Extending PostgreSQL with Column Store Indexes

Minoru Nakamura, Tsugichika Tabaru, Yoshifumi Ujibashi, Takushi Hashida, Motoyuki Kawaba, Lilian Harada
Computer Systems Laboratories
Fujitsu Laboratories Ltd.
Kawasaki, Japan
{nminoru, tabaru, ujibashi, hashida, kawaba, harada.lilian}@jp.fujitsu.com

***Abstract* — The importance of database systems to support mixed online transaction processing (OLTP) and online analytical processing (OLAP) workloads, the so-called OLXP workloads, has attracted much attention recently. Some research projects and also commercial database systems with focus on OLXP have appeared in the last few years. In this paper, we present our work aiming at extending the PostgreSQL OSS database system to efficiently handle OLXP workloads. Besides PostgreSQL's traditional OLTP-oriented row data store, a new OLAP-oriented column data store is added in the form of a new index. Unlike previous works focusing on OLXP, our column store index has no restrictions concerning data size and/or updatability. Therefore, transactional data inserted to PostgreSQL row data store become immediately available for efficient query processing using this new column store index.**

Keywords - PostgreSQL; columnar data; OLTP; OLAP; OLXP.

I. INTRODUCTION

Historically, database systems were mainly used for online transaction processing (OLTP) where transactions access and process only few rows of the data tables. Then, a new usage of database systems where queries access substantial portions of the data tables in order to aggregate few columns have evolved into the so-called online analytical processing (OLAP). The execution of OLAP query processing led to resource contentions and severely hurt mission-critical OLTP. Therefore, the data staging architecture where a dedicated OLTP database system whose changes are extracted, transformed and loaded into a separate OLAP database system was used for decades [1].

However, new real-time/operational business intelligence applications require OLAP queries on the current, up-to-date state of the transactional OLTP data [2] [3]. This OLXP workload, i.e., mixed workloads of OLTP and OLAP on the same database, has been extensively addressed recently. Some approaches adopt the columnar data store for both OLTP and OLAP [4] but it is still to be proven that columnar data stores can efficiently support mission-critical OLTP applications. Other hybrid approaches allow a data table to be represented simultaneously in both formats: row data for OLTP and columnar data for OLAP but with some constraints in the columnar data size and its updatability/synchronization with row data [5] [6].

In this work, we propose a new approach to enhance PostgreSQL [7] so that it can effectively handle OLXP workloads without any constraints on data sizes nor updatability. In our approach, columnar data are created as indexes of data tables with no size restrictions and are completely synchronized with row data for any insertions/updates/deletions in the data tables. There are other works that extend PostgreSQL to support columnar data stores for time-series data [8] and OLAP [9]. However, to the best of our knowledge, this is the first work extending the PostgreSQL OSS database to support OLXP.

In section II we describe how data are stored and queries executed using the proposed column store index. Section III shows some preliminary evaluation results. Some concluding remarks are presented in section IV.

II. COLUMN STORE INDEX

In the following, we give a brief description of how we are extending PostgreSQL to efficiently support a column store index.

A. Data store

Updates to PostgreSQL's row data store have to be immediately reflected to the column store index without degradation of the performance of OLTP transactions. Some previous works adopt the Write-Optimized-Storage (WOS) /Read-Optimized-Storage (ROS) approach where the updates are first buffered at WOS in row format and then asynchronously transferred to ROS in columnar format [10]. However, unlike previous approaches, as illustrated in Fig. 1, instead of writing all the row data into WOS, we only write PostgreSQL's original Tuple Identifiers (TIDs) into the Insert List (InsL). Only when transferring data from InsL to ROS, the columnar data values identified by the TIDs are used. For performance reasons, deletions are not in-place, but a Deletion List (DelL) and a ROS delete-vector are used to immediately hide data that are physically deleted later. Data in ROS are grouped in units, called extents, for data management.

We use the same Multi-Version Concurrency Control (MVCC) used in PostgreSQL to guarantee data consistency when transferring data from InsL to ROS (the same data cannot be found at InsL and ROS), and to handle uncommitted transactions (only insert/update/delete of committed transactions are reflected to ROS).

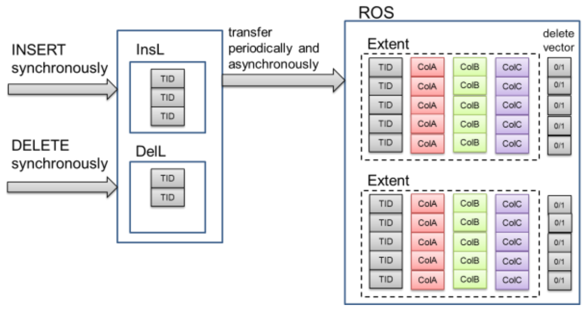

Figure 1. InsL/DelL and ROS Configuration

B. *Query Execution*

When the Query Optimizer chooses to execute a query/subquery using the columnar data instead of the traditional row data, the execution is handled by our new columnar data engine. For each query, the necessary portion of data in InsL is temporarily converted to a columnar data format (called Local ROS) and merged with the ROS data for processing, as shown in Fig. 2.

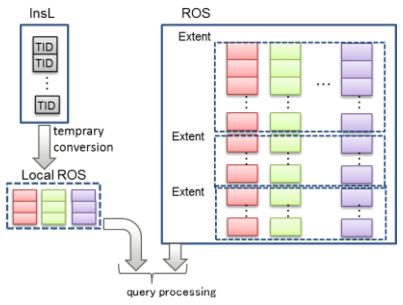

Figure 2. Combining Local ROS and ROS for Query Processing

The extents are processed in parallel by PostgreSQL's Dynamic Background Workers. For an efficient parallel processing, a mechanism where data containing pointers can be shared among the multiple processes is necessary. However, using PostgreSQL's Dynamic Shared Memory, the mapping location is different between the processes and thus it doesn't allow the straightforward sharing of pointers. Aiming at solving this problem, we designed a new shared memory mechanism called Shared Memory Context (SMC). SMC interface is compatible with PostgreSQL's memory context interface. Therefore, pre-existing PostgreSQL's routines can be called, and memory that is newly allocated within those routines is mapped to SMC space. As illustrated in Fig. 3, using SMC, the Backend Process and the Dynamic Background Workers can share the necessary data for an efficient processing of queries in parallel.

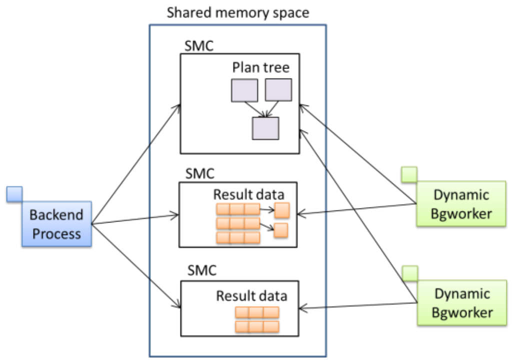

Figure 3. Parallel Processing using SMC

III. Performance Evaluation

Although we are at a preliminary stage of evaluation using the DBT-3 benchmark [11], the results are promising. For instance, for query 1 of DBT-3, a speed-up ratio of 50 was achieved, when using the column store index against PostgreSQL's original row data, on the same server (a 2-CPU machine with 16 cores).

IV. Conclusion and Future Work

In this paper, we have briefly presented our work in progress on extending the PostgreSQL OSS database system with a column store index to handle OLXP workloads. We have introduced new mechanisms to efficiently synchronize the inserts/updates/deletes of row data with the column store indexes, and to efficiently process them in parallel using a new shared memory mechanism that is fully compatible with PostgreSQL's memory context interface.

We are now planning to evaluate the extensions we have introduced to PostgreSQL by using the CH-benCHmark [12] and some real applications.

References

[1] S. Chaudhuri and U. Dayal, "An Overview of Data Warehousing and OLAP Technology", Proc. VLDB, 1997, pp.65-74.

[2] A. Kemper and T. Neumann, "Hyper: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots", Proc. IEEE ICDE, 2011, pp.195-206.

[3] H. Plattner, "The Impact of Columnar In-MemoryDatabases on Enterprise Systems", Proc. VLDB, 2014, pp.1722-1729.

[4] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth", Proc. ACM SIGMOD, 2012, pp.731-741.

[5] "Oracle Database In-Memory", Oracle White Paper, October 2014, Available from: http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html [retrieved: March, 2015].

[6] P. Larson, et al., "Enhancements to SQL Server Column Stores", Proc. ACM SIGMOD, 2013, pp.1159-1168.

[7] "PostgreSQL", Available from: http://www.postgresql.org/ [retrieved: March, 2015].

[8] K. Knizhnik, "In-Memory Columnar Store extension for PostgreSQL", Available from: http://www.pgcon.org/2014/schedule/events/643.en.html [retrieved: March, 2015].

[9] "PostgreSQL Columnar Store for Analytics Workloads", Citusdata, Available from: http://www.citusdata.com/blog/76-postgresql-columnar-store-for-analytics.

[10] Mike Stonebraker, et al., "C-store: a column-oriented DBMS", Proc. VLDB, 2005, pp.553-564.

[11] Database Test Suite, Available from: http://sourceforge.net/projects/osdldbt/files/dbt3 [retrieved: March, 2015]..

[12] R. Cole, et al., "The mixed workload CH-bencCHmark", Proc. DBTest, 2011, article no. 8.