OPTIMIZING ACCESS ACROSS HIERARCHIES IN DATA WAREHOUSES — QUERY REWRITING

ICS 624 FINAL PROJECT

MAY 2011

By

成玉
Cheng, Yu

Supervisor:
Lipyeow Lim, PhD

Contents

List of Figures

1 Introduction

In data warehouses and on-line analytical processing (OLAP) environments, each dimension is associated with multiple application-specific hierarchies. This is due to the fact that different departments in an enterprise define different hierarchies on the same dimensional data. Usually, for a particular application-specific hierarchy, precomputed aggregates for some internal nodes are cached. A new solution was proposed to optimize the performance by detecting common sub-structures among hierarchies and rewriting aggregation queries to exploit precomputations across hierarchies.

In this project, we continue the work with the proposed system. We implemented the query rewriting module and conducted a series of benchmarks to test its performance.

1.1 Data Warehousing

In enterprise data warehouses, an OLAP data mart consists of a fact table and several dimensions. Each dimension is associated with a primary hierarchy and multiple complex and unbalanced application-specific hierarchies. The following diagram illustrates an example of a fact table and its multiple dimensions.

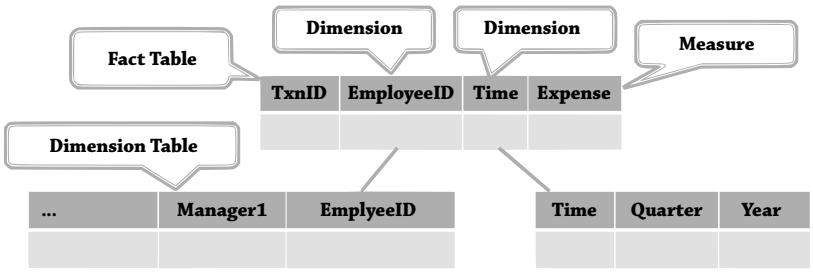

Figure 1: Data Warehouse Fact Table with Multiple Dimensions

1.2 Primary Hierarchy

A set of primary hierarchies are defined on each common fact table. Each fact table dimension is associated with one primary hierarchy. Potentially, every node in the primary hierarchy can be queries for its corresponding data. This include both the leaf nodes and the internal nodes. The following diagram demonstrates an example of a primary hierarchy, which is for the organization dimension.

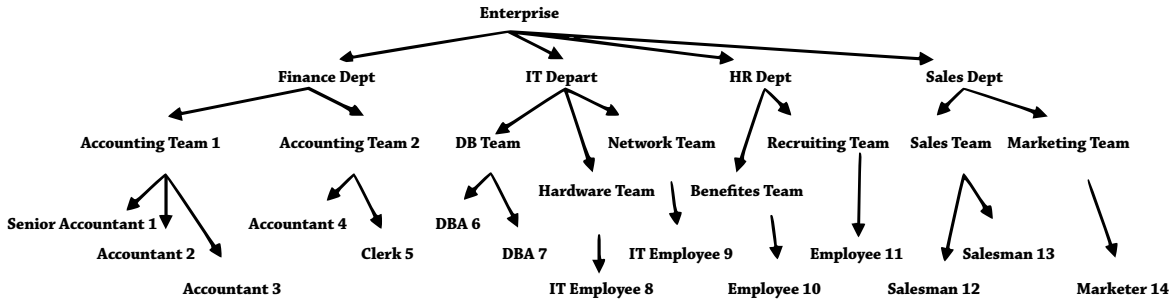

Figure 2: A Primary Hierarchy

1.3 Application-Specific Hierarchy

A large number of application-specific hierarchies exists. Each fact table is associated with multiple application-specific hierarchies. The leaf nodes of these hierarchies refer to nodes in the dimensional primary hierarchy. The internal nodes indicate different ways of grouping the data based on the specific applications. So, only the leaf nodes of the application-specific hierarchies are there to be queried. The following diagram demonstrates two examples of application-specific hierarchies against the same primary hierarchy. T primary hierarchy shown in Figure 2. These two application-specific hierarchies are geography based and project based.

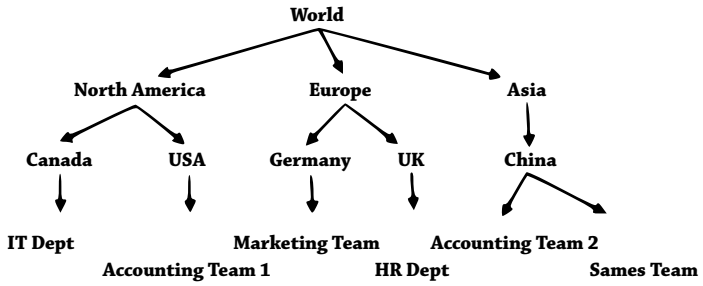

Figure 3: An Application-Specific Geography Hierarchy

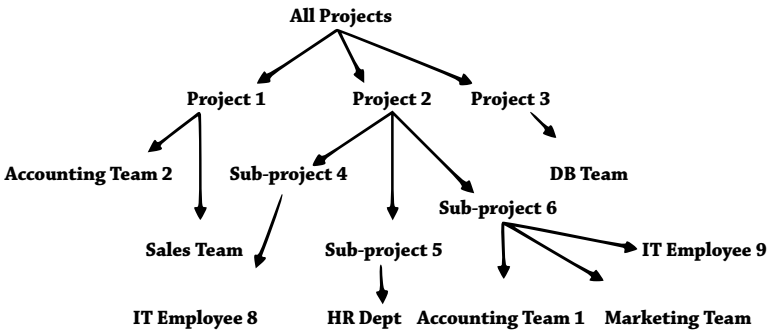

Figure 4: An Application-Specific Project Hierarchy

1.4 Current Querying Strategy

The current OLAP querying strategy of optimizing the performance involves pre-computing aggregates for some internal nodes of specific application hierarchies. There are two main problems with such a strategy. One, this pre-computation is unfeasible for all internal nodes of all application hierarchies. Two, OLAP query engines cannot exploit precomputed aggregates across hierarchies.

The following diagram demonstrates an example of the second drawback scenario for this current strategy. If the aggregate for $Accounting\ Team\ 2 + Sales\ Team$ is already explored in the geography hierarchy, when trying to query them again in the project hierarchy, the OLAP query engine should be able to recognize that they are isomorphic sub-structures, and hence use the precomputed result from the geography hierarchy in the project hierarchy.

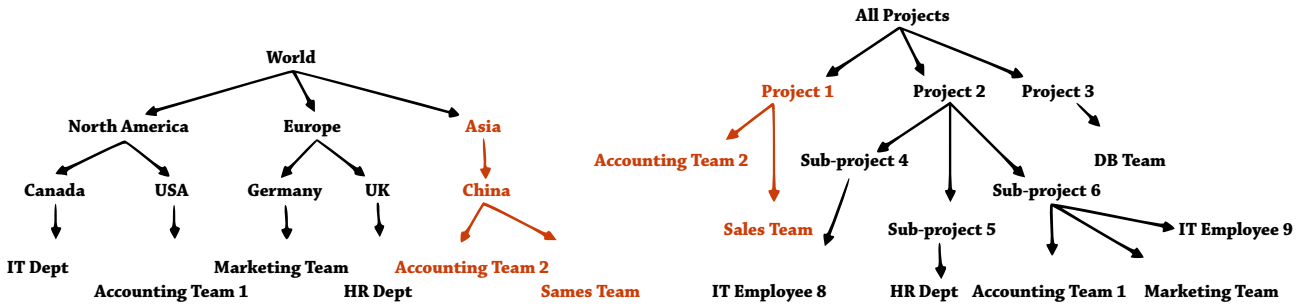

Figure 5: Overlap Sub-Structures

2 Proposed Solution

The proposed OLAP query optimization system involved two phases. The system first detects common sub-structures in different application-specific hierarchies. This is done off-line. The system then rewrites the aggregation queries in the on-line phase, and hence exploit any precomputation on these shared sub-structures. This system results in significant savings of computation and faster query response times.

2.1 Phase I – Overlap Discovery

The goal for the first phase is to discover and store overlapping relationships in a forest of application-specific hierarchies within an OLAP environment. The input for this phase are a primary hierarchy and

a set of application-specific hierarchies. The output for this phase is a catalog table containing the overlapping information between application-specific hierarchies. For the overlapping sub-structure shown in Figure 5, the catalog table would contain the following entry.

Sub-Hierarchy I	***Sub-Hierarchy II***
Geography-Hierarchy-Asia	*Project-Hierarchy-Project1*
:	:

Figure 6: Catalog Table Entry

2.2 Phase II – Query Rewrite

The goal for the second phase is to rewrite aggregation queries by taking advantage of the discovered overlapping sub-structure relationship among application-specific hierarchies. The input for this phase are an aggregation query, an application-specific hierarchy that this aggregation query is going to be executed against, and the catalog table containing the overlapping information of a pre-evaluated forest of application-specific hierarchies. The output of this phase is an alternate formulation of the input aggregation query.

In this implementation, the application-specific hierarchy that the aggregation query is going to be executed against is one of the hierarchies of the pre-evaluated forest of the application-specific hierarchies.

2.2.1 Assumption

In this phase, we will assume that the aggregates of the overlapping sub-structures discovered in phase I are pre-computed. In theory, the catalog table contains only the locations of overlapping sub-structures, and hence direct accesses to these sub-structures. The aggregates, however, are not necessarily precomputed. But in this project, for simplicity, we will assume the aggregates are pre-computed. In other words, this procedure does not contribute towards the execution time of phase II. For the overlapping sub-structure shown in Figure 5, the catalog table would contain the following entry, assuming whatever aggregates we are computing is 172 for this particular sub-structure.

Sub-Hierarchy I	***Sub-Hierarchy II***	***Aggregates***
Geography-Hierarchy-Asia	*Project-Hierarchy-Project1*	172
:	:	:

Figure 7: Catalog Table Entry with Precomputed Aggregates

2.2.2 Tasks

Specifically, in this phase, the system first evaluate the input application-specific hierarchy and retrieve the leaf nodes of the overlapping sub-structures it shares with the other application-specific hierarchies. The system, then, go through the input aggregation query and replace the sections that form the sets of leaf nodes of pre-computed overlapping sub-structures.

3 Program Modules

There are five program modules that are implemented for the proposed system and the performance evaluation, the synthetic forest generating module, the overlapping discovering module, the query rewriting module, the random query generating module and the benchmark module for testing phase II.

3.1 Synthetic Forest Generating Module

This program module generates application-specific hierarchies as an XML file. The generated synthetic data has different sizes and characteristics. The XML file represents a forest of application-specific hierarchies. In this module, we can control the characteristics of the forest by varying the fan-out, depth, probability of expansion, and probability of sharing parameters.

3.2 Overlapping Discovering Module

This program module discovers and stores overlapping relationships in a forest of application-specific hierarchies. It returns a list of node pairs indicating the roots of maximal overlapping sub-hierarchies. By overlapping sub-hierarchies, we mean isomorphic sub-structures shared among hierarchies. Maximal overlapping sub-hierarchies imply that there are no larger overlapping sub-hierarchies with a larger node set. In other words, there is no supersets of the discovered maximal overlapping sub-hierarchy.

3.3 Query Rewriting Module

This program module checks for all possible rewrites and returns an altered aggregation query formulation. It evaluates a input application-specific hierarchy and retrieves the leaf nodes of the overlapping sub-structures it shares with other application-specific hierarchies in a forest. Then it goes through the input query and replaces the sections that form the sets of leaf nodes of pre-computed overlapping sub-structures.

3.4 Random Query Generating Module

This program module generates queries with specified sizes and overlapping properties against pre-computed aggregates. Specifically, three parameters are specified, a catalog table, the size of the query, the percentage value indicating how much this query has in common with the leaf nodes pointed to by the specified catalog table.

3.5 Benchmark Module

This program module connects to an MySQL database. In this database we have one table that is populated test data. This module then sends numerous aggregation queries to the database to execute and collects timing information for performance analysis. This is done on queries with different sized and different overlapping properties. Also, timing data is collected for both the raw queries and the re-written queries.

4 Benchmarks

Experiments were designed and conducted to test the performance of the query re-writing module, phase II of the proposed OLAP query engine optimization system. The main goal is to evaluate whether or not the query rewriting would dominate the execution time of aggregation queries. Also, we would like to know how much does the query-rewriting help, and what kind of aggregation queries obtain significant help.

The timing results are collected for executing aggregation query, raw and re-written, against a dummy database. The following are examples of the dummy database, the aggregation query, and the query results

id	*name*	*value*
:	:	:
27	L45	35
:	:	:
82	L57	79
:	:	:
197	L34	21
:	:	:

Figure 8: Example Table in the Database to be Queried

Aggregation Query:	L34, L45, L57.
Query Result:	$21 + 35 + 79 = 135$

Figure 9: Example Aggregation Query and Result

4.1 Data Preparation

We first create a database and a table in the database. This table contains dummy data. In this table, we have three columns. Other than the primary key column "id", we have "name" as the leaf node label, and "value", a dummy numerical value representing the data for a particular leaf node.

```
CREATE TABLE db1.table1 (
    id INTEGER NOT NULL DEFAULT NULL AUTO_INCREMENT,
    name TEXT CHARACTER SET latin1 NOT NULL,
    value INTEGER NOT NULL,
    PRIMARY KEY (id))
ENGINE = MyISAM
ROW_FORMAT = DYNAMIC;
```

Figure 10: Create a Table with Test Data

We then populate the table with experimental data. For a given catalog table, all leaves (e.g. L57) are inserted with dummy data (e.g. 36). For a given set of randomly generated queries, all necessary leaves are inserted. More dummy data is inserted, and all data is shuffled to avoid sequential access. For each entry to be added to the test data table, the following query is sent to the database to execute.

```
INSERT INTO table1 (name, value) VALUES ('L57', 36);
```

Figure 11: Insert an Entry to the Test Data Table

Before conducting any experiments, we also need to populate the catalog table cache with aggregates. This is because they are assumed to be pre-computed as we discussed previously. For a given catalog table, we query all leaf nodes of every shared sub-hierarchy that is recorded. We then aggregate "value" of these leaf nodes and store the sum for each shared sub-hierarchy in memory. For each leaf node, the following query is sent to the database to execute

```
SELECT value FROM table1 WHERE name = 'L57';
```

Figure 12: Query a Given Leaf Label in the Test Data Table

4.2 Results

In each experiment below, a set of aggregation queries were generated. For each aggregation query, we query all leaves, aggregate "value", and return the sum. This is done on both the set of raw queries and the set re-written queries. The two results returned by a raw query and its re-written query are verified to be the same.

4.2.1 Overlapping Property

Three sets of 1000 random aggregation queries were generated. Each query is 1000 nodes long and has a small (10%), medium (50%), or large (90%) overlapping percentage with the given catalog table. Both the raw and re-written queries were executed and the time of execution was recorded.

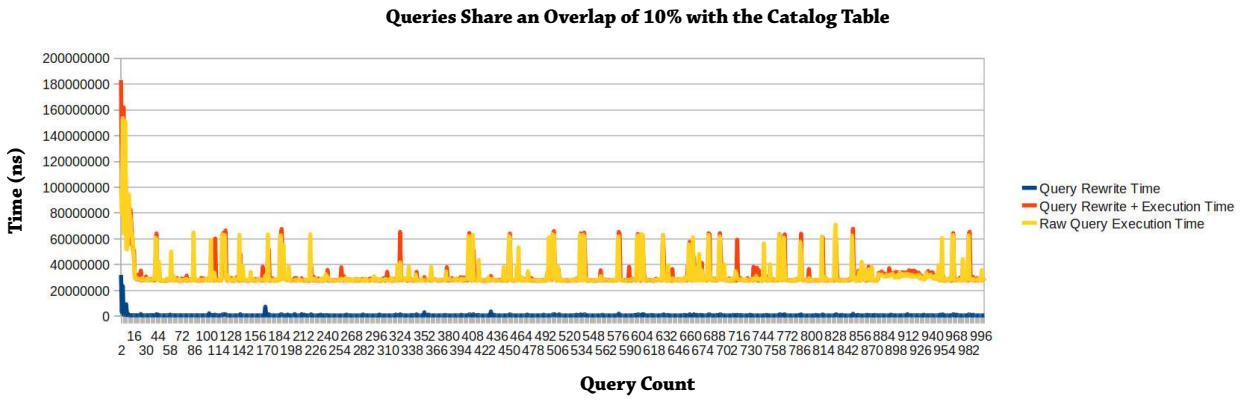

Figure 13: Results of Queries with Small Overlap with the Catalog Table

We observe here that re-writing queries does not dominate the query execution time. Queries with a small overlap property do not take advantage of the catalog table as we do not see a difference in execution time between the raw queries, shown in yellow, and the re-written queries, shown in red.

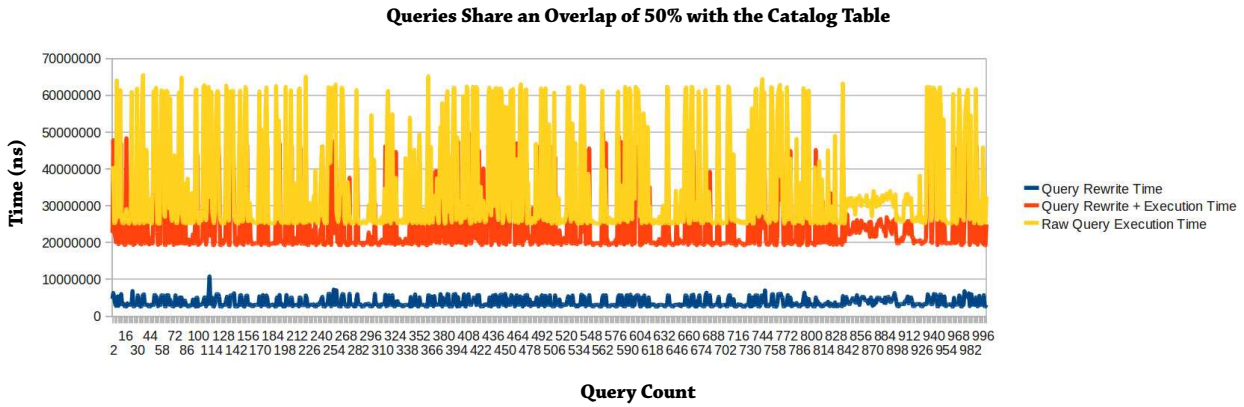

Figure 14: Results of Queries with Medium Overlap with the Catalog Table

We observe here that the benefit of re-writing queries start to become visible. We can see that the re-written queries executed noticeably faster. The base line of the red curve is noticeably lower than the base line of the yellow curve.

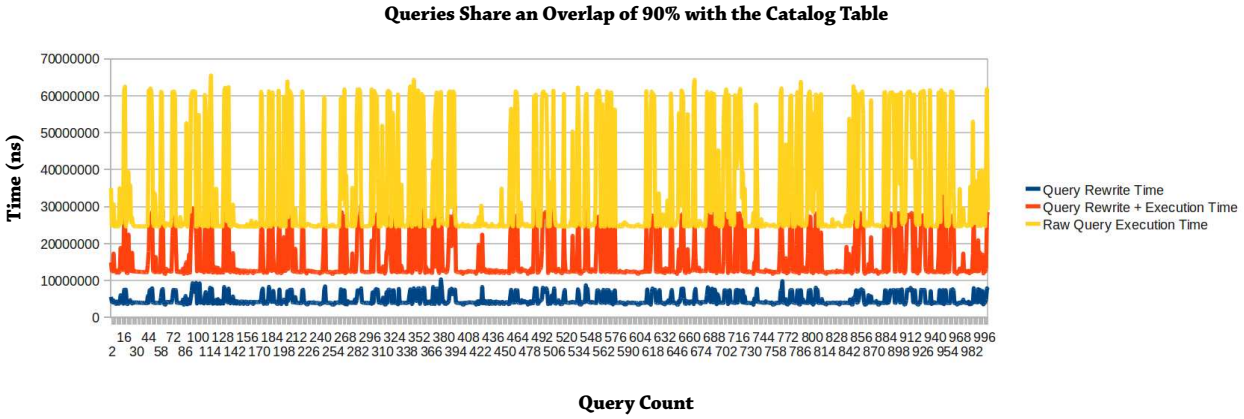

Figure 15: Results of Queries with Large Overlap with the Catalog Table

We observe here that the re-written queries executed considerably faster than the raw queries. The amount of time spent to re-write a query stays the same and this time does not dominate the total execution time.

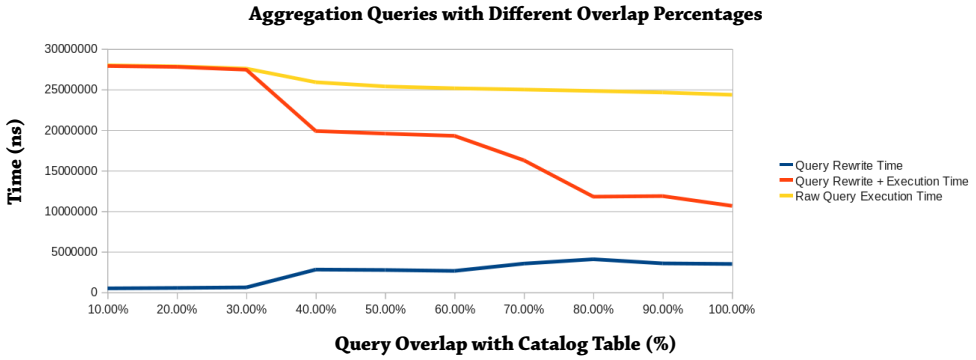

Figure 16: Results of Queries with Different Overlap Properties with the Catalog Table

From the summarized view shown above, we see that the time for query re-writing, shown in blue, does not dominate the query execution time, shown in red, in the proposed system. The blue curve plateaued as the overlap property increases. At the same time the total execution time, shown in red, dropped steadily, and it showed obvious improvement comparing to the the execution time for the raw queries.

4.2.2 Aggregation Query Size

Two sets of 1000 random aggregation queries were generated. Each query has a 90% overlap with the given catalog table, and has a size of 50 or 25000 nodes long. Both the raw and re-written queries were executed and the time of execution was recorded.

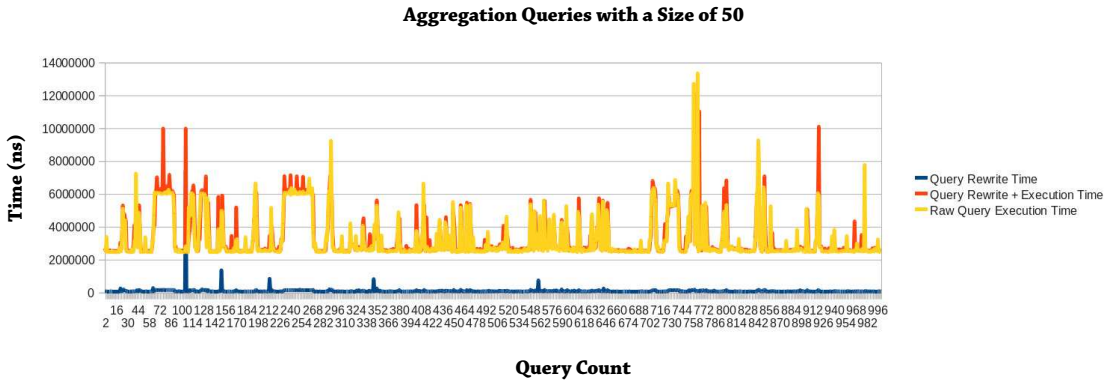

Figure 17: Results of Queries with Small Size

We observe here that re-writing queries does not dominate the query execution time in the proposed system. We also notice that queries with small sizes do not take advantage of the catalog table as we do not see a difference in execution time between the raw queries, shown in yellow, and the re-written queries, shown in red.

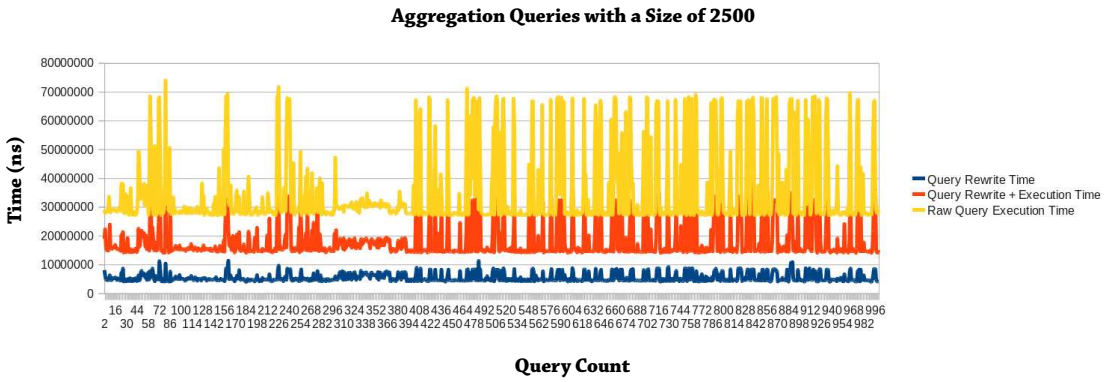

Figure 18: Results of Queries with Large Size

We observe here that re-writing queries significantly improved the performance of execution time on large queries. The base line of the red curve is significantly lower than the base line of the yellow curve.

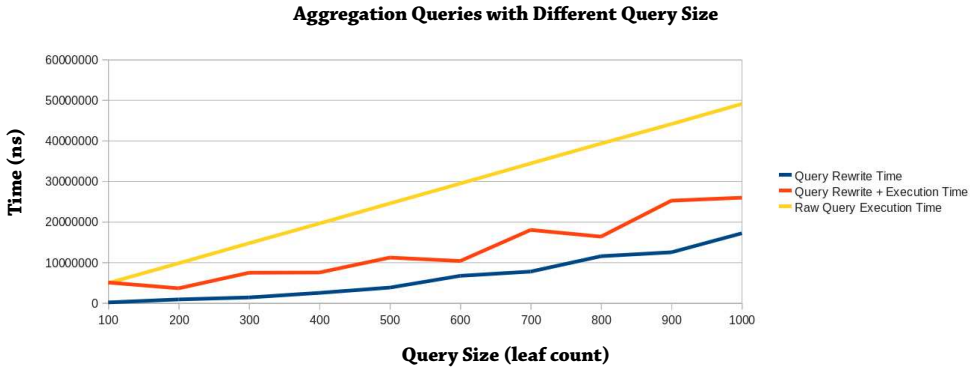

Figure 19: Results of Queries with Different Sizes

From the aggregated view shown above, as expected, we see a linear increase in the execution time for the raw queries. The increase in execution time for the re-written queries is not as dramatic as the raw queries. The query re-writing improves the performance for larger aggregation queries.

5 Conclusions

We identified a problem in the current OLAP environment. It is caused by uncontrolled proliferation of application hierarchies in enterprise data warehouses, and the inability of OLAP query engines to exploit aggregates across different hierarchies. We proposed a solution to improve this situation. The proposed system first finds hierarchy overlap information from an application-specific hierarchy forest, and then re-writes aggregation queries to exploit overlaps and use pre-computed aggregates.

Previous work has done to implement and test the module that searches and stores sub-hierarchy overlapping information. This procedure was shown to be feasible. In this project, we implemented the query re-writing module and conducted benchmarks to test the performance of this module. The results showed that query re-writing does not dominate the execution time of aggregation queries; query re-writing significantly improves the performance of queries with large overlaps; and query re-writing significantly improves the performance of large queries.

For possible future work, we would like to design and conduct end-to-end experimental evaluation with both phases. We would also like to experiment on large forest with large catalog table that requires storage in database. Also, we would like to improve the caching mechanism, e.g., caching intermediate sub-hierarchies.