The Cubetree Storage Organization

Nick Roussopoulos & Yannis Kotidis
Advanced Communication Technology, Inc.
Silver Spring, MD 20905
Tel: 301-384-3759 Fax: 301-384-3679
{nick,kotidis}@act-us.com

1. Introduction

The Relational On-Line Analytical Processing (ROLAP) is emerging as the dominant approach in data warehousing. In order to enhance query performance, the ROLAP approach relies on selecting and materializing in summary tables appropriate subsets of aggregate views which are then engaged in speeding up OLAP queries. However, a straight forward relational storage implementation of materialized ROLAP views is immensely wasteful on storage and incredibly inadequate on query performance and incremental update speed.

ACT Inc. has developed a unique storage organization for on-line analytical processing (OLAP) on existing relational database systems (RDBMS). The Cubetree Storage Organization[1] (CSO) logically and physically clusters data of materialized-views, single- and multi-dimensional indices on these data, and computed aggregate values all in one compact and tight storage structure that uses a fraction of the conventional table-based space. This is a breakthrough technology for storing and accessing multi-dimensional data in terms of storage reduction, query performance and incremental bulk update speed. Each Cubetree structure is not attached to a specific materialized view or an index but acts like a place holder which stores multiple and possibly unrelated views and multi-dimensional indices without having to analyze and identify the dense and sparse area of the underlying multi-dimensional data. In CSO tuples are clustered regardless of the skewedness of the underlying data. Given a set of views that need be supported, the CSO optimizer finds the best placement on one or more cubetrees for achieving maximum clustering. The underlying cost model is tailored to the CSO and uses standard greedy algorithms for selecting the optimal configuration. The DBA can tune the optimization to upper-bound either the duration of the incremental update window or the maximum storage used.

CSO offers a scalable and yet inexpensive alternative to existing pricey data warehousing and indexing tools. CSO is developed to be used as a stand-alone system, or as a co-resident of an RDBMS or tightly coupled with the extensible facilities of an Object-Relational DBMSs.

2. OLAP and the Data Cube

On Line Analytical Processing (OLAP) is critical for decision support in today's competitive and low margin profits era. The right information at the right time is a necessity for survival and data warehousing industry has an explosive market growth compared to a stagnant one for the database engines. The data cube is a multi-dimensional aggregate operator and has been very powerful for modeling multiple projections of data of a data warehouse. It is, however, very expensive to compute. And to access the some or all of these projections efficiently, requires extensive multi-dimensional indexing which adds an equal or even higher cost to the data warehouse maintenance cost. Computing the data cube requires enormous hardware resources and takes a lot of time. Indexing can be done using either a multi-dimensional (MOLAP) approach or a Relational (ROLAP) modeling exemplified by star-schemas and join bitmap indexes. Both approaches speed up queries but suffer enormously in maintenance of the indexes which essentially takes the data warehouse out for an extensive period. The downtime window is critical in applications globally available for the

[1] US Patent Pending

most of the day.

1. Performance- the ultimate reality check

The CSO avoids the high cost of creation and maintenance by sorting and bulk incremental merge-packing of cubetrees. The sorting portion of it is the dominant factor in this incremental bulk update mode of operation while the merge-packing portion achieves rates of 5-7 GB per hour on a single disk I/O. No other storage organization offers this scaleable bulk merge-packing technology. Experiments with various data sets showed that multi-dimensional range queries perform very fast and this performance is the result of the packing and compression of CSO and the achieved reduction of disk memory over conventional relational storage and indexing. In a recent test using the TCP-D benchmark data the CSO implementation achieved at least a 2-1 storage reduction, a 10-1 better OLAP query performance, and a 100-1 faster updates over the conventional (relational) storage organization of materialized OLAP views indexed in the best possible way.

The CSO optimizer selects the best set of views to materialize and the best placement on one or more cubetrees for achieving maximum clustering. The underlying cost model is tailored to the CSO and uses standard greedy algorithms for selecting the optimal configuration. The DBA can tune the optimization to upper-bound either the duration of the incremental update window or the maximum storage used. Another feature of the CSO software is that the incremental maintenance can be done on-line while the data warehouse is operational using the previous version of the cubetrees while creating the new ones. Although such a background continuous maintenance steels I/O from query processing, it achieves no down-time for the warehouse and can take advantage of low-peak periods.

4. The Informix CSO Datablade

The CSO has also been also developed as a datablade for the Informix Universal Server (IUS) Environment. IUS and CSO harmoniously inter-operate in a tightly coupled configuration that has no overhead. This architecture takes advantage of the extensibility of Object-Relational database management system (O-R DBMS) and allows to include user-defined aggregate and statistical functions not normally offered by RDBMSs. The CSO datablade offers all the advantages of the stand-alone CSO software, namely efficient OLAP aggregate queries and bulk-incremental update on them, but at the same time transparent calls of the CSO software from within the IUS SQL environment. Results from the CSO search are merged with results from the IUS database.

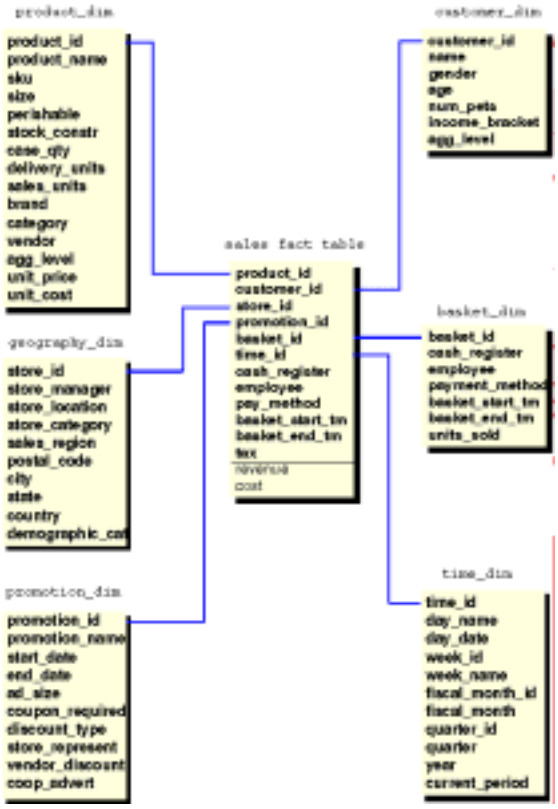
5. Extensibility

The CSO has been developed with extensibility in mind. New data types for the dimension and measure attributes can be facilitated. Similarly, the aggregate functions on the measure attributes can be user-defined for specialized computation. Such functions can return either a scalar or a vector value. Additional indexing on the aggregate values themselves can also be stored in the same cubtree storage. Other functions that extend the SQL have been developed and incorporated into the Cubetree SQL some of which are illustrated in the examples given in the appendix.

6. A Grocery Benchmark

This benchmark uses a synthetically generated grocery demo dataset that models supermarket transactions. The data warehouse is organized according to the star schema below. The sales fact table includes 12 dimension attributes and two measure attributes, namely revenue and cost. The hardware platform used for this benchmark is a 180MHz Ultra SPARC I with a single SCSI 9GB hard drive.

We have precomputed seven aggregate tables as materialized views and stored them using the Cubetree Storage Organization. These views aggregate data over some attributes chosen from the fact and the dimension tables and compute the sum function for both measures The following table summarized the characteristics of the stored data:



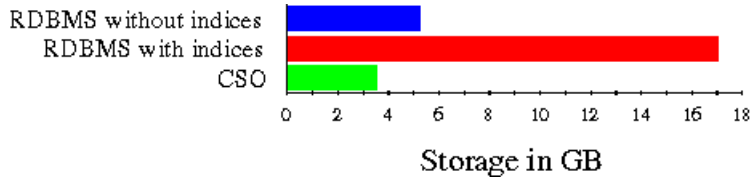
View Name	Gross revenue & cost by:	Number of dimensions	Number of rows
view 1	State	1	49
view 2	store, state	2	2,000
view 3	product, store	2	17,158,551
view 4	Customer, product, store	3	39,199,723
view 5	pay method, time, store	4	29,382,902
view 6	store, product	2	10,000
view 7	store, product, customer, pay method, employee, category	6	40,067,142
Total		12	125,820,367

Loading the views data into the cubetrees requires sorting it first in the CSO order and bulk-packing it into the cubetrees. The following table shows the time for the initial load of the cubetrees and the time for each bulk-incremental update of approximately 0.6 GB of data. In the initial load of the views the dominant cost is the sort time as it was done on a single disk with no parallel I/O. Bulk-packing and incremental bulk merge-packing of of the CSO utilizes almost the full disk bandwidth, roughly 6 GB per hour on a single disk.

Increment (million rows)	Total rows	Sort time (hours:mins:secs)	Pack time (hours:mins:secs)	Cubetree size (GB)	Packing rate (GB/Hour)
88.2	88.2	2:07:04	0:22:21	2.40	6.44
18.2	106.4	0:25:18	0:33:01	2.93	5.33
19.4	125.8	0:27:20	0:40:22	3.50	5.20

To emphasize the storage advantage of the CSO, we loaded summary tables as materialized views on a commercial RDBMS using plain unindexed relational tables to store the data. The space requirements of the views stored as tables was 5.15GB compared to a total of 3.5 for the CSO, i.e. 47% more. The CSO at the same time provides full indexing capabilities for the stored OLAP data, whereas to make the summary tables of acceptable query performance they require indexing that adds more than three times the raw data.

This graph illustrates the storage required in each of these three storage configurations:



OLAP queries using the CSO are very fast because they take advantage of the unique clustering of the data. The retrieval has almost zero *noise to data ratio*, that is after the first record satisfying the view is found, the remaining records are very close with not much of noise data interleaved with useful data. CSO utilizes both the high sequential transfer speed rate of the storage and the clustering of the data. The table below is a small sample of queries, their result size, and the real time of execution for the grocery benchmark.

QUERY DEFINITION	# OF ROWS IN VIEW	# OF ROWS RETURNED	TIME (SECS)
select product_id, store_id, sum(revenue), sum(cost) **from** view3 **where** (store_id = 155) **or** (store_id = 156) **group by** product_id, store_id;	17,158,551	17,143	0.8
select store_id, basket_start_tm, basket_end_tm, sum(revenue-cost) **as** profit **from** view5 **where** (store_id >=50) **and** (store_id <=1500) and (pay_method='ATM') **and** (between(basket_start_tm,'12:00','12:40')) **and** (between(basket_end_tm,'12:00','12:40')) **group by** store_id, basket_start_tm, basket_end_tm;	29,382,902	234,699	1.87
select customer_id, product_id, sum(revenue) **from** view7 **where** (store_id=901) **and** (pay_method='Cash') **and** (category='FOOD') **and** (employee='Jim_G') **group by** customer_id, product_id	40,067,142	7	0.12
select store_id, product_id, sum(revenue), sum(cost) **from** view4 **where** (customer_id=5561) **group by** store_id, product_id	39,199,723	260	0.06
select store_id, basket_start_tm, basket_end_tm, sum(revenue), sum(cost) **from** view5 **where** (pay_method='Check') **group by** store_id, basket_start_tm, basket_end_tm;	29,382,902	7,344,419	50.21
select store_id, sum(revenue-cost) **as** profit **from** view2 **where** (state='MD') **group by** store_id;	2,000	41	0.01

Appendix: The Cubetree SQL

The Cubetree SQL is easy to use and in many cases linguistically simpler than standard SQL. To create a new cubetree the user can execute the following procedure from an SQL interface:

execute procedure create_cube*(<cube name>,<dimension list>,<measure list>, <view list>, <function list>,<target directory>);*

where:

<cube name>:	can be any legal relational table name and is being used for accessing the data later on.
<dimension list>:	is a list of all dimensions along with their data types present in the cube.
<measure list>:	defines the measure attributes. There can be multiple measure values for each record
<view list>:	is a selection of views from the defined dimensions that the user wants to be materialized. If omitted the full Data Cube is created.
<function list>:	defines a set of functions that will be computed for the given measures.
<target directory>:	specifies where the cubetree is being stored.

For example in order to create a three dimensional partial Data Cube in the *store_id, product_id* and *customer_id* dimensions with two measure attributes *revenue* and *cost* for the following views:

View1:	**select**	store_id, sum(revenue) as gross_revenue, sum(cost) **as** total_cost
	from	fact_table
	group by	store_id
View2:	**select**	store_id, store_id, customer_id, sum(revenue) **as** gross_revenue, sum(cost) **as** total_cost
	from	fact_table
	group by	store_id

one can use the following procedure call:

execute procedure create_cube('cubed', ' store_id int product_id int customer_id int', 'revenue float cost float', '((store_id), (store_id, customer_id, product_id))', 'sum(revenue) **as** gross_revenue sum(cost) **as** total_cost', '/data/Cubes');

Updating the Cubetree is extremely simple. Suppose that a temporary table *delta* holds a new set of updates we can make the following call to refresh all views stored in *cubed*:

execute procedure update_cube('cubed','delta');

For greater flexibility *delta* can also be a predefined view that filters-in data for the Cubetree or even an external to the RDBMS source that provides the system with batch updates. Querying the Cubetree is simpler than straight SQL. For example, to find the profit for a given store the Cubetree SQL query is:

select (gross_revenue – total_cost) **as** profit
from cubed
where store_id = 419;

which is equivalent to the straight SQL query:

```
select  sum(revenue-cost) as profit
from    fact_table
where   store_id = 419
group by store_id;
```

Similarly, to find all sales to a customer for a range of products at a given store which generated revenue above a certain value one writes the Cubetree SQL query:

```
select  product_id, gross_revenue
from    cubed
where   product_id>=10000 and product_id<12000
and     customer_id=711 and store_id = 26 and gross_revenue>5000;
```

which is simpler than the the straight SQL query:

```
select  product_id, sum(revenue)
from    fact_table
where   product_id>=10000 and product_id<12000
and     customer_id=711 and store_id = 26
group by product_id
having sum(revenue) > 5000;
```