋈promotion 287997024:1000	0.81	4.97	0.63
Store_sales⋈store_return 287997024:28795080	16.71	6.07	2.54
workloads in [7]			
Workload A 268435456:16777216	15.56	11.88	2.23
Workload B 128000000: 128000000	38.38	11.71	4.02

evaluation of the leaf tables. Once the predicate filters are generated, they will be shared by all worker threads, which will continue with the query processing independently.

The intermediate results generated by each worker, such as the *Measure Index* and the group vectors are used exclusively by the worker itself. The parallelization terminates after each worker generates a multidimensional aggregation array out of its own partition. In the end, the multidimensional arrays are integrated into the final result. In fact, the query processing model of A-Store is intrinsically easy to be parallelized, as each horizontal partition can be processed independently. To ensure load balance, A-Store always allocates more worker threads than the available physical threads on the computing platform, such that all physical threads can be saturated.

6 Experiments

To obtain an objective profile of A-Store's performance, we compared it against several modern analytical databases whose design and implementation are highly optimized for main memory and modern multicore processors. They include MonetDB (Version 11.15.19), Vectorwise (Version 2.5.2) and Hyper [12]. While MonetDB and Vectorwise are main-memory optimized databases instead of fully memory resident database, we preload the entire data into their memory space to eliminate I/O latency. To achieve it, we execute each query for 3 times, and use the shortest execution time as the final execution time.

Furthermore, we optimized Vectorwise with the "optimized mydatabase" command to achieve the optimal performance. For some queries, the performance can be improved by several times. MonetDB and Hyper do not provide any configuration option, as they are supposed to be self-tuned.

Our experimental evaluation were conducted on an HP Z820 workstation, which was equipped with two 2.60 GHz Intel Xeon processors E5–2670 (each with eight cores and a 20 MB L3 Cache), and 256 GB DDR3 RAM. The operational system is CentOS Linux version 6.5.

6.1 Micro Benchmark Testing

We first set up some micro Benchmarks to evaluate the performance of the key operators.

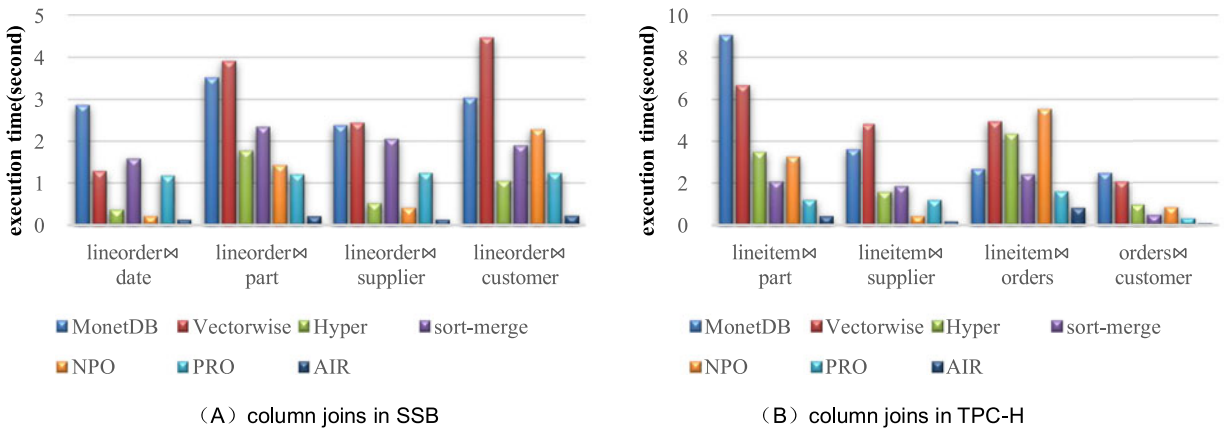
6.1.1 Join Operator

Our microbenchmark consists of a number of join operations selected from the benchmarks of SSB, TPC-H and TPC-DS. Table 2 shows the results of our experiments. We can see that NPO outperforms PRO in joins with small dimension tables (with maximum number of rows up to 1,000,000). This is expected, because the shared hash table can fit in the LLC when dimension tables are small. Although AIR's CPU cost grows as the size of the dimension table increases, it outperforms PRO and NPO in all cases, because it only needs to perform positional lookups.

6.1.2 Join Operatoins in SSB and TPC-H

To evaluate the join performance of the candidate MMDBs. We designed a number of column join queries on SSB and TPC-H, which are all in the form of "select count(*) from *TableA*, *TableB* where *TableA.foreignkey* = *TableB.primarykey*;".

We also included several state-of-the-art join algorithms, such as hash join (NPO and PRO) [7] and sort-merge join [13] in the experiments. The results are shown in Fig. 8. As we can see, MonetDB outperforms Vectorwise in five join queries, and Hyper outperforms MonetDB and Vectorwise in seven join queries. Hyper has similar performance as the



（A）column joins in SSB

（B）column joins in TPC-H

Fig. 8. Foreign key-Primary key column joins for SSB and TPC-H (SF = 100).

hand-code join algorithms. Among these systems and algorithms, AIR shows similar performance as NPO on join queries with small dimensions (e.g., *date* and *supplier)*. For join queries with large dimensions, AIR is much more efficient than the others, owing to its compact array store and array index referencing mechanism.

6.1.3 Key OLAP Operators in SSB

A typical OLAP query needs to invoke operators that are responsible for predicate processing, star-join, grouping and aggregation, respectively. Table 3 gives the results on a micro-benchmark adapted from SSB (SF = 100), which can assess the efficiency of different operators.

We used four columns from the fact table to evaluate the performance of predicate processing. The selectivity of the predicates on the four columns varied from $(1/2)^4$ to $(1/16)^4$. As we can see, the C++ implemented A-store achieved similar code efficiency as Hyper's JIT compliant implementation.(Hyper is slightly faster than A-store, for it conducts optimization at the register level.) A-store is 2–3 times faster than Vectorwise due to the simplicity of its query plan. MonetDB is slow in predicate processing, probably because MonetDB uses BAT.join() instead selection vector to integrate multiple results of predicate processing.

We use the query "select count(*), lo_discount, lo_tax from lineorder group by lo_discount, lo_tax;" to measure the performance of *group-by* operators (the results contain 99 groups). For this query, the aggregation results can always fit in the LLC, no matter we use hash based or array based aggregation. A-store outperforms the other three MMDBs as it uses less CPU cycles in aggregation bucket location.

To evaluate performance on Star-join, we simplified the SSB queries by using count(*) instead of other aggregation expression and eliminating all group-by clauses. As introduced previously, A-store employs AIR based scanning (with the help of predicate filters) to achieve the effect of star-join. In contrast, Hyper and Vectorwise employ pipelining star-join. As shown in Table 3, pipelining star-join performs well in queries with high selectivity, including Q1.1 (1.9%), Q2.1 (0.8%), Q3.1 (3.4%) and Q4.1 (1.6%). For the rest of the queries, A-Store outperforms the others. As dimension increases, the performance gap between A-store and Hyper increases.

TABLE 3
Key OLAP Operators in SSB

Type	Query Execution time(ms)	A-store	Hyper	Vectorwise	MonetDB
Predicate processing	(1/2)^4	635	245	1050	18700
	(1/4)^4	401	213	969	7200
	(1/8)^4	281	158	940	4000
	(1/16)^4	218	119	928	2900
	Grouping & Aggregate	249	280	311	3500
star join	Q1.1	306	222	750	6600
	Q1.2	117	168	694	6400
	Q1.3	97	166	1208	6200
	Q2.1	306	337	1667	1000
	Q2.2	249	404	1308	654
	Q2.3	236	218	929	514
	Q3.1	489	676	1988	1700
	Q3.2	305	278	1577	710
	Q3.3	127	255	1132	494
	Q3.4	125	257	1133	493
	Q4.1	473	881	2414	3600
	Q4.2	503	792	2194	2800
	Q4.3	309	345	1582	2100
	AVG	280	384	1429	2559

6.2 Benchmark Performance

To evaluate the performance of entire OLAP engines, we conducted experiments on full scale SSB. In the experiments, we tested A-store, the three MMDBs and their denormalization versions.

6.2.1 Effects of Denormalization

Denormalization removes the costly join operation from OLAP queries. Our first set of experiments was intended to measure how MonetDB, Vectorwise and Hyper can benefit from denormalization. We generated a denormalized SSB table for each MMDB and rewrote all SSB queries for the denormalized table. As shown in Fig. 1, except MonetDB, all the systems benefit from denormalization. Table 4 shows the break-down of the execution time on predicate processing and grouping&aggregation. As we can see, Hyper is 2–3 times faster than Vectorwise in predicate processing, and slightly faster than Vectorwise in grouping&aggregation. Similar to the results in Table 3, MonetDB is much slower in predicate processing than the others. Its

TABLE 4
Predicate Processing, Agg-Groupby on Denormalization Performance Comparision

Query Execution time(s)	Predicate processing on denormalization			Grouping&Aggerate on denormalization		
	MonetDB	Vectorwise	Hyper	MonetDB	Vectorwise	Hyper
Q1.1	21.61	0.35	0.17	0.65	0.20	0.14
Q1.2	24.87	0.65	0.23	0.69	0.20	0.13
Q1.3	23.05	0.39	0.12	0.67	0.42	0.14
Q2.1	23.30	0.53	0.35	3.34	4.63	5.25
Q2.2	18.87	1.66	0.35	3.26	4.58	5.25
Q2.3	23.27	1.11	0.35	3.29	4.53	5.29
Q3.1	23.27	1.15	0.41	4.88	8.16	6.07
Q3.2	59.31	0.81	0.32	61.02	9.12	7.84
Q3.3	57.82	0.84	0.45	60.74	9.33	7.49
Q3.4	6.41	0.55	0.23	56.48	10.45	8.01
Q4.1	24.27	1.71	0.45	3.05	4.82	1.62
Q4.2	23.48	1.31	0.48	5.28	7.21	5.97
Q4.3	11.34	0.60	0.38	180.85	15.40	8.47
AVG	**26.22**	**0.90**	**0.33**	**29.55**	**6.08**	**4.74**

overheads on grouping is also high for some of the queries. This might be due to the BAT algebra whose query plans are not easy to optimize.

6.2.2 Testing on SSB

In the experiments on SSB, we applied all the optimization techniques of A-Store, as it is supposed to achieve the best performance. The parallelism degree of A-Store was set to 32 threads.

Table 5 shows the results of the experiments on SSB. Similar to previous results, the denormalized version of MonetDB does not perform well, due to its overheads in predicate processing and grouping. For Vectorwise and Hyper, denormalization improves OLAP performance by 27% (Vectorwise) and 15% (Hyper) respectively, though it consumes a significant amount of additional memory. Compared to denormalized Hyper, our hand-code denormalization implemention is able to achieve a 50% reduction of execution time .This improvement is mainly due to the application of array aggregation, selection vector and dictionary compression.

A-store (virtual denormalization) is slightly slower than real denormalization, while it is significantly faster than all the other MMDBs and their denormalized versions. The main performance penalty for A-Store comes from randomized accesses to predicate filters which could occasionally incur cache thrashing. As we can see, for queries involving limited number of joins, such as Q1.1, Q1.2 and Q1.3, A-Store could even outperform the real denormalization approach. This is because the predicate filter on the small dimension tables, such as the date table (255 rows), can readily fit in the L1 cache. In general, cache thrashing does not seem to be a severe issue for A-Store, since predicate filters are usually small. Moreover, the advantage of real denormalization is at the price of extra RAM space. In our experiments, the real denormalization approach consumed more than five times as much space as that of A-Store (262.08 GB versus 45.82 GB). Thus, virtual denormalization seems to be able to achieve a good trade-off between space consumption and performance.

Virtual denormalization enables A-Store to greatly improve join performance, as indicated in Table 5. Compared with normal aggregation methods, the array based aggregation approach adopted by A-Store also proves to be more efficient than hash based approaches. For instance, by looking into the query execution traces of MonetDB, we found that the majority of its time was spent on join and

TABLE 5
Star Schema Benchmark Performance of Different Database Implementions

Query Execution time(s)	MonetDB_D	MonetDB	Vectorwise_D	Vectorwise	Hyper_D	Hyper	A-Store	Denormalization
Q1.1	22.19	5.91	0.38	0.75	0.21	0.25	0.39	0.43
Q1.2	24.59	5.90	0.66	0.64	0.23	0.16	0.12	0.12
Q1.3	23.27	5.77	0.44	0.61	0.12	0.16	0.10	0.14
Q2.1	24.00	1.18	1.44	1.88	0.62	0.48	0.46	0.35
Q2.2	18.98	0.88	1.81	1.45	0.36	0.59	0.31	0.20
Q2.3	23.53	0.66	1.18	1.04	0.37	0.26	0.26	0.10
Q3.1	74.70	2.00	1.88	2.48	0.90	0.93	0.60	0.36
Q3.2	60.77	0.95	0.88	1.74	0.39	0.40	0.25	0.12
Q3.3	57.21	0.69	0.85	1.40	0.43	0.29	0.15	0.10
Q3.4	5.87	0.59	0.66	1.30	0.23	0.29	0.15	0.10
Q4.1	25.94	3.25	1.80	2.77	0.58	1.00	0.58	0.35
Q4.2	24.34	3.14	1.84	2.94	0.52	0.93	0.53	0.24
Q4.3	11.56	1.31	1.79	2.07	0.44	0.46	0.34	0.12
AVG	**30.53**	**2.48**	**1.20**	**1.62**	**0.41**	**0.48**	**0.32**	**0.21**

TABLE 6
Query Processors of A-Store to be Evaluated

Algorithm	row-wise scan or vector based column-wise scan	use of predicate vector	array oriented column-wise aggregation
AIRScan_R	row-wise	no	no
AIRScan_R_P	row-wise	yes	no
AIRScan_C	column-wise	no	no
AIRScan_C_P	column-wise	yes	no
AIRScan_C_P_G	column-wise	yes	yes

aggregation. According to the literature [14], [15], Vectorwise is supposed to outperform MonetDB significantly, as it applies pipelining to avoid intermediate result materialization. However, the advantage of Vectorwise does not show clearly on SSB. As SSB queries are normally very selective and their joins are always between a big table and several small tables, it appears that the intermediate results generated by MonetDB are small in size. MonetDB performed poorly on the quries of Q1, due to the overheads of predicate processing on the fact table. On other query groups, MonetDB has similar performance as Vectorwise. Hyper is much faster than MonetDB and Vectorwise, for it uses a JIT compilation technique. As A-store applies a common query execution plan for all queries, it has similar code efficiency as Hyper. Its AIR mechanism proved to be faster than traditional hash join employed by Hyper.

6.3 Detailed Performance Analysis

To obtain detailed characteristics of A-Store, we evaluated 5 variations of A-Store, which applied different optimization methods. The purpose is also to illustrate how to step by step improve OLAP performance with various optimization techniques. The difference of the 5 query processors is summarized in Table 6.

All the five processors used our array oriented storage model and our query processing model based on AIR. Thus, they are all marked with AIRScan. AIRScan_R scans the virtually denormalized table (with dictionary compressed columns) in a row-wise manner. AIRScan_C_P_G applies all the three optimization techniques introduced in Section 4. The other three query processors selectively apply a subset of the optimization techniques. The experiments were conducted on SSB. The degree of parallelism was set to 32 threads.

Fig. 9 shows the detailed query execution time for the five variations of A-Store, as well as that of MonetDB, Vectorwise and Hyper. As shown in the first column of Fig. 9, except row-wise scan (AIRScan_R and AIRScan_R_P), the other three variations of A-Store all outperform the three MMDBs. Even with row-wise scan and simple dictionary compression optimization, A-Store can still outperform MonetDB and Vectorwise. Only Hyper outperforms the row-wise versions of A-Store. It is also clear that all the optimization techniques of Section 4 can improve the performance of A-Store. The application of predicate vector can reduce the average execution time from 752.68 to 675.49 ms (the result of AIRScan_R_P). The application of vector based column-wise scan can further reduce the execution time to 513.40 ms (the result of AIRScan_C_P). If all the optimization techniques are applied, the average execution time can be reduced to as low as 322.61 ms (the result of AIRScan_C_P_G).

6.4 Breakdown of Processing Time

By comparing the row-wise versions against the column-wise versions of A-Store, we find that column-wise scan is superior to row-wise scan for almost all testing queries. Especially for query with low selectivity, such as Q1.3, column-wise scan can outperform row-wise scan by several times. This is because the use of a selection vector allows the system to skip a significant proportion of the table during the scan. The array based aggregation seems to work much better than traditional hash based aggregation too. Its application immediately reduced the execution time by 60% (from 513.40 to 322.61 ms).

Fig. 10 shows the breakdown of the average query execution time over the three different stages of query processing. They are: 1) the stage of processing leaf table and generating predicate vectors and group vectors; 2) the stage of processing the foreign key columns and generating *Measure Index*; 3) the stage of scanning the measure columns and performing aggregation. We mainly considered the three variations in Table 6 that use column-wise scan. For the approach of

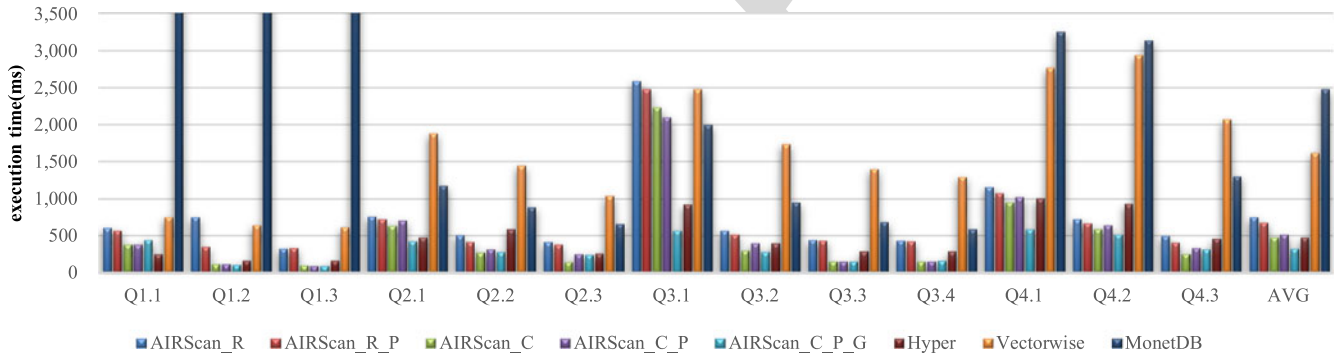


Fig. 9. Performance evaluaion for different variations of query processors of A-store.

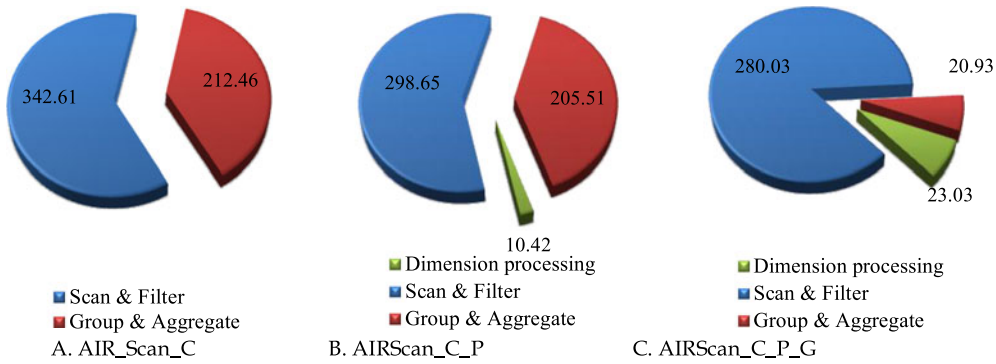


Fig. 10. Breakdown of processing time for the column oriented query processors.

AIRScan_C, as it does not use predicate filters or group vectors, its first stage is ignored. As we can see, the processing of the leaf tables is fast, as the leaf tables in a star schema are usually small. The predicate filters and group vectors generated by the process can bring substantial performance gain. We can also see that the array based aggregation methods can outperform the hash based methods by almost an order of magnitude.

7 Related Work

Main memory databases have been investigated for a long period since 1980's [16]. Due to the cost and the limited capacity of RAM, MMDB was mainly used for real-time transactional processing (OLTP) in the past. The well-known MMDBs dedicated for OLTP include Timesten [17], solidDB [18], eXtremeDB [19], etc. Driven by the advance of hardware technology, the capacity of main memory has undergone an exponential growth in recent years. Main memory has become spacious enough for OLAP applications. A number of analytical MMDBs thus emerged. MonetDB, Vectorwise and SAP HANA are the most representative ones.

The column-store pioneer, MonetDB [2], is the earliest prototype of analytical MMDBs. It first applied a column oriented storage model for in-memory data. Based on the model, a number of optimization techniques, such as cache-conscious hash join, BAT algebra and multicore parallelization were proposed to achieve superior performance. Eventually, several modern MMDBs adopted the column oriented approach of MonetDB and made further development on it. Some well known examples include Vectorwise [20], SAP HANA [21], [22], [23], Hyper [11], DB2 BLU [24], SQL Server Column Store [25], etc. A significant body of research work was dedicated to make column-wise query processing more efficient. Due to limited space, we do not elaborate all these results in this paper.

A-Store differs from the existing in-memory column store in several aspects. First and foremost, A-Store applies virtual denormalization to eliminate the cost of join, which is the most demanding operator in MMDB. Instead of resorting to sophisticated join algorithms, virtual denormalization treats array indexes as primary keys, such that it transforms a join operation into a scan-and-address operation, which is not only simple but also easily parallelizable. Second, virtual denormalization enables A-Store to apply a uniform query execution plan to most SJPGA queries, which largely reduces the burden of query optimizers and simplifies code complexity. Third, a number of optimization techniques are applied to further enhance the performance of A-Store. Most of the techniques utilize array indexes to accelerate data access. The techniques of column-wise scan and predicate filter have been adopted by other systems as well, though in different forms. The technique of array based column-wise aggregation is relatively new, and it can make significant improvement on performance.

Join index [26] is a common approach used to alleviate the cost of joins. In particular, bitmap join index is widely adopted in data warehouse products. However, the sole purpose of join index is to accelerate join with pre-generated (bitmap) join index. It is usually used as an auxiliary data structure, which does not affect the basic query processing framework. Updates on join index are usually quite expensive. By contrast, virtual denormalization represents a coherent query processing mechanism. It integrates the join information into the storage model itself. In other words, by using array indexes as the primary key, the foreign key columns in the fact table automatically act as join indexes. Therefore, it incurs minimized storage and maintainance cost. More importantly, our approach builds a set of query processing methods on top of the storage model, with the aim to achieve the best possible OLAP performance.

Denormalization is widely adopted by data warehouses to achieve improved performance [27], [28], [29], [30], while the space consumption and the redundancy it introduces always limit its application in real world. Most MMDBs normally do not consider denormalization, due to the expensiveness of RAM. Blink [31], [32] and WideTable [33] are two of the rare cases that applies real denormalization to MMDB. Blink is a row-wise OLAP engine, which aims to improve the scalability of MMDB on multi-core processors and large RAMs. It performs both vertical partition and horizontal partition of the tables to allow efficient ALU operations. The original version of Blink chooses to denormalize a database into a single table, to completely eliminate the cost of join. The later version, a.k.a. DB2 BLU [24], abandoned this strategy due to its excessive redundancy. WideTable also adopts fully materialized denormalization. It mainly applies dictionary compression to minimize redundancy. Both approaches show that denormalization can speedup query processing and utilize multicore platforms effectively. The design of A-Store shares the inspiration of Blink. As opposed to Blink and WideTable, the denormalization of A-Store is performed in a virtual way, such that issues raised by redundancy can be eased.

A-Store uses array indexes as key values. This approach helps to minimize data access latency in main memory. Similar approaches have been adopted by other in-memory systems. The most outstanding example is DBGraph [34]. By using physical RAM addresses as references, DBGraph can perform join by direct graph traversal. However, designed upon a graph model, DBGraph's query processing model is different from that of A-Store.

The principle of in-memory computing is to simplify the computation model for better CPU efficiency. Array is widely used by modern main memory databases to achieve good performance. Examples include CSB+-Tree, CST-tree, BAT storage model in MonetDB, etc. Therefore, an array oriented MMDB may get the best out of in-memory computing. Moreover, an array based OLAP engine is adaptive to morden coprocessor platform, such as GPU and Intel Xeon Phi. A-Store is designed to be such an array oriented database.

8 Conclusion

In this paper, we present A-Store, which applies the strategy of denormalization to accelerate analytical query processing in main memory. A-Store can also be regarded as a specialized system for multidimensional model and SPJGA queries, which are the most common model and queries in OLAP. A-Store adopts an array oriented storage model. Its query processing model is based on a simple scan and filtering process. The simplicity of A-Store allows it to achieve high CPU efficiency and good parallelizability, as proven by our experimental study. Through A-Store, we show that denormalization can be a good optimization strategy for main memory databases. It is highly suitable for today's multicore platforms. We also learned the following rationales for designing efficient in-memory data processors. First, main memory is not merely a faster storage device; its efficient addressing mechanism should be fully considered in system design. Second, a one-size-fit-all design will sometimes make database more complicated and result in low instruction efficiency; instead, by designing a specialized query processing model, e.g., by using virtual denormalization, we could achieve enhanced performance.

Acknowledgments

Xuan Zhou is the corresponding author. This work was partially supported by the National High-tech R&D Program (863 Program) (2015AA015307) and the NSFC Porject (No. 61272138), the Basic Research funds in Renmin University of China from the central government (12XNQ072, 13XNLF01), a grant from HUAWEI Technologies CO., Ltd (HIRP OPEN 20140510). The authors would like to thank Xiaoli Liu and Huijun Liu from HUAWEI for project cooperations.

References

[1] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *Proc. Int. Conf. Manage. Data*, Jun. 2013, pp. 1243–1254.

[2] P. A. Boncz, "Monet: A next-generation DBMS kernel for query-intensive applications," Ph.D. dissertation, Dept. Electr. Eng., Universiteit van Amsterdam, CWI, Amsterdam, The Netherland, 2002.

[3] M. Zukowski, "Balancing vectorized query execution with bandwidth-optimized storage," Ph.D. dissertation, Dept. Electr. Eng., Universiteit van Amsterdam, CWI, Amsterdam, The Netherland, 2009.

[4] P. O'Neil, B. O'Neil, and X. Chen. (2007). The star schema benchmark (SSB). [Online]. Available: http://www.cs.umb.edu/~poneil/StarSchemaB.PDF

[5] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," *Commun. ACM*, vol. 51, no. 12, pp. 77–85, Dec. 2008.

[6] S. Manegold, P. A. Boncz, and N. Nes, "Cache-conscious radix-decluster projections," in *Proc. Int. Conf. Very Large Databases*, Sep. 2004, pp. 1–24.

[7] C. Balkesen, J. Teubner, G. Alonso, and T. Ozsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," *Proc. Int. Conf. Data Eng.*, Apr. 2013, pp. 362–373.

[8] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *Proc. Int. Conf. Manage. Data*, Jun. 2011, pp. 37–48.

[9] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *Proc. Int. Conf. Manage. Data*, Mar. 2009, pp. 283–296.

[10] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *Proc. Int. Conf. Manage. Data*, Jun. 2008, pp. 967–980.

[11] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. Int. Conf. Data Eng.*, Apr. 2011, pp. 195–206.

[12] (2014, Oct. 12). [Online]. Available: http://hyper-db.de/

[13] C. Balkesen, G. Alonso, J. Teubner, and M. Tamer Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 85–96, 2013.

[14] P. A. Boncz, M. Zukowski, and N. Nes. (2005, Jun.). MonetDB/X100: Hyper-pipelining query execution. in *Proc. 2nd Biennial Conf. Innovative Data Syst. Res.*, pp. 225–237. [Online]. Available. http://www.cidrdb.org/cidr2005/papers/P19.pdf

[15] P. A. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark," in *Proc. TPC Technol. Conf. Perform. Eval. Benchmarking*, Aug. 2013, pp. 61–76.

[16] G.-M. Hector and S. Kenneth, "Main memory database systems: An overview," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 6, pp. 509–516, Dec. 1992.

[17] (2014, Oct. 12). TimesTen. [Online]. Available. http://www.oracle.com/technetwork/products/timesten/overview/index.html

[18] (2014, Jul. 30). SolidDB. [Online]. Available. http://www-01.ibm.com/software/data/soliddb/

[19] (2014, Mar. 15). EXtremeDB. [Online]. Available. http://www.mcobject.com/extremedbfamily.shtml

[20] M. Zukowski and P. A. Boncz, "Vectorwise: Beyond column stores," *IEEE Data Eng. Bull.*,vol. 35, no. 1, pp. 21–27, May 2012

[21] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in SAP HANA database: The end of a column store myth," in *Proc. Int. Conf. Manage. Data*, May 2012, pp. 731–742.

[22] (2014, Nov. 19). SAP In-Memory Computing on IBM eX5 Systems. [Online]. Available. http://wenku.it168.com/d_001604407.shtml

[23] (2014, Aug. 12). SAP HANA Performance—Efficient Speed and Scale-Out for Real-Time Business Intelligence. [Online]. Available. http://sap.ebizdemo.net/files/HANA-Scale-Out-Performance.pdf

[24] V. Raman et. al., "DB2 with BLU acceleration: So much more than just a column store," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1080–1091, Aug. 2013.

[25] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou, "SQL server column store indexes," in *Proc. Int. Conf. Manage. Data*, Jun. 2011, pp. 1177–1184.

[26] P. Valduriez, "Join indices," *ACM Trans. Database Syst.*, vol. 12, no. 2, pp. 218–246, Jun. 1987.

[27] H. Wang, X. Qin, Y. Zhang, S. Wang, and Z. Wang, "LinearDB: A relational approach to make data warehouse scale like mapreduce," in *Proc. 16th Int. Conf. Database Syst. Adv. Appl.*, Apr. 2011, pp. 306–320.

[28] G. Sanders and S. Shin, "Denormalization effects on performance of RDBMS," in *Proc. 34th Annu. Hawaii Int. Conf. Syst. Sci.*, Jan. 2001, p. 3013.

[29] S. K. Shin and G. L. Sanders, "Denormalization strategies for data retrieval from data warehouses," *J. Decision Support Syst.*, vol. 42, no. 1, pp. 267–282, Oct. 2006.
[30] Z. Morteza, P.-A. Somnuk, and S.-C. Haw, "Optimizing the data warehouse design by hierarchical denormalizing," in *Proc. Int. Conf. Appl. Comput. Sci.*, Nov. 2008, pp. 131–138.
[31] R. Johnson, V. Raman, R. Sidle, and G. Swart, "Row-wise parallel predicate evaluation," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 622–634, Aug. 2008.
[32] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo, "Business analytics in (a) blink," *IEEE Data Eng. Bull.*,vol. 35, no. 1, pp. 9–14, May 2012.
[33] Y. Li and J. M. Patel, "WideTable: An accelerator for analytical data processing," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 907–918, Aug. 2014.
[34] P. Pucheral, J.-M. Thévenin, and P. Valduriez, "Efficient main memory data management using the DBGraph storage model," in *Proc. Int. Conf. Very Large Databases*, Aug. 1990, pp. 683–695.
[35] K. M. Azharul Hasan, T. Tsuji, and K. Higuchi, "An efficient implementation for MOLAP basic data structure and its evaluation," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, Apr. 2007, pp. 288–299.

Yansong Zhang received the Ph.D. degree from the School of Information at the Renmin University of China, Beijing, China, in 2010. He is an associate professor in the School of Information, Renmin University of China. His research interests include main-memory database, data warehouse, OLAP, and coprocessor processing.

Xuan Zhou received the PhD degree from the National University of Singapore, Singapore, in 2005. He was a researcher at the L3S Research Centre, Germany, from 2005 to 2008, and a researcher at CSIRO, Australia, from 2008 to 2010. Since 2010, he has been an associate professor at the Renmin University of China. His search interests include database system and information management. He has contributed to a number of research and industrial projects in the European Union, Australia, and China.

Ying Zhang received the MSc degree from the VU University, Amsterdam, in 2004, and the PhD degree from the Centrum Wiskunde & Informatica (CWI), Amsterdam, in 2010. She is a postdoctoral researcher in the Database Architecture Group at CWI. Her primary research interests include database architecture designs for large volume, distributed, and heterogeneous scientific data.

Yu Zhang received the MSc degree from the Xiamen University Xiamen, China, in 2005. She is currently a doctoral candidate in the School of Information at the Renmin Universityof China. Her research interests include GPU based OLAP and database optimization.

Mingchuan Su received the MSc degree from the School of Information at the Renmin University of China, Beijing, China, in 2014. His research interests include main-memory database and query optimization.

Shan Wang received the BSc degree from Peking University in 1968 and the MSc degree from the Renmin University of China in 1981. She is a professor in the School of Information at the Renmin University of China. Her research interests include many aspects of data management. Her current research focuses on high-performance database, data warehouse, knowledge engineering, and information retrieval.