Extending PostgreSQL to handle OLXP workloads

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

***Abstract*— The importance of database systems to efficiently support OLTP and OLAP mixed workloads, the so-called OLXP workloads, has been recognized recently. Some research projects and also some commercial database systems with focus on the processing of OLXP workloads have appeared in the last few years. In this paper, we present our work to extend the PostgreSQL OSS database system to efficiently handle OLXP workloads. Besides PostgreSQL's traditional OLTP-oriented row data store, we provide a new OLAP-oriented column data store in the form of a new index. Unlike previous works that support both row and column data stores, we propose a column index with no restrictions concerning data size and/or updatability. Therefore, transactional data inserted to PostgreSQL row data store become immediately available for efficient analytical processing using the proposed column store index.**

Keywords— PostgreSQL; OLTP; OLAP; OLXP; row data; column data.

I. INTRODUCTION

Historically, database systems were mainly used for online transaction processing (OLTP) where transactions access and process only a 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 a 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 the approach taken 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 addressed in some research projects and commercial products recently. Some approaches adopt the row data store for both OLTP and OLAP [4] but it is still to be proven that row data stores can efficiently support mission-critical OLTP workloads. Other hybrid approaches allow a data table to be represented simultaneously in both formats: row data for OLTP and column data for OLAP but with some constraints in the column data size and its updatability/ synchronization with row data [5],[6].

Here we present a new approach to enhance PostgreSQL [7] so that it can effectively handle OLXP workloads without any constraints on data size nor updatability. In our approach, column data are created as indexes of PostgreSQL's data tables, and are synchronized instantly with row data for any insertions/updates/deletions in the data tables. There are other works that extend PostgreSQL to support column 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 PostgreSQL's data store and query processing engine are extended to support the new column store index. Section III shows some performance evaluation results. Finally, we present some concluding remarks in section IV.

II. COLUMN DATA STORE

In the following we present the mechanisms we have proposed to extend PostgreSQL in order to efficiently support column data.

A. Creation of Column Store Index

Using indexes is a common way to enhance database performance. An index allows the execution of some queries much faster than they could do without an index. PostgreSQL provides several index types: B-tree, Hash, GiST and GIN. Each index type uses a different algorithm that is best suited to different types of queries. In our approach, a column data store can be generated as a new type of index to support OLXP in PostgreSQL. Such an index is called Column Store Index (CSI), and can be created by the following CREATE INDEX command:

```
CREATE INDEX indexname ON tablename USING
csi (a, b, c);
```

With this CREATE INDEX command, a CSI named *indexname* is created for the data table named *tablename*. The data corresponding to the columns *a*, *b*, and *c* from the row data of *tablename* are written into different files, one for each column, and compose CSI.

PostgreSQL provides a set of index construction and maintenance functions. When an INSERT/UPDATE command is executed to insert/update a row data into the data table, a callback function to insert the new data into each existing index is called. CSI catches such callback function (aminsert) so that those insertions/updates can be reflected in CSI. However, in PostgreSQL, the existing indexes are not notified for a

978-1-4673-7551-1/15/$31.00© 2015 IEEE

DELETE command. In order to immediately reflect such a deletion in CSI, we extend PostgreSQL so that a callback function (amdelete) is called when a DELETE command is issued.

B. Column Store Structure

Updates on PostgreSQL's row data store have to be immediately reflected to the column data store. However it should be done without degrading the performance of OLTP transactions that run on the row data store. In the following we describe how it is achieved using CSI.

Figure 1 shows an overview of PostgreSQL's data store with its main components enhanced by the proposed CSI. As illustrated in the figure, the original data table has its data in row format, and CSI has the data in columnar format.

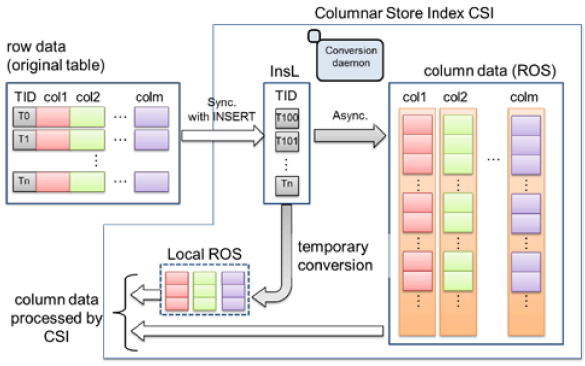

Fig. 1. Column Store Configuration

In the following we describe the three main CSI components, namely, the Insert/Delete Lists, ROS and Local ROS.

1) Insert and Delete Lists: The insertion of data into CSI is done using PostgreSQL's Custom Index framework and is executed immediately after a row is inserted into the original table. Since CSI maintains one file for each column data, when new data is inserted/updated/deleted, a write operation is necessary for each of those files and this would degrade the OLTP performance. In order to alleviate the insertion/delete cost, Insert/Delete Lists are introduced. Some previous works adopt the Write-Optimized-Storage (WOS) /Read-Optimized-Storage (ROS) approach [10] when implementing a column data store. In those approaches the updates are first buffered at WOS (where data are stored in row format) and then asynchronously transferred to ROS (where data are stored in columnar format). Although our approarch works similarly to WOS, different to the previous approaches that buffer the row data, in CSI only the original PostgreSQL table's row unique identifier (Tuple-ID, TID) are stored in an Insert List (InsL) and a Delete List (DelL).

As shown in Figure 2, the data transfer from InsL/DelL to ROS is executed periodically and asynchronously to the user query execution by a conversion daemon. When the OLTP load is too high, the converter is scheduled to be delayed in order to avoid OLTP performance degradation. The conversion daemon refers the data in the original table using

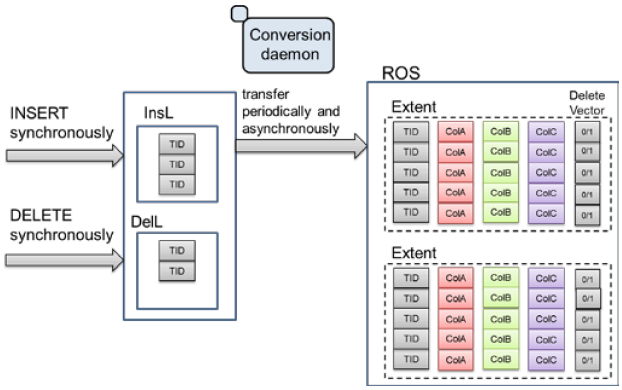

Fig. 2. InsL/DelL and ROS

the TID stored in InstL/DelL, and transfers each column data to ROS.

2) ROS: In order to increase the efficiency of CSI data management, data in ROS are grouped in units called *extents*. By default, 262,144 rows are packed into an extent, although the size of an extent is configurable. Data conversion, files writing, and data exchange with the query execution engine are all done in extents units. Data in the original table is transferred to ROS by refering to TIDs in InsL, when the number of TIDs in InsL is enough to fill one extent. Data in ROS are not updated with data overwriting. When transferring data to ROS, data whose TIDs are only in InsL are inserted into ROS, data whose TIDs are in both InsL and DelL are ignored, and data whose TIDs are only in DelL are inserted into a DeleteVector prepared in each extent.

*3) LOCAL ROS: S*ince information from InsL/DelL is transferred to ROS by the conversion daemon asynchronously, in a given time, all the data to be processed are not stored in ROS, but there are some data remained in the lists. When data are scanned by a query, those data remained in the lists are temporarily converted into column format called Local ROS. More details on how to construct and use Local ROS will be described in the next sections.

C. Concurrenry Control of the Column Store

Multi-Version Concurrency Control (MVCC) is the method PostgreSQL uses to handle data consistency when multiple queries are accessing the same table. With MVCC, each user connected to the database sees a snapshot of the database at a particular instant in time. When there is an update to an item of data, instead of overwriting the old data with new data, it will mark the old data as obsolete and add a newer version. PostgreSQL's MVCC uses increasing transaction IDs (XIDs) to achieve transactional consistency. Read transactions use XIDs to determine what state of the database to read, and read the appropriate version of the data. Specifically, each row in the data table contains the XID of the transaction that generated and deleted it, namely Xmin and Xmax, respectively. By comparing the XID of the read transaction with Xmin and Xmax, it is possible to determine if that row data should or should not be visible to that read transaction.

PostgreSQL's transactional consistency has to be preserved when using CSI. If a transaction is reading data from CSI at the same time that another transaction is writing to the data table, the read transaction should never see a half-written nor inconsistent piece of data. As we have explained, the TID of the inserted data is first stored in InsL, and then asynchronously and periodically transferred to ROS by the conversion daemon. In order to preserve transactional consistency in CSI, we introduce an MVCC that has to guarantee the following two conditions:

(1) Any INSERT and DELETE done by a transaction that needs the database to be rolled-back should be discarded so that the database state is the same as it was before those changes were made.

(2) When data are read from ROS while the ROS conversion daemon is running, the same data should not be read twice (from InsL/DelL and ROS).

In order to guarantee (1), InsL and DelL register the same Xmin and Xmax as the row data in the original table so that the same visibility check that is applied in the original table can be used (Figure 3). When transferring data to ROS, only the TIDs in InsL of rows whose generating transaction has already committed, and that are visible from all the existing transactions are used. Analogously, only the TIDs in DelL of rows whose deletions can be visible from all the existing transactions are marked in DeleteVector. Therefore, only data that are not affected by database rollbacks are transferred to ROS, and this approach makes (1) easily achievable.

In order to guarantee (2), each scan query should see the snapshot of the data in both InsL/DelL and ROS at the time the query initiates. InsL/DelL snapshot is generated as the Local ROS composed of data that are determined as visible at the time the query starts. Analogously, ROS snapshot is generated by checking the visibility of data in ROS.

In case each ROS data had an Xmin and an Xmax for its visibility check, the processing overhead would be too high. Since only data that were visible from all the existing transactions were used at each ROS extent creation, the check visibility of a query can be executed at ROS extent units.

In CSI, ROS extents contain information of the transaction that generates and deletes those extents, namely Xgen and Xdel, respectively. As we have described before, a ROS extent is never overwritten. It is generated by converting data from InsL/DelL to ROS, or transferring surviving data from an extent to a new extent, or by a combination of both transfers. In the ROS extent visibility check, the XID of the last ROS transferring transaction that run before the user read query is used. This XID is compared to Xgen and Xdel so that only in case Xgen ≤ XID < Xdel is true, the extent is visible.

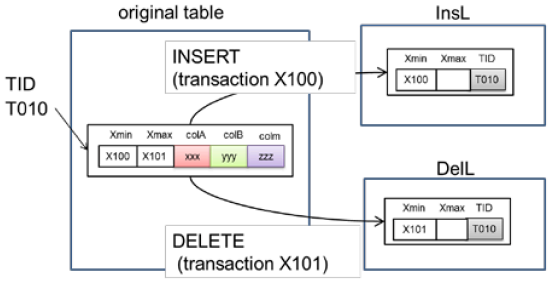

Fig. 3. InsL and DelL Visibility Check Information

Let's illustrate how MVCC is processed in CSI using the example in Figure 4. At a given time, InsL contains TIDs of 3 rows stored in PostgreSQL tables, namely T100, T101 and T102. At the same time, ROS contains 5 extents that were generated by ROS conversion transactions whose XID were X100, X120, X130, X140 and X150 (those XIDs are registered as the extents' Xgen). All Xdel contain the initial value (=∞). Now let's suppose that there is a user transaction with XID=X190 containing two read queries that will use CSI. To check which ROS extents are visible, query #1 is assigned XID=X150 since it is the XID of the last succeeded ROS conversion transaction. For all the 5 ROS extents we find that Xgen≤X150< ∞ and thus, they are all visible to query #1. The visibility check is also done with InsL data using their Xmin and Xmax, and let's suppose that in this example, the columns corresponding to the 3 TIDs in InsL are visible and thus, used to generate the Local ROS for query #1.

Now let's suppose that a ROS conversion whose XID=X200 runs before query #2, and that the columns corresponding to T100 and T101 satisfied the 2 conditions to be transferred to ROS, that is, they were generated by transactions that have already committed, and are visible to all currently running transactions. Let's also suppose that the number of deleted data in extent 1 became higher than a pre-defined threshold. Therefore, the transaction with XID=X200 generates a new ROS extent by copying data that is not obsolete from extent 1, and transferring the columns corresponding to T100 and T101 from InsL to the new extent 5. On completion of the transfer, X200 is registered as Xgen of the new extent 5, and as Xdel of the old extent 1. Note that extent 5 is invisible from query #1, even after the transaction with XID=X200 finishes, because X150 is older than Xgen (=X200).

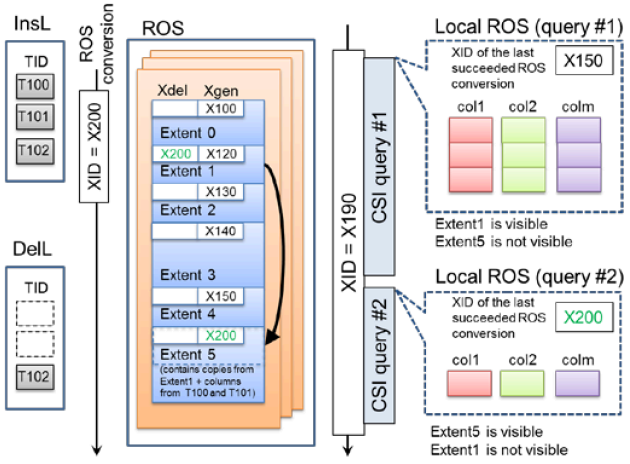

Fig. 4. MVCC in CSI

Since the last succeeded ROS conversion transaction that run before query #2 was X200, it is used as the XID of query #2. From the check visibility of ROS extents, those that satisfy Xgen≤X200< ∞ are extents 0, 2, 3, 4 and 5. Note that for extent 1, X120≤X200<X200 is false (since the necessary data was already copied to extent 5 by the ROS conversion transaction) and thus extent 1 is not visible. Regarding InsL, as shown in the Fig. 4, the visibility check ensures that only the column data corresponding to T102 is used to generate the Local ROS for query #2.

In this way, by combining the visibility check of InsL/DelL and ROS, the same transaction consistency as the original PostgreSQL is guaranteed in CSI.

D. Query Execution

1) Plan Rewritting: SQL queries that are submitted to PostgreSQL enhanced with CSI go through the usual PostgreSQL query compilation, including optimization that produces a plan tree of operators. In the case where the cost evaluation shows that the query is faster when accessing the CSI, the plan is rewritten. Since we are implemeting our extensions based on PostgreSQL 9.4, for the plan rewritting, we backported and utilized the Custom Scan API that will be introduced in PostgreSQL 9.5 [11].

The nodes that are target for plan rewriting are the SeqScan node, the Aggregation node and the Sort node. As shown in Figure 5, when the plan tree of the best plan generated by PostgreSQL's planner contains such nodes and it is identified that column data processing would result in lower cost, the plan tree is rewritten by replacing the original nodes with the CSI Scan node, CSI Aggregation node and CSI Sort node.

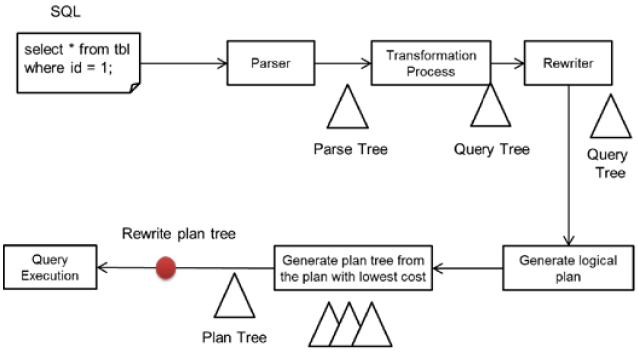

Fig. 5. Plan Rewriting

2) Plan Execution: The rewritten plan is executed using the set combining ROS and LocalROS visible data, as shown in Figure 6. As described before, Local ROS is generated by referring to data whose TID are in InsL/DelL and are converted to the same columnar format as ROS. Data read from ROS is filtered by the deleted data registered in the DeleteVector and DelL.

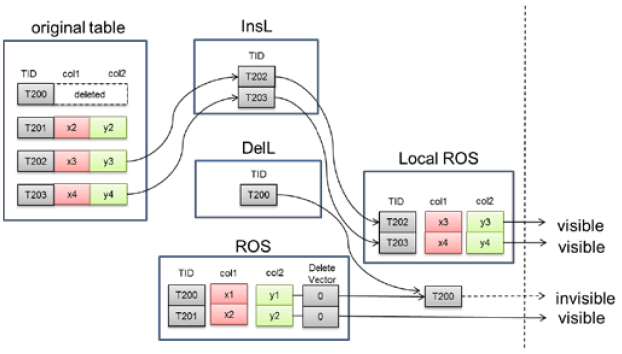

Fig. 6. Column Data to Process

2) Vector Processing: PostgreSQL plans contain expression trees to process conditional statements such as those found in WHERE clauses. When executing the plan, the expression tree is scanned and the operations are executed. For instance, the expression tree illustrated in Figure 7 is generated by PostgreSQL for the expression (A* 3) + B. However, the scan of such an expression tree generates a large amount of branch mispredictions that decrease performance. In order to reduce the incurred overhead, data are read and processed in units of *vectors* (e.g. 128 rows) and, as shown in Fig. 8, each node of the expression tree, instead of processing only one row-at-a-time, processes a vector-at-a-time.

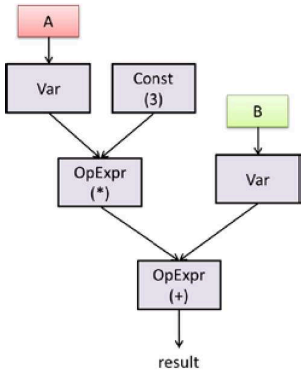

Fig. 7. PostgreSQL's Expression Tree for the expression (A* 3) + B

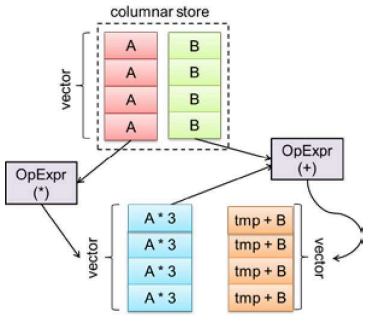

Fig. 8. Vector Processing for the expression (A* 3) + B

E. Parallel Processing

Sharing of data between the parallel workers is needed for parallel execution. In PostgreSQL 9.4, a shared memory mechanism named Dynamic Shared Memory (DSM) was introduced with Dynamic Background Worker (DBW) [12]. However, although DSM is a mechanism that allows the sharing of memory, the mapped location is different for the processes and thus, data that contain pointers cannot be shared naively between the processes and a large amount of data has to be copied for inter-process data sharing.

We developed the Shared Memory Context (SMC) that is a mechanism to overcome DSM's drawback and replace it. The proposed SMC is a memory that is shared among the Backend Process and DBWs for parallel processing. Unlike DSM, the related processes can share the same memory address and thus the pointer mapping problem is solved and a large amount of data copy is avoided to improve performance.

Queries using CSI are executed in parallel, whenever possible. Each DBW processes a different extent as a parallel processing unit. The parallel DBWs use the algorithm selected by the plan (Figure 9). For example, in the case a hash-aggregation is selected by the planner, each parallel DBW applies the hash-aggregation algorithm to its corresponding extent, and writes their result in SMC. Those partial results are then merged by the Backend Process which generates the final aggregation result that is returned to the upper plan node.

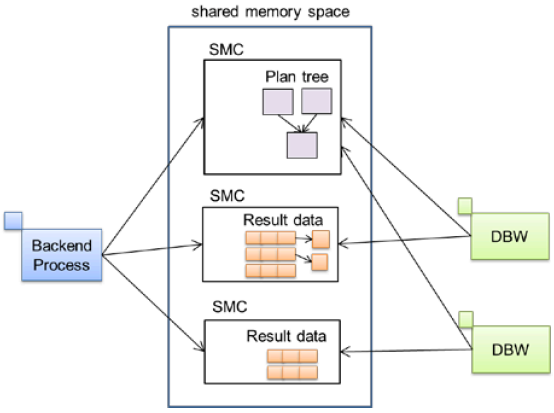

Fig. 9. Query Execution using SMC

Moreover, SMC is compatible with the interface of the Memory Context mechanism of PostgreSQL (e.g. palloc/pfree). Therefore, it becomes possible to call an existing PostgreSQL routine, and memory that is allocated at the routine can be easily shared using SMC. This way, structures such as plan trees and hash tables that are provided by PostgreSQL can be shared as they are, without any modification. Details on the design and implementation of SMC will be found in another paper that is under consideration for publication.

III. Performance Evaluation

While the improvements in performance provided by CSI vary by workload and data characteristics, we have started some evaluation of CSI with promising results. Although we are at a preliminary stage of evaluation using the DBT-3 benchmark [13], the performance of CSI processing compared on the same server (a 2-CPU machine with 16 cores) against PostgreSQL's traditional row data processing showed speedups that vary in orders of magnitude. For instance, for query 1 of DBT-3, a speed-up ratio of 50 was achieved. A detailed report on these results is planned for a future paper.

IV. Conclusion and Future Work

In this paper, we have presented our work in extending the PostgreSQL OSS database system with a column store index to handle OLXP workloads. We have introduced new mechanisms to efficiently reflect the OLTP inserts/updates/deletes on row data using a column store index that preserves transactional consistency and has no limitations in size. Using the proposed column store indexes, OLAP queries can be efficiently parallelized based on a new shared memory framework that is fully compatible with PostgreSQL's memory context interface.

We have plans to evaluate the effectiveness of CSI by continuing our performance analysis using the DBT-3 benchmark, and also extend it using the CH-benCHmark [14] and some real applications.

References

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

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

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

[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.

[5] "Oracle Database In-Memory", Oracle, Available from: http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html.

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

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

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

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

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

[11] K. Kaigai, "Custom Plan Patch", Available form: http://www.postgresql.org/message-id/9A28C8860F777E439AA12E8AEA7694F8F8F6BA@BPXM15GP.gisp.nec.co.jp.

[12] Background Worker Processes, Chapter 45, Available from: http://www.postgresql.org/docs/9.4/static/bgworker.html.

[13] Database Test Suite, Available from: http://sourceforge.net/projects/osdldbt/files/dbt3.

[14] R. Cole. et al., "The mixed workload CH-bencCHmark, Proc.DBTest, 2011.