Efficient OLAP with UDFs

Zhibo Chen
University of Houston
Department of Computer Science
Houston, TX 77204, USA

Carlos Ordonez
University of Houston
Department of Computer Science
Houston, TX 77204, USA

ABSTRACT

Since the early 1990s, On-Line Analytical Processing (OLAP) has been a well studied research topic that has focused on implementation outside the database, either with OLAP servers or entirely within the client computers. Our approach involves the computation and storage of OLAP cubes using User-Defined Functions (UDF) with a database management system. UDFs offer users a chance to write their own code that can then called like any other standard SQL function. By generating OLAP cubes within a UDF, we are able to create the entire lattice in main memory. The UDF also allows the user to assert more control over the actual generation process than when using standard OLAP functions such as the CUBE operator. We introduce a data structure that can not only efficiently create an OLAP lattice in main memory, but also be adapted to generate association rule itemsets with minimal change. We experimentally show that the UDF approach is more efficient than SQL using one real dataset and a synthetic dataset. Also, we present several experiments showing that generating association rule itemsets using the UDF approach is comparable to a SQL approach. In this paper, we show that techniques such as OLAP and association rules can be efficiently pushed into the UDF, and has better performance, in most cases, compared to standard SQL functions.

Categories and Subject Descriptors

E.1 [**Data Structure**]: [Trees]; H.2.4 [**Database Management**]: Systems—*Relational Databases*; H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experiments, Measurement

Keywords

OLAP, UDF, CUBE

1. INTRODUCTION

On-Line Analytical Processing (OLAP) [2, 6] is a set of exploratory database techniques that allow the user to efficiently retrieve specific aggregations [6]. OLAP users try to find interesting or unexpected results by analyzing subsets of datasets using aggregations to generate different levels of detail. The underlying structure of OLAP is the dimensional lattice, which stores the aggregations for all levels of detail. Figure 1 shows the lattice for a dataset with four dimensions. Without optimizations, OLAP processing can be slow, but techniques [6], such as precomputation by aggregating on all dimensions, help improve performance. For the most part, an OLAP server, which sits between the DBMS and the client computers, is required to perform OLAP queries. Other ways of approaching OLAP queries have also been studied. Research have been conducted on representing OLAP lattices outside the database [16] as well as generating the lattice within the DBMS using only SQL [12]. These two approaches showcase the generation of lattices using the flexible C and the more restrictive SQL. There is a marked difference in the speed of these two approaches with the restrictive SQL being much slower than C. However, few have tried to merge the best of both approaches using user-defined functions (UDFs).

UDFs allow the user to write their own C-level code, which can then be compiled into object code and embedded directly into the DBMS. The result is a function that can be used just like any standard SQL commands. The UDF is basically an application programming interface (API) that allows end-users to extend the DBMS without actually coming in contact with the source code. In our research, we used table-valued functions (TVF) to push OLAP into the UDF. TVFs are able to take in values and return an entire table as a result [5]. In the past, UDFs could not perform an I/O task on the database and could only return values. However, with TVFs, we can now read and return entire tables. This development allows us to generate and hold entire tables within main memory and delay writing to the physical disk space until the generation is completed. In this paper, we propose using User-Defined Functions (UDFs) to generate entire OLAP lattices in main memory and only write the results to an output table once the lattice is completed.

This paper is organized as follows. Section 2 introduces some important definitions. The lattice structure and algorithm is presented in Section 3. We discuss and show some experimental results in Section 4. Section 5 explains research related to this paper. Our conclusions and possible future work is presented in Section 6.

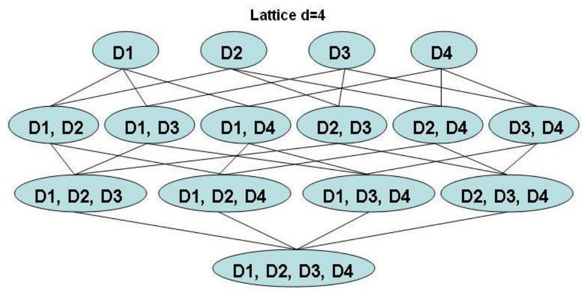

Figure 1: Dimension Lattice; d=4.

2. DEFINITIONS

We focus on a fact table F with n records having d cube dimensions [6] where $D = \{D_1, \ldots, D_d\}$, and e measure attributes, where $e \geq 1$. The precomputed version of table F is called table C and also includes d dimensions, but may have a smaller n. The dimensional lattice is the data structure that represents all subsets of dimensions and their containment. The size of the lattice depends on d and is computed as 2^{d-1} for binary dimensions. The data that is stored within the dimensional lattice is obtained from the measure attributes. We call each subset combination of dimensions a lattice node, which can be further divided into lattice groups, each of which represents one specific combination of values. We consider the empty set to be the top node and the full dimension set to be the bottom node. One level of the lattice contains all combinations that have the same number of dimensions.

3. UDF-BASED LATTICE GENERATION

The creation of an OLAP lattice within the UDF requires a special data structure that should be able to handle the large number of combinations and also allow for quick outputs to tables. The UDF is different than creating the same structure outside the DBMS because we cannot indefinitely maintain the entire structure in memory. Instead, the structure has to quickly be converted into rows that can then be stored within a table. In this section, we will first introduce the lattice structure that can accommodate all these requirements. Next, we will provide the algorithm for generating this structure and writing to the table. Then, we will show how this structure can be applied to an OLAP example. Finally, we will show how this structure can also be used toward association rule itemsets.

3.1 Lattice Structure

The format of our lattice structure is shown in Figure 2 for three dimensions. The structure is tree-like with each node of the tree representing one combination set within a lattice. A two-tiered design is used in which the first level includes information regarding the node and its combination of dimensions. The second level stores data regarding the groups within each node. The data stored within each node depends on the task that the OLAP structure will accomplish. Example of stored data include the mean, median, sufficient statistics, or a combination of the values. While the number of nodes within the structure only depends on the number of dimensions, the number of groups within a node depends entirely on the dataset. A dataset with more

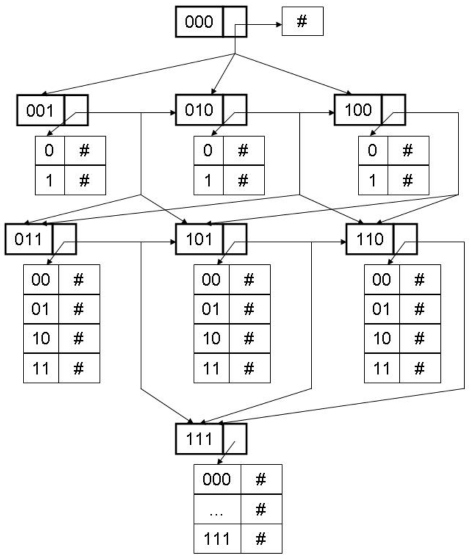

Figure 2: Data Structure; d=3.

distinct tuples will cause more groups to be created than a sparse dataset. On the top level, every node is connected with its supersets and subsets in order to allow for quick traversal of this structure. For example, the node {011} is connected to two subsets {001} and {010} as well as one superset {111}. While the links are not used during lattice generation, they are helpful for future processing by allowing for quick access to all related nodes above and below a specific node. The second level is not as widely connected, with each group only being linked to its neighbors in the same node. While some other data structures have indexes that allow for quick access of specific nodes [7], we do not require such indexes since future explorations will not be conducted in main memory. Instead, the UDF is designed to only hold the data structure in main memory and once generated, the lattice will then be inserted into a table.

3.2 Algorithm

Prior to the execution of this algorithm, table F needs to be precomputed into a new table C by aggregating on all dimensions [6]. The rationale is that by aggregating on all the dimensions, we can create a new table which will be much smaller than the original table. In effect, table C is table F aggregated to the finest granularity. In the worst case that all tuples of F are distinct, then table C will be the same size as F and we suffer no penalties. However, should there be a significant reduction in the input table size, then performance would improve because the algorithm would have to have process as many tuples. As a result, we use table C as a substitute for table F throughout the algorithm.

The algorithm consists of two parts: the updating of the nodes/groups and the traversal of the lattice. For each tuple of the input table, we need to determine if a node exists within the lattice structure. If it does not exist, then a new node will need to be created and linked with the appropriate subsets and supersets. These links can be accomplished by the removing or inserting of individual dimensions, determining if those nodes exist, and creating a link if possible. On the other hand, if the node already exists, then the

```
Algorithm to create the lattice structure in UDF
Input: precomputed fact table $C$
1 : Create a new lattice structure $LAT$
2 : while more tuples exist in $C$ do
3 :     rowval ← next tuple from $C$
4 :     setD ← "000"
5 :     while more combinations exist do
6 :         if node with setD combination not exist then
7 :             node ← new Node with setD as description
8 :             insert node into $LAT$
                and link with subsets and supersets
9 :         else
10:             node ← node with setD as description
11:         end if
12:         dims ← extractDims(rowval, node)
13:         if dims exists in node then
14:             group ← getGroup(node, dims)
15:             update stored values in group
16:             updateGroup(node,group)
17:         else
18:             newGroup ← new group
19:             populate group with required values
20:             insertGroup(node,newGroup)
21:         end if
22:         setD ← addOne(setD)
23:     end do
24:     Insert or replace node into $LAT$
25: end do
26: Write lattice structure to DBMS
```

node can be directly used because we know the links have already been created. We do not pre-generate the entire lattice structure in memory because of the sparse nature of the datasets. This sparseness becomes more apparent as the number of dimensions increases.

Once the appropriate node is extracted or created, we will then either update a group or create a new group for that particular node. Afterwards, we would need to retrieve the group containing those specific values or create a new group with those values. If the group is in existence, then we need to add the measure attributes associated with the current tuple to those already stored in the group. If the group is new, then we would create a new group and insert the measure attributes. The process for adding or inserting these new attributes depends on the final goal of the lattice. If only a total number is required, then we would simply add one to the group. On the other hand, if the lattice is to be used for more complex calculations, such as statistical tests [12], then we may need to store the sufficient statistics of the measure attributes. Regardless of what's stored inside the lattice, the procedure is the same. For a new group, we must insert the group at the end of the group list within the node and change the appropriate links. The order of the groups within a node does not matter because we will not be using this structure for exploration.

For each tuple, we need to traverse the lattice structure. We can accomplish this by changing each dimensions into a 1 for on or a 0 for off. When combined, this forms a binary value that can be easily incremented. The exact order in which we traverse the lattice does not matter. As a result, we can begin with all 0s for the binary representation, which represents the empty set or root node of the lattice structure.

Table 1: Sample OLAP Input Table.

ID	D1	D2	D3	M1
1	0	0	1	15.2
2	1	0	1	19.3
3	1	0	0	81.5
4	1	1	0	62.8
5	1	0	1	60.5

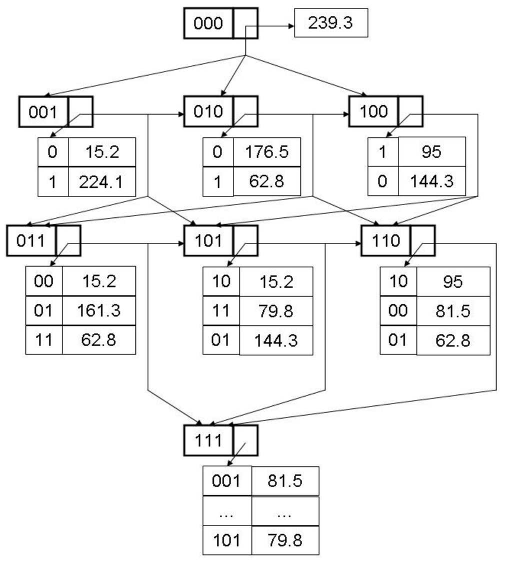

Figure 3: Lattice Structure for OLAP Example.

Once we complete each node, we can add one to the binary representation to obtain the next node. Once the representation reaches all 1s, then we know we have traversed the entire lattice. For example, if we have a lattice structure representing three dimensions, we would start with the binary 000, meaning no dimensions are in the set. The next two nodes to be visited would be 000+1 or 001 and 001+1 or 010, which corresponds to the node D1 and D2, respectively.

Since the algorithm updates the structure each time a new tuple is read into memory, we cannot apply a support or population threshold on the data structure. The algorithm only performs one-pass on the entire condensed dataset, C, and thus we do not know the actual size each node in the lattice until the entire table is read. The problem is that this does not allow for the inclusion of support related constraints, which are helpful in increasing performance. Multiple passes would need to be performed in which the data structure is broken into multiple levels. In this way, we can control the support between levels.

3.3 Generation of OLAP Lattice

We will now show an example of the generation of an OLAP lattice using our proposed structure. Table 1 shows a sample dataset F with three dimensions and one measure attribute. Though we are using only binary values for the dimensions because they are used in many data mining problems, our algorithm can handle more dimension values. For this example, suppose we wish to create a lattice that would hold the sum of the measure attribute. Then, we would

Table 2: Sample Association Rule Input Table.

ID	D1	D2	D3
1	0	0	1
2	1	0	1
3	1	0	0
4	1	1	0
5	1	0	1

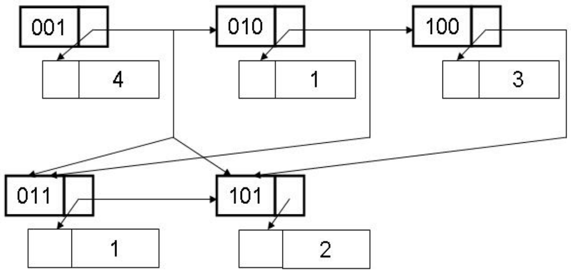

Figure 4: Lattice Structure for Association Rules.

start with the first tuple, and traverse the lattice while updating or creating new groups for specific nodes. Assume that in this example, we are only storing the sum of the values. Then, the first node we would reach is the node 000. Since this node has not been created, we would create the node and insert it into the lattice structure. Node 000 represents the empty combination or the one where none of the dimensions matter. This means that all of the tuples will be added to this node since we have no criteria.

As a second example, if we were at the node with binary representation 010, then we would be observing only the second dimension when determining which group to update. For the first tuple, this node will also need to be created, but future tuples will not need to create the node, only update it. The number of groups within this node is equal to the number of distinct values for the second dimension. In our case, D2 has a value of 0 for the first tuple. Thus, we would create a new group to represent the value 0 and store 15.2 within it. How this group represents 0 depends on the implementation of the data structure. For example, if hashes are used, then the group would have a key of 0 and a value of 15.2. Let us now look at what would happen if we are at the same node, 010, with the second tuple. Again, we would only be interested in the second dimension and the group 0. Since this group was already created by the first tuple, we would need to retrieve the data and update it to 34.5. This value would then be stored in group 0 of the node 010. This addition or updating of groups within nodes would continue until the lattice is completely traversed. Figure 3 shows the lattice structure stored by the UDF for the sample data set.

3.4 Generation of Association Rule Itemsets

The first step in most association rule algorithms is to generate the itemsets from the dataset. We will now show how our lattice structure can also be used to generate such itemsets. Our proposal will not consider any constraints and will instead generate data for all possible itemsets. There are two major differences between using the lattice structure for OLAP and for association rules. While OLAP considers each node to represent the combination of the dimensions, association rule considers each node to represent the set of items that can be found within the same transaction. For example, if we arrive at node {011}, OLAP would see this as meaning we are looking at the combination of D1 and D2. On the other hand, association rule would see this node as meaning both item D1 and item D2 are present in a tuple. The second major difference is that each node of OLAP may contain many groups while the nodes of association rules would only contain one group. For example, nodes in OLAP contain groups that have both 0s and 1s. However, the association rule nodes would only care about those dimensions whose values are 1. Please note that we are not looking at the negation of items.

Table 2 displays an example of a data set that can be used to create the association rule itemsets. In this case, the dimensions are binary because most association rule algorithms use only binary values. By not using the traditional transaction format, we can deal with less tuples and be provided with more information per row. The procedure for creating a lattice structure for itemsets is virtually identical to creating it for a lattice. We begin with the node 000, but in this case, we do not even create the node. The reason is that 000 represents the case where none of the dimensions are observed, and since association rule works on the existence of items, there needs to be at least one dimension that is on. For example, if we look at the next node, 001, then we see that dimension 1, or D1, is on while the rest of the dimensions are off. If we look at the sample dataset, then we can see that the first tuple does not have a 1 in D1. This means that the first tuple will not affect such a node, and we will not create this node. Instead, we will move on to the next node and so on. This is a classical case of how association rule would disregard this node for itemsets while OLAP, which counts 1s and 0s, would create a group representing the value 0. Let us now move forward several nodes to the node 100, which looks at D3. Since the first tuple affects the node 100, a new node will be created with these descriptions and placed into the lattice structure. Within this node, we will have to create a new group with a store data value of 1 because only one tuple so far touches this node. We would then traverse the rest of the nodes. After going through the entire dataset, the final lattice structure for this example is shown in Figure 4.

3.5 Storage of Lattice Structure

The efficient storage of the lattice structure must also be studied since we are using UDFs, which loses all data stored in main memory upon completion. In order to allow for the most efficient access of specific nodes of the structure, we implemented a full index on the output table. In the table, we use "ALL" to represent the dimensions that were not used in the aggregation. Table 3 shows an example of a lattice structure stored in the database. This procedure is suitable for all situations except in the case when one of the dimensions has a value that is also represented by "ALL". However, these cases are rare and can be solved individually. The size of the output table depends on the values that are stored within the lattice. For example, if we are storing only counts within the lattice, then a four dimensional dataset with one measure attribute would create an output table with four dimensions columns and one count column. On the other hand, suppose we are storing sufficient statistics within the

Table 3: Sample Storage Table for Lattice Structure.

D1	D2	D3	Data
ALL	ALL	ALL	239.3
0	ALL	ALL	15.2
1	ALL	ALL	224.1
ALL	0	ALL	176.5
⋮	⋮	⋮	⋮

lattice. Then, for the same dataset would generate an output table with four columns that are dimensions and three columns for the sufficient statistics. The three columns being N (count), L (sum of measure attribute), and Q (sum of measure attributes squared). Regardless of the size of the output table, the primary index on all the dimensions allows for efficient retrieval of specific combinations.

4. EXPERIMENTAL EVALUATION

We conducted our experiments on a server with 3.2GHz CPU, 4GB RAM, and 600GB HD. The DBMS is SQL Server 2005, and we performed our UDFs using table-valued functions. The following subsections first presents the two datasets used throughout this experiments. Next, we will show and analyze the performance of the UDF versus a pure SQL approach on both datasets. Then, we will compare the UDF to the CUBE operator that can be found within the DBMS also on both datasets. Finally, we will show that the lattice structure used in the UDF can also be implemented to generate association rule itemsets on the Heart dataset.

4.1 Datasets

We used two datasets to experimentally show the performance of the UDF versus both SQL and the CUBE operator inside of SQL Server. The first dataset is a private medical dataset with heart data with n=655, d=21, and e=4. The dimensions represent factors of a patient while the measure attributes represent the degree of stenosis, or narrowing of the arteries in the heart. We choose to analyze this medical dataset because despite its small size, it still produces a large number of patterns. The second dataset is a processed version of the ORDERS table from the TPC-H benchmark database with the default parameters. We decided to use the total price as the measure attribute, removed the comments column, and expanded the date to three columns representing year, month, and day. The resulting dataset has the following properties: n=1.5 million, d=8, and e=1.

4.2 Performance for OLAP

In this section, we will compare the execution times of generating the entire lattice using UDFs versus the execution times for using pure SQL. The SQL approach uses the GROUP BY command to obtain each node of the dimensional lattice. In order words, for the node {D1,D2}, the SQL would include a GROUP BY on D1 and D2. Once the groups are obtained, the values are inserted into an output table for further exploration or analysis. Additional information regarding the SQL approach can be found in [12].

Figure 5 shows a comparison of the execution times when varying d to generate an OLAP lattice using UDFs and pure SQL on the Heart dataset. The use of main memory within the UDF helps decrease the execution time to less than 10%

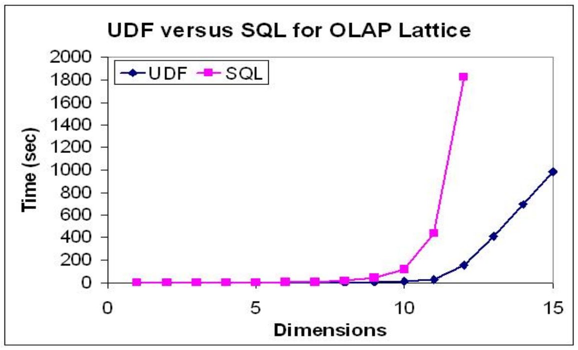

Figure 5: Scalability of UDF and SQL for d.

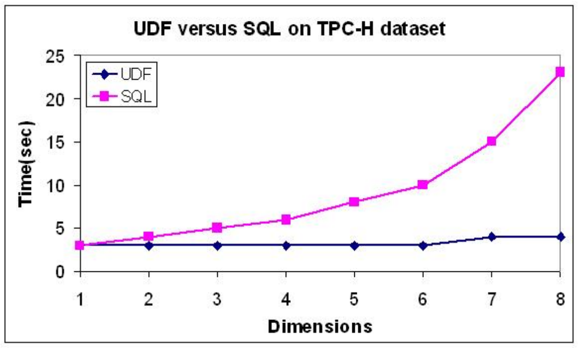

Figure 6: Scalability of UDF and SQL for d on TPC-H Dataset.

of the time it takes for SQL to create the lattice. While both approaches are still affected by the combinatorial growth of the lattice as d increases, the UDF approach has a significantly slower growth rate than the SQL approach. For the TPC-H dataset, Figure 6, we can observe a similar divergence in which the SQL approach is able to keep pace with UDF only for the first few dimensions. Once the lattice size reaches a large size, the UDF becomes much faster. Both figures also show how the gap between the two approaches widens as d increases regardless of which dataset is used.

There are several reasons behind why UDF is over ten times faster than SQL. First, the SQL approach requires one aggregation to be performed for each combination of the dimensions. As a result, the input table is scanned many times during the course of a lattice generation. In fact, the number of table scans is equal to the number of dimensional nodes in the lattice. While this is not inhibiting for very small d, once d surpasses six or seven, the number of table scans becomes a bottleneck. On the other hand, the UDF only performs one scan of the input table and generates the lattice as it is reading the tuples. The size of the lattice still affects the UDF because it needs to update many of the nodes for each of the tuples. While this may seem slow, such updates are faster than the aggregations. The second reason supporting the UDF approach is that it is performed in main memory. The lattice structure is used to hold the entire lattice in memory until completion. At that time, the entire structure is written into a table. Therefore, permanent disk space is only used during the last phase of the generation. We found that the time to write to the output table takes up nearly 50% of

Table 4: **UDF versus SQL for OLAP varying n with d=12 on Heart dataset.**

n	PreComputation(sec)	UDF(sec)	SQL(sec)
655	0	159	1827
6,550	0	159	1828
65,500	1	159	1828
655,000	2	160	1829
6,550,000	8	161	1829

the execution time of the entire UDF, especially for larger dimensions. For the SQL approach, everything is performed on disk and in permanent space. Though each aggregation results only in an insert, this is still much slower than in memory. In addition, the table scans all require reading from the disk and results in longer execution times for the SQL approach.

We now observe the effect of varying n on both the UDF and SQL approaches to generating the lattice. In this case, we are holding d at 12 and duplicating the dataset by factors of 10 to create the larger tables. The precomputation step of both approaches helps keep the execution times steady even as n grows into the millions. The reason is that by condensing the input table into distinct values, we combined many of the tuples that were identical in terms of the dimensions. For example, in a dataset with twelve dimensions, there are a total of 4096 different combinations of the dimensions, which means that the precomputed input table can only have a maximum of 4096 rows. Thus, for all these large tables, we can condense them to input tables with at most slight over four thousand tuples. The key to the increasing n is that the complexity depends less on how many tuples there are and more on d. It is the number of dimensions that controls the maximum number of distinct tuples possible in a table. As a result, for tables with large n and small d, there will be a large decrease in the number of rows that leads to an equally large increase in performance. Thus, the precomputed step helps to make both approaches efficient even for large n. There will only be a problem if there is a large d, which has been explained previously. We can see from Table 4 that the execution times exhibit very little change though the UDF approach is still over ten times faster than the SQL approach. While we did not vary the n for the TPC-H dataset, it is safe to assume that for a dataset with a smaller d, the result will be similar to that obtained for the Heart dataset. In the case of the TPC-H dataset, we are looking at the maximum of 256 tuples in the precomputed table regardless of the size of the dataset.

Though the previous experiments have binary dimensions, our data structure can also handle dimensions with more distinct values. Neither the algorithm nor the structure needs to be changed because more distinct values only produces more groups, not nodes. The performance of the algorithm should not be affected because the groups are still either altered or added into each node. The only difference is that there will be more groups under each node.

4.3 Comparison with CUBE operator

Whenever the generation of an OLAP lattice is considered, we must look at the CUBE operator provided in SQL Server. This operator also creates the lattice and the main difference is the use of NULL in the place of the ALL keyword. Table 5 and Table 6 show a comparison of the times for the UDF approach and for using CUBE in both the Heart and TPC-H datasets. As we can see, the CUBE operator is faster than the UDF for dimensions higher than eight, but it cannot go further than ten dimensions on the Heart dataset. In fact, the CUBE operator cannot go beyond ten dimensions for any dataset because of the limitation set by the DBMS. On the other hand, the UDF approach is only limited by the memory space that the lattice uses, allowing it to reach higher dimensions.

In addition to the limitation on number of dimensions, the CUBE operator is also restrictive because we cannot alter the way it behaves. For example, depending on the final goal of the OLAP lattice, we can apply constraints that will remove parts of the lattice and improve performance. For example, if we use the lattice to perform analysis [12], then we can include a depth constraint that will limit the size of the lattice. The depth constraint [13] determines how deep the lattice will be calculated. If we set a depth of four on a twelve dimension table, then the lattice will only be calculated up to a level of four. This results in nodes that have a maximum of four dimensions. Such a constraint can be implemented in the UDF approach, but cannot be accomplished in with the CUBE operator. In fact, because we cannot change the CUBE operator in any way, no constraints or changes can be implemented. Thus, we believe that the flexibility of our UDF approach makes it a more ap-

Table 5: UDF versus CUBE operator varying d on Heart dataset. *denotes unable to compute.

d	UDF Time(sec)	CUBE Time(sec)
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
8	2	1
9	5	1
10	10	1
11	31	*
12	159	*
13	412	*
14	695	*
15	985	*

Table 6: UDF versus CUBE operator varying d on TPC-H dataset. n=1.5M

d	UDF Time(sec)	CUBE Time(sec)
1	3	2
2	3	2
3	3	2
4	3	2
5	3	2
6	3	2
7	4	2
8	4	3

Table 7: UDF versus CUBE Operator varying n with d=10 on Heart dataset.

n	UDF Time(sec)	CUBE Time(sec)
655	10	1
6,550	10	1
65,500	11	3
655,000	12	7
6,550,000	17	14

Table 8: UDF versus SQL for Association Rule varying d on Heart dataset.

d	UDF Time(sec)	SQL Time(sec)
2	0	1
4	0	2
8	1	5
10	12	12
12	23	19
14	124	26
16	680	57

pealing approach than using the CUBE operator because of the ability to implement constraints or other optimizations.

Table 7 shows the performance of the two lattice generation methods when the size of the dataset is increased. The UDF approach is affected much less than the CUBE approach as n increased. Our UDF approach increased less than one fold when the dataset was increased by 10,000 times, but the CUBE approach saw an increase in execution time of fourteen folds. Not only this, when we approached millions of rows, the execution times for both datasets appear to be converging. The main reasoning behind narrowing of the gap between the two approaches is the use of the precomputation step for our UDF approach. As explained above, it reduces the size of the input table to more manageable levels. Thus, we can see that while the UDF approach appeared to be much slower than using the CUBE operator for small datasets, the difference is less pronounced for large datasets. The main advantage of the UDF approach over the CUBE operator is the flexibility that the UDF enjoys. Optimizations such as constraints can be applied to the UDF by the user while the CUBE operator cannot be changed to apply these optimizations.

4.4 Performance for Association Rules

We will now compare our UDF approach against a purely SQL approach in terms of generating association rule itemsets. In both approaches, we are generating the full set of itemsets, with no support threshold or set containment restrictions. Table 8 shows the execution times for both the UDF approach and the SQL approach. As can be seen, the UDF is comparable with the SQL approach until ten dimensions, after which the SQL approach performs much better. The main difference between the two approaches is that the pure SQL approach includes the support constraint that the UDF approach is not able to mimic. For SQL, the strategy is to use multiple passes to determine support at least level of the itemsets and filter based on that. Since the UDF approach uses only one pass, we are unable to confidently determine the support of any itemsets until all of

Table 9: UDF versus SQL for Association Rules varying n with d=12 on Heart dataset.

n	UDF Time(sec)	SQL Time(sec)
655	23	19
6,550	23	23
65,500	24	54
655,000	26	114
6,550,000	32	*

the dataset has been read. This difference greatly affected the efficiency of the code since the SQL approach was able to prune portions of the itemsets while the UDF approach needed to generate the full set of itemsets.

Though it may seem that the SQL approach is better than using UDF, the truth is not as simple. The dataset that was used to conduct the experiment only had 655 tuples. If we increase the size of the dataset, by duplicating the rows, a completely different pattern emerges. Table 9 shows the execution times for increasing large datasets. We see that while the SQL approach performs well at lower n, when we increase the size of the dataset, the perform drops. On the other hand, the performance for the UDF remains virtually constant. Additional experiments showed that this trend holds in all of the other dimensions. The only change between the different dimensions is that the point at which UDF surpasses SQL in performance changes.

5. RELATED WORK

We propose the generation of OLAP lattices within user-defined functions. To the best of our knowledge, no one has pushed OLAP processing into the UDF. OLAP lattices and cubes have been generated using various structures both outside the database and inside the database. The authors in [4] developed a statistical tree that stored the OLAP lattice in a tree format with the ALL sections being calculated on the fly. A condensed version of an OLAP lattice that used prefix and suffix redundancy reduction to create smaller lattices was presented in [16]. On the other hand, [12] presents one case where the OLAP lattice was generated entirely using SQL without using UDFs or stored procedures. The authors in [9] combined OLAP with association rules using OLAP servers from MS SQL Server 2000. Our approach differs from these references because we use UDFs to perform similar tasks.

On the UDF side, much research has been implemented on comparing the effectiveness of UDF versus SQL. The authors in [14] not only showed how UDFs are just as efficient in arithmetic operations as standard SQL, but also showed how UDFs can be used to extend the DBMS with capabilities that SQL does not have. UDF is taken one step further in [11] by showing that it can be used to build and score statistical models. Various other approaches to extend the capabilities of SQL have also been taken. The authors in [3] showed how primitive operators can be used to perform tasks such as pivoting and sampling. The efficient computation of percentages in horizontal and vertical formats is proposed by [10]. Similar to these approaches, our research involves extending SQL to perform additional tasks. However, we choose to use UDFs and extended SQL with the ability to efficiently produce OLAP lattices.

Association rules [1], like the OLAP lattice, has generally be implemented outside the database. The authors in [7], developed an algorithm for generating large numbers of rules efficiently. Constraints were also developed to both improve performance and decrease the number of rules obtained in [13]. The generation of on-line association rules was proposed in [8]. While all of the above references proposed new structures or methods of generating association rules, none of them are implemented within the database. We show that our lattice structure can also be used to store association rule itemsets. The authors of [15] introduces the computation of association rules using UDFs, but the process is split into multiple UDFs. In our case, not only are we creating one user defined table valued function that will generate the itemsets in one call, we have also provided a more versatile UDF that can be used for OLAP processing.

6. CONCLUSIONS

In this paper, we propose to push the generation of the OLAP lattice into UDFs instead of completely within the database or entirely outside the database. The rationale is to find a middle ground between the highly efficient algorithms of C and other high-level languages with SQL within a more secure database. We developed a new lattice structure to store the lattice while it is in main memory. Even though the structure would have to be written to physical space at the completion of the UDF, we experimentally found that the resulting execution time was still over ten times faster than generation the lattice with standard SQL. This trend holds not only for datasets of various dimensions, d, but also of various length, n. In addition, we also compared the UDF approach to the built-in CUBE operator found in SQL Server. We found that while CUBE was more efficient than our UDF approach for some dimensions, it is limited both by the number of dimensions and by its flexibility. Not only can it not exceed ten dimensions, but the user has no control over any aspect of the generation. On the other hand, performing the generation in UDF had a higher limit with the flexibility to apply constraints or change the output representation. Furthermore, we also used the lattice structure to generate association rule itemsets and compared it with an approach using only standard SQL. Based on the experiments, we can conclude that UDFs are slower than using standard SQL for datasets with small n. However, as n increases, the performance of UDF catches up to and surpasses that of the SQL approach. Thus, we show that the generation of an OLAP lattice or association rule itemsets within UDFs is possible and warrants further research.

For future work, we want to increase the maximum number of dimensions that can be handled by the UDF by using techniques for dealing with sparse data. The memory consumption of the methods compared in this paper needs to be investigated in more detail. We want to investigate the support constraint and implement it as the dataset is being read. In addition, we want to explore the possibility of using bitmaps or other indexing data structures to represent the lattice. This would allow us to both improve performance and increase the capacity of the UDF. Finally, we want to compare the UDF approach with other similar methods that allowed for either OLAP processing or generated association rules. Such comparisons would let us study where our approach ranks and also help pinpoint specific trouble locations in the implementation or algorithm.

7. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Conference*, pages 207–216, 1993.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[3] J. Clear, D. Dunn, B. Harvey, M.L. Heytens, and P. Lohman. Non-stop SQL/MX primitives for knowledge discovery. In *ACM KDD Conference*, pages 425–429, 1999.

[4] L. Fu and J. Hammer. Cubist: a new algorithm for improving the performance of ad-hoc OLAP queries. In *DOLAP Workshop*, 2000.

[5] B. Hamilton. *Programming SQL Server 2005*. O'Reilly Media, 1st edition, 2006.

[6] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 1st edition, 2001.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Conference*, pages 1–12, 2000.

[8] C. Hidber. Online association rule mining. In *ACM SIGMOD Conference*, 1999.

[9] R. Messaoud, S. Rabaséda, O. Boussaid, and R. Missaoui. Enhanced mining of association rules from data cubes. In *DOLAP*, pages 11–18, 2006.

[10] C. Ordonez. Vertical and horizontal percentage aggregations. In *ACM SIGMOD Conference*, pages 866–871, 2004.

[11] C. Ordonez. Building statistical models and scoring with UDFs. In *ACM SIGMOD Conference*, pages 1005–1016, 2007.

[12] C. Ordonez and Z. Chen. Evaluating statistical tests on OLAP cubes to compare degree of disease. *IEEE Journal of Transactions on Information Technology in Biomedicine*, 2008.

[13] C. Ordonez, N. Ezquerra, and C.A. Santana. Constraining and summarizing association rules in medical data. *Knowledge and Information Systems (KAIS)*, 9(3):259–283, 2006.

[14] C. Ordonez and J. García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *ACM CIKM Conference*, pages 503–512, 2006.

[15] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *ACM SIGMOD*, pages 343–354, 1998.

[16] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the petacube. In *ACM SIGMOD Conference*, pages 464–475, 2002.