© 2016 IJSRSET | Volume 2 | Issue 1 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

OLAP Query Optimizer: SELECTION Operation

Kothari Badal K.*, Harish Nagar, Dr. Ashok R. Patel

Department of Computer Science and Engineering,. Mewar University, Rajasthan, India

ABSTRACT

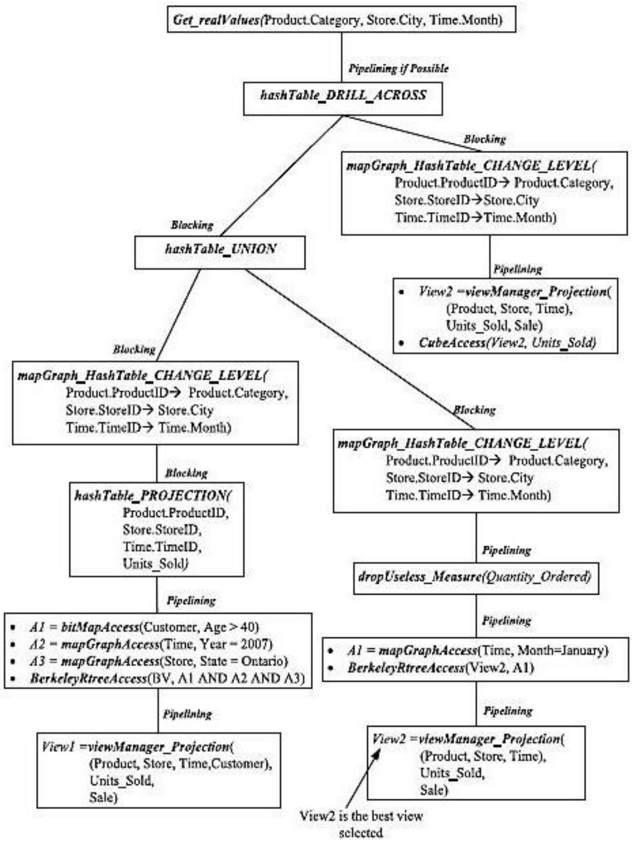
A query processor (query compilation and execution) is an essential component in any database management system (DBMS). Specifically, query compilation transforms user queries into a sequence of database operations, while query execution executes those given operations. Retrieve Information from the OLAP Storage is very important task, but because of large amount of OLAP data it's taking tremendous time for the execution of the query. In this paper, we shall cover algorithm for the SELECTION operator (OLAP Algebraic Operator), which is used to access the data of the OLAP storage.

Keywords: OLAP Algebra, OLAP Query Optimizer, OLAP Query, OLAP selection Operator

I. INTRODUCTION

In this paper, we look at the algorithms that are used to access the data of the OLAP storage. Moreover, we shall cover the algorithms applied in the execution of our OLAP algebraic operator (SELECTION) against the indexed cube (stored in the Berkeley DB) and associated dimension tables. Berkeley databases are used in our server to store the indexed cube in one physical file. We assume that we have enough memory to hold the result of any OLAP operator and any extra data structure. Note that extensions to external memory are expected in the future. Finally, for each physical operator in the OLAP physical query plan, we determine the appropriate algorithm(s) that can be used to answer them (e.g., algorithm x implements the physical operator BerkeleyRtreeAccess()).

The result of query compilation is an OLAP physical query plan explained in below figure, which defines an efficient execution plan for the received OLAP query. We order the execution of all nodes of the physical plan tree in a bottom-up, left-to-right manner. In other words, we order the nodes of the tree in such that a pre-order traversal traverses the entire physical query tree. Our OLAP query optimizer can generate a sequence of function calls - one for each physical operation in the physical plan - and pass them to the OLAP query engine for execution.


Get_realValues(Product.Category, Store.City, Time.Month)
Pipelining if Possible
hashTable_DRILL_ACROSS
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Category, Store.StoreID→Store.City Time.TimeID→Time.Month)
Pipelining
View2 =viewManager_Projection((Product, Store, Time), Units_Sold, Sale)
CubeAccess(View2, Units_Sold)
Blocking
hashTable_UNION
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Category, Store.StoreID→ Store.City Time.TimeID→ Time.Month)
Blocking
hashTable_PROJECTION(Product.ProductID, Store.StoreID, Time.TimeID, Units_Sold)
Pipelining
A1 = bitMapAccess(Customer, Age > 40)
A2 = mapGraphAccess(Time, Year = 2007)
A3 = mapGraphAccess(Store, State = Ontario)
BerkeleyRtreeAccess(BV, A1 AND A2 AND A3)
Pipelining
View1 =viewManager_Projection((Product, Store, Time,Customer), Units_Sold, Sale)
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Category, Store.StoreID→ Store.City Time.TimeID→ Time.Month)
Pipelining
dropUseless_Measure(Quantity_Ordered)
Pipelining
A1 = mapGraphAccess(Time, Month=January)
BerkeleyRtreeAccess(View2, A1)
Pipelining
View2 =viewManager_Projection((Product, Store, Time), Units_Sold, Sale)
View2 is the best view selected

In addition, the server must also select an algorithm for each OLAP operator in the OLAP logical plan in order to turn the preferred logical plan into a physical plan. We note that the algorithm for each OLAP operator (e.g., SELECTION) depends on the functionality developed.

In this paper, we discuss the SELECTION of algorithms for OLAP operators defined in our OLAP algebra.

II. METHODS AND MATERIAL

A. Hoosing A Selection Method

The selection is the driving operation behind most analytical queries. Therefore, one of the important steps in choosing a physical plan is to select an implementation for each selection operator. As was illustrated, SELECTION in the preferred logical OLAP query plan is of the form: SELECTION(Dim.DimID = x AND Dim1.Dim1ID = y OR Dim2.Dim2ID ...)C. SELECTION is defined as a listing of dimensions related via AND and OR, where each dimension is associated with a condition. For simplicity, we consider the SELECTION with only one dimension (i.e., Dim) like SELECTION(Dim.DimID = x), such that x is a set of DimIDs that satisfy the user's query condition (UC) associated with one dimension called Dim (Note that DimID is the most detailed level of dimension Dim). The user's query condition associated with dimension Dim is of the form "Dim (A OP c)", where A can be a hierarchical or non-hierarchical attribute of dimension Dim, OP can be any comparison operator defined by our OLAP query grammar (e.g., <, >, =, IN LIST), and c is a constant or set of constants. UC is a compound condition of one or more simple conditions against dimension Dim (connected via logical operators AND and OR). We would like to eliminate the inner/natural joins between the cube and dimension tables that would ordinarily be required to exclude cube rows that do not satisfy the query restriction. The implementation of SELECTION is divided into the following three steps.

First, we need to find all dimension members (DimIDs) satisfying the query restriction called UC (defined by the user). For simplicity, we consider the query condition UC =Dim (A OP c).

1. If A is a hierarchical attribute level in dimension Dim, then we retrieve all DimIDs (most detailed integer values) that satisfy the comparison UC(AOP C), using the enhanced hierarchy manager (mapGraph).
2. If A is a non-hierarchical attribute level, then we retrieve all DimIDs that satisfy UC, using the FastBit compressed bitmap index created for each non-hierarchical attribute level in the dimension.

If UC is the AND/OR of simple conditions, then we use mapGraph and/or FastBit bitmap indexes to identify the set of DimIDs that satisfy UC. Using the mapGraph and the bitmap indexes ensure that the resulting DimIDs that satisfy the query condition (UC) associated with dimension Dim are sorted. This result is organized as an ordered set of contiguous ranges that is stored in a main-memory sorted array. Given a DimID value v, we can directly apply a binary search within the sorted array to verify the existence of that given value. We can use similar techniques to find and store the dimension IDs for other user's dimension conditions mentioned in the SELECTION operator. An example of this will be provided shortly.

Second, the SELECTION at this step has the most detailed dimension values that satisfy the user's conditions on those given dimensions (e.g., SELECTION(Dim.DimID = x AND Dim1.Dim1ID = y OR ...) V, such that x and y are all DimIDs and Dim1IDs that satisfy the user's conditions on dimensions Dim and Dim1 respectively). We access the Berkeley database Hilbert R-tree index of view V, and use the Linear Breadth First (LBF) Search algorithm to efficiently answer the SELECTION operator. We stress that the initial LBF pre-dates the work in this research and answers very simple range queries. However we will soon see how the initial Sidera LBF is enhanced to answer complex range queries.

Finally, if no indexes are available for dimension tables and views, then we can answer the SELECTION operation by sequentially scanning dimension tables and views to find those rows that match the condition.

B. Physical Operators For Selection

We explained how the SELECTION operation is resolved. Specifically, we first use the hierarchy manager and the bitmap index manager to convert the user's condition to a condition that is in turn answered by accessing the appropriate R-tree index view/cuboid. Consider SELECTION(D(Cond)) C. Cond is a user's condition of the form A OP c, where A is an attribute of dimension D, OP is a comparison operator (IN LIST, >, <, etc), and c is a constant or list of constants. C is the index cuboid/view that returns cells satisfying the user’s condition Cond. We simply replace SELECTION(D(Cond)) C, by the following physical operators:

- If (A is a non-hierarchical attribute of dimension D) THEN F = bitMapAccess(D, A OP c) ELSE F = mapGraph(D, A OP c)

BerkeleyRtreeAccess (C, F). Here F is a set of dimension IDs satisfying the user’s condition on dimension D, C is the Hilbert R-tree index cuboid needed to answer the query.

C. In-Memory Hash Table Representation

As was discussed in the previous section, some of our OLAP physical operators require an in-memory hash table data structure for efficient searching and inserting. In practice, the entry of a hash table is of the form (k,v), where k represents the search key of the hash table and v its associated value [56]. In our case, the value of the search key k is the value(s) of the feature attributes that will be in the result of a given OLAP operator, while v is the value of the measure attributes. In general, a hash table consists of an array of size N, and a hash function h that maps values of a given type (string, array of integers, etc.) to integers between [0, N-1].

In our case, for each physical operator that needs an internal hash table to be executed, we create a hash table (hT) of size N, where N is equivalent to the cardinality product of the result of the OLAP operator, and a hash function h that maps the values for one or more feature attributes to a specific integer between 0 and N-1. Algorithm-1 shows an implementation of our hash function. The input of the algorithm consists of a list of feature attributes fA, an array of cardinality products (aCP) and an array (aV) that possesses the values of the feature attributes to be mapped to an integer between [0, N-1]. Let the list of feature attributes be of the form fA = {f1 , f2 , . . ., fi , . . ., fn }, where n is the number of feature attributes in the result of a given OLAP physical operator. We can thus say that aCP can be written as {CP f1, CP f2 , . . ., CP fn }, where the value of CP fi represents the cardinality product of all subsequent feature attributes {fi+1 , fi+2 , . . ., fn }. Note that CP fn equals 1. aV has n values {aV1 , aV2 , . . . , aVn }, one value for each feature attribute fi in fA. It is crucial for one to maintain the exact sequential order of the numerical values in aV as they each represent a specific feature attribute. Algorithm returns the hash key for the values of the feature attributes (aV). Our hash function ensures O(1) processor running time for searching, inserting and deleting entries from the hash table. Moreover, the example below will illustrate how our hash function ensures that the entries of the hash table are sorted according to the list of attributes in fA.

```
Algorithm-1: Hash Function Algorithm
Input:    List of Feature attributes fA{d1.d1ID,
          d2.d2ID, . . .,dn.dnID} where n is the number
          of feature attributes of the result, a list of
          cardinality products aCP{CP1(d2.d2ID,
          d3.d3ID, . . ., dn.dnID), CP2(d3.d3ID,
          d4.d4ID, . . ., dn.dnID), . . ., CPn(1)}, and the
          values of the feature attributes is v(d1ID,
          d2ID, . . ., dnID)
Output:   An integer x between 0 and N-1, where N is
          the cardinality product for attributes in fA.
1         Initialize x to 0
2         for each feature attribute in array fA stored at
          index i do
3         x = x + (v[i]-1) * CP[i]
4         end for
5         return x
```

Let us assume that we need to find the hash value of the following set of feature attributes (CustomerID, StoreID, ProductID) (3, 10, 5). The input of Algorithm is:

- fA = {Customer.CustomerID, Store.StoreID, Product.ProductID}
- Array aCP of cardinality products. aCP = {600, 50, 1}, 600 is the cardinality product of (StoreID, ProductID), while 50 is the cardinality product of ProductID.
- Array aV is the values of the feature attributes in fA, aV= {3, 10, 5}. In this case, 3 is the value of CustomerID, 10 is the value of StoreID and finally 5 is the value of ProductID.

Using the hash function outlined in Algorithm, the hash value of key (3,10,5) is: (3-1) * 600 + (10-1) *50 + 5-1 = 1200 + 450 + 4 = 1654 < 3000. This means that key(3,10,5) is stored in the array hA at the index of 1654.

D. Index Based Selection Algorithm

Algorithm-2 is an algorithm applied to answer the SELECTION operator efficiently. Before we access the indexed cuboid/group-by to return the cube cells that satisfy the query restriction, we transform the user's query constraints that are specified on the attributes of the dimensions into the most detail-oriented level. Algorithm utilizes a function called transformSELECTION() to convert the user's query restriction into a most detail-oriented value that can be utilized by our OLAP query engine. After this process, we open the Berkeley DB database object that represents the appropriate Hilbert R-tree index for the group-by (e.g., called V) to answer the selection operator. Finally, a processSelection() is applied which uses the Hilbert R-tree index for view V to answer the transformed user's condition and return the result.

```
Algorithm-2: SELECTION Algorithm
Input:    A user-defined OLAP selection condition dC, a hierarchy manager (mapGraph)
          containing the hierarchical attributes data, a cube C, an appropriate view V
          to answer the SELECTION operator, and a bitmap index manger biM that contains
          the bitmap indexes for the needed non-hierarchical attributes.
Output:   Fully resolved SELECTION (I with all detailed level values satisfying dC).
1         create a new array OP of size n, where n is the number of logical operators
          (AND and OR) that are used to form compound conditions, each associated with
          a dimension.
2         Use dC to get those logical operators and store them in OP.
3         Invoke transformSELECTION(dC,mapGraph, biM)
4         Open the Berkeley database object called db that contains the Hilbert R-tree
          index for group-by V .
          db.open(NULL, C, V , DB-RTREE, DB RDONLY, 644);
5         get result I from disk, I = processSelection(dC, db, OP)
```

The primary focus of Algorithm is to replace the user's query restrictions that are specified within the SELECTION operator into other restrictions (dC) that can be solved against the indexed data stored in the physical cube. As was illustrated, a SELECTION(Dim.A OP c) View is translated into a SELECTION(Dim.DimID = x)View where x is a set of DimIDs satisfying the condition (Dim.A op c).

```
Algorithm-3: SELECTION Transformation Algorithm
Input:    A user-defined OLAP selection condition dC, a hierarchy manager mapGraph,
          OP array of logical operator, and a bitmap index manager biM.
Output:   The user's condition in the most detail-oriented form (primary key form).
1         for each dimension condition Ci in dC do
2         for each expression ej in Ci do
3         if attribute (A) involved in ej is a hierarchical attribute level then
4         arrayj = mapGraph.getBaseID(A, ej)
5         Else
6         arrayj = biM.getBaseID(A, ej)
7         end if
8         if Logical operator between ej and ej−1 equals AND then
9         arrayj = setIntersection(arrayj , arrayj−1)
10        Else
11        arrayj = setUnion(arrayj , arrayj−1)
12        end if
13        end for
14        create a new range array newR of size |arrayj |
15        store integer values in arrayj as a sorted set of contiguous ranges
16        Remove the current SELECTION condition Ci and replace it with Di.DiID =newR
          such that newR has all IDs that satisfy condition Ci associated with Di.
17        end for
```

III. RESULTS AND DISCUSSION

Cost of the Selection Operation

We must be able to estimate the cost of each OLAP physical operator that we use in the physical OLAP query plan. It is well-understood that it is slower to retrieve data from a disk than do anything with the data once it is in the main memory. Therefore, we use the number of disk I/O to estimate the cost of an OLAP operation. However, we shall also mention the processor running time when the amount of process time is proportional to a specific variable (i.e., n2).

The input argument for the SELECTION operator is a Hilbert packed R-tree indexed group-by stored as a Berkeley DB database object on disk. Also, the SELECTION requires the data of non-hierarchical and hierarchical attributes in order to convert the user's query restriction to the most detail-oriented level form restriction. At run-time, the enhanced mapGraph hierarchy manager is used to represent the data of hierarchical attributes. In addition, we create another in-memory index manager called the Bitmap Index Manager to represent the data of each required non-hierarchical attribute in the SELECTION operator. We also assume that we have enough memory to store those two managers (mapGraph and indexManager).

The result of the SELECTION is left in memory unless it is required to be returned to the disk. It is important to mention that the sorted arrays that are used to store the set of contiguous ranges are left in memory as well, until the SELECTION operation terminates. Recall that the sorted arrays represent the query restriction in the most detailed level form.

Theorem-1: The cost of the SELECTION operator is bounded as the cost of sequentially scanning B(V) and D(V), where V is the appropriate packed R-tree index to answer the SELECTION, B(V) is the number of index blocks, and D(V) is the number of disk blocks. Cost = B(V) + D(V) I/O.

Proof: SELECTION uses the Linear BFS strategy to retrieve records that satisfy its condition. LBFS uses a top-to-bottom/left-to-right search pattern for the packed R-tree indexed cube. The indexed cube is stored physically on disk per consecutive disk IDs, using the same top-to-bottom/left-to-right fashion. Also, the data blocks follow this ordering. The worst case is to scan sequentially all index blocks and data blocks. Number of Disk I/O is B(V) + D(V) blocks.

We note, however, there is also a large amount of processor time that may aff ect our assumption that only the disk I/O time is significant. If the condition of the SELECTION has k distinct feature attributes, then k sorted arrays are used to store IDs that satisfy the user's condition, where the larger sorted array has n IDs. We also assume that D(V) has m records (cells).

Theorem-2: The worst case processor running time of the SELECTION operator has a bound of O(m * log(n)).

Proof: In the worst case, we scan sequentially all index blocks and data blocks of view V. For each index block b, we perform a binary search to check if it intersects the selection condition that is stored as a set of sorted arrays. The worst case processor running time for the index scan is k * log(n) * B(V). Also, in the worst case, for each record (cell) of V we have to perform a binary search to check if it intersects the selection condition. The worst case running time for the data scan is k * log(n) * m. Finally, the worst case processor running time is k * log(n) * B(V) + k * log(n) * m which can be written as k * log(n) * (B(v) + m). This result can be re-written as k * log(n) * (O(m)) because m, number of records, dominates the number of index blocks. Finally, since k represents a small number of feature attributes, the worst case running time can be bounded as O(m * log(n)) in practice.

The cost of the SELECTION algorithm can be determined by the sums of (a) the disk I/O and (b) the processor running time, as follows:

1. The worst case number of disk I/O is B(V) + D(V) disk I/O.
2. The worst case processor running time is O(m * log(n)).

In practice, we observe that for most queries the number of disk I/O dominates the processor running time. The processor time still has some effect on the total execution time.

IV. CONCLUSION

In this paper, we have presented number of algorithms for execution of the operations of our OLAP algebra. These algorithms build upon the efficient OLAP Sidera data storage and data structures. Moreover, the query engine uses an in-memory hash table structure that allows efficient implementation of these algorithms. In the next chapter, we will discuss various experimental results that support the design decisions that we have made.

In summary, our OLAP query processor complements the efficient OLAP storage engine and the OLAP query grammar and algebra by providing the final piece that the Sidera DBMS requires in order to support high performance OLAP DBMS within the ROLAP environment.

V. REFERENCES

[1] Oracle essbase. http://www.oracle.com/us/solutions /ent-performancebi/business intelligence/ essbase/ index.html.

[2] T. Eavis and R. Sayeed. High performance analytics with the r3-cache. *Data Warehousing and Knowledge Discovery (DaWak)*, 2009.

[3] D. Kossmann J.-P. Dittrich and A. Kreutz. Bridging the gap between olap and sql. *In International conference on Very Large Data Bases (VLDB)*, pages 1031–1042, 2005.

[4] Thomas P. Nadeau and Toby J. Teorey. Olap query optimization in the presence of materialized views. *HICCS*, 2003.

[5] O. Romero and A. Abello. On the need of a reference algebra for olap. *In International conference on Data warehousing and Knowledge Discovery (DaWak)*, pages 99–110, 2007.

[6] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree - a dynamic index for multidimensional objects. *VLDB*, pages 507–518, 1987.

[7] Sas olap server. http://www.sas.com/technologies/dw/storage/mddb/index.html.

[8] Olap4j. http://www.olap4j.org

[9] Olapdml. http://oracle.com