PantherBRACE - CS2550 Spring 2018 Term Project

Design Document

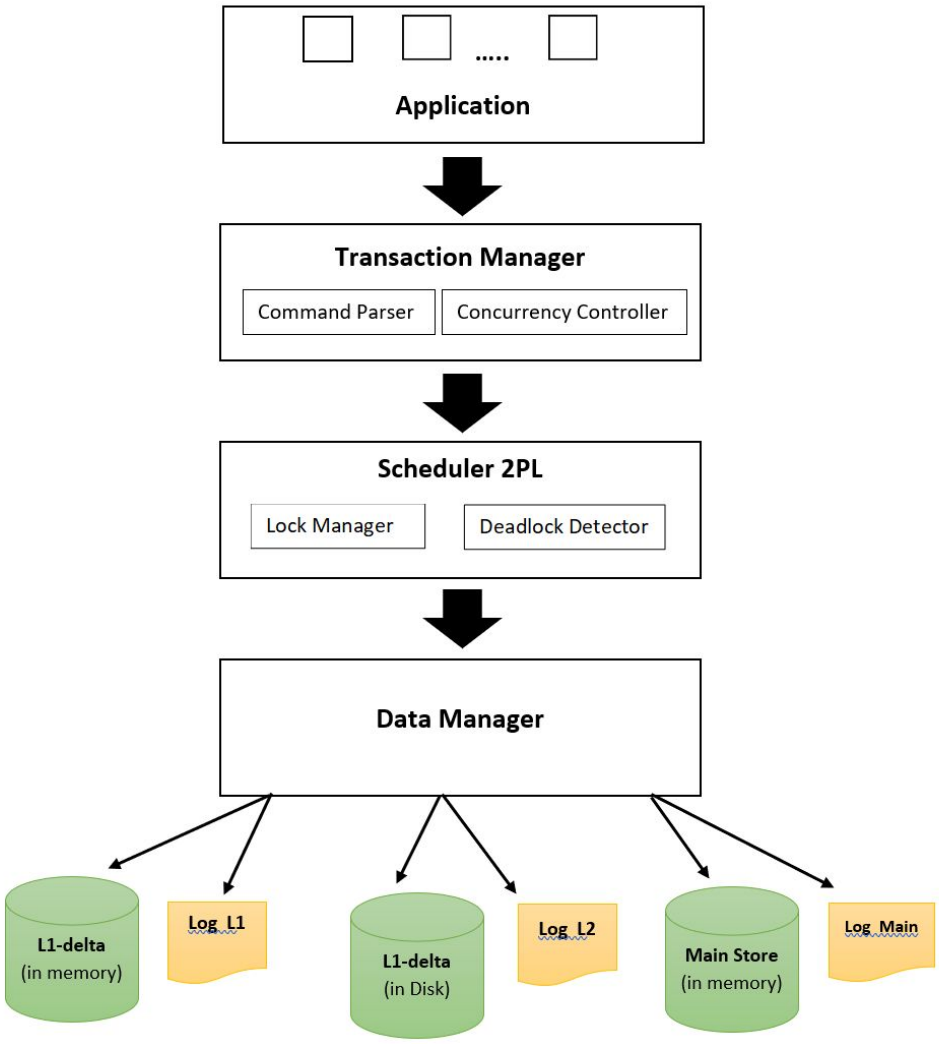
Ajesh Koyatan Chathoth (ajk158)
Mehrnoosh Raoufi (mer140)

Overview

PantherBRACE (Panther Blockchain tRAnsactions and Concurrency Engine) is going to provide limited transactional support (does not support durability) for a small analytics database system. It basically implements SAP HANA database to support both OLTP and OLAP. Mainly, it consists of three module: **Transaction Manager**, **Scheduler** and **Data Manager**. We describe each of them in the rest of this document.

Database Design Architecture

PantherBRACE consists of several layers. The architecture as well as the program flow is depicted in the diagram below.



Modules

1. Transaction Manager

TM is responsible for reading commands from scripts run by different application programs. It consists of a script parser and a concurrency control module.

1.1. Command (Script) Parser

This module reads the script run by various applications accessing the database concurrently. The parser identify the commands based on format given in below table.

1.1.1. Script format

Following script commands need to be supported by the parser.

Command	Description
B Emode	Begin (Start) a new transaction (EMode=1 – update) or new query (EMode=0 – read).
C	Commit (End) current transaction (query).
A	Abort (End) current transaction (query).
R table val	Retrieve the tuples with ID=val in table. If the record does not exist, it returns -1. table: Table a single letter such as X or Y.
M table val	Retrieve the record(s) which have val as area code in Phone in table. If the record does not exist, it returns -1. table: Table a single letter such as X or Y.
W table (t)	Write/Insert the tuple t into table. table: Table a single letter such as X or Y.
D table	Delete all tuples in the table. table: Table a single letter such as X or Y.
G table val	Counts the number of customers in area-code=val in table. table: Table a single letter such as X or Y.

1.2. Concurrency Control Module

This module implements the method to concurrent reading from scripts and pass the commands to the Scheduler. This module has to use multiple threads to achieve concurrency.

Method	Description
Weighted Round Robin	reads proportional number of lines for each line from the smallest file at a time in turns.

Random	reads from files in random order, and reads a random number of lines from each file.

1.3. Methods

Transaction Method	Implements
Read	R table val, M table val
Write /Insert	R table val
Delete	D table
Count	G table val

2. Scheduler based on 2PL

2.1. Lock manager

Since this database system supports concurrency it needs to provide a mechanism to support atomicity for operations as well. To this end, lock manager should maintain a lock compatibility table which exposes the conflict between operations. Then, based on this compatibility table can decide whether to grant a new lock request. Lock granularity would differ based on the storage type. For L1 that is stored row-wise, locking item is a single row while for L2 and Main Store locking item is a whole column. The lock compatibility table is shown in the table below.

	read	Bulk read	write	count	delete
read	-	-	x	-	x
Bulk read	-	-	x	-	x
Write	x	x	x	x	x
count	-	-	x	-	x
delete	x	x	x	x	x

2.2. Lock Table

For each type of storage there will be a corresponding lock table. Each lock table consists of locking items, currently granted locks for that item, and a queue of requester for lock on that item. It will generally looks like the table below.

Locking item	Granted Locks	Lock Request Queue
X (can be a row or column)	<T1, wl>	<T2, wl>, <T3, rl>
Y (can be a row or column)	<T4, rl>, <T5, rl>	<T6, wl>, <T7, rl>

It should be noted that for each application there would exist an instance of each three locking table since tables of different application do not have conflict and are independent.

2.3. DeadLock Detection Module

For deadlock detection a wait-for graph is formed from all uncommitted transactions and each time a transactions arrives at the system it is inserted as a new node to this graph. Afterward, a cycle detection algorithm is executed on the graph. If it detects existence of a cycle, deadlock is detected. Then a mechanism will be applied to avoid this deadlock. To do this, the coming transaction would not be granted for execution. Whenever a transaction commits, its corresponding node will be omitted from the graph.

3. Data Manager

Data manager is responsible for the maintenance of data in the tables. Singleton design pattern must be used to implement DM to make sure only one instance can be created by the system.

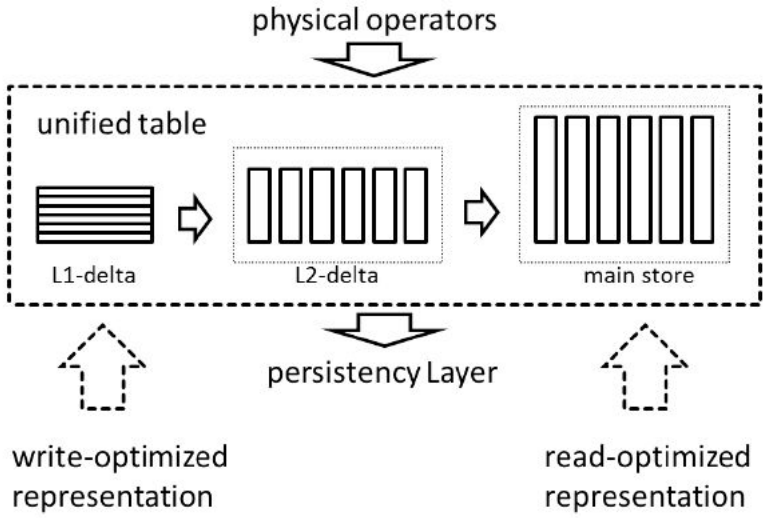
3.1. Table

Table consists of tuples (records) with the following fields:

Fields	Data type
ID (Primary Key).	Integer (2 bytes)
ClientName	18-character long string (18 bytes)
Phone	12-character long string (12 bytes)

3.2. Store management

In order to improve the efficiency on transaction processing, we use the technique of unified table structure and ensures excellent performance for both scan-based aggregation queries but also for highly selective point queries. There are three stores used for this purpose.



3.2.1. **L1-delta store**

It resides in the main memory so it is volatile and is relatively small. It is supposed to accept all kind of request; small insert, frequent updates so it should be optimized for write. To this end, we store it *row-wise*.

Data Structure: It would be implemented as a linked-list of a Struct that consists of attributes (ID, ClientName, Phone). It is row-wise in a way that each item of the linked-list representing a row.

3.2.2. **L2-delta store**

It resides in the hard disk. It is persistent and quite large. To be persistent, it needs to be stored as a file in the hard disk. This storage is meant to be optimized for bulk insertion so that it would be stored column-wise.

Data Structure: There are three attributes for each database table (ID, ClientName, Phone) so basically, L2 will be three distinct linked-list, one for each attributes. Each element of the each linked-list consists of corresponding types of attribute in addition to an index.

3.2.3. **Main Store**

It resides in main memory so it is volatile and relatively smaller than L2-delta stroe. It is meant to be optimized for read operation. Thus, it is stored column-wise similar to L2-delta but it is not maintained in a file.

Data Structure: Similar to the L2 it will be three distinct linked-list for each attributes (ID, ClientName, Phone). These lists are going to be kept in a sorted manner so that it would be efficient for read operation.

3.3. Methods

3.3.1. Initialize/Configure

Each application will run under a limited designated buffer space. The size of the buffer is specified at the beginning of each execution . Thus, this method is responsible to allocate the specified buffer size to each application before the execution.

3.3.2. L1-2-L2 delta Merge

Different data structures share a set of common data types and the access is exposed through a common abstract interface with row and column iterator. Record life cycle is organized in a way to asynchronously propagate individual records through the system without interfering the current running database operations.

Procedure: This involve transformation from row to column format. That is by appending new entries to the dictionary (in parallel) and storing column values using the dictionary encodings (in parallel) and finally removing propagated entries from the L1-delta.

Merge type: Incremental merge

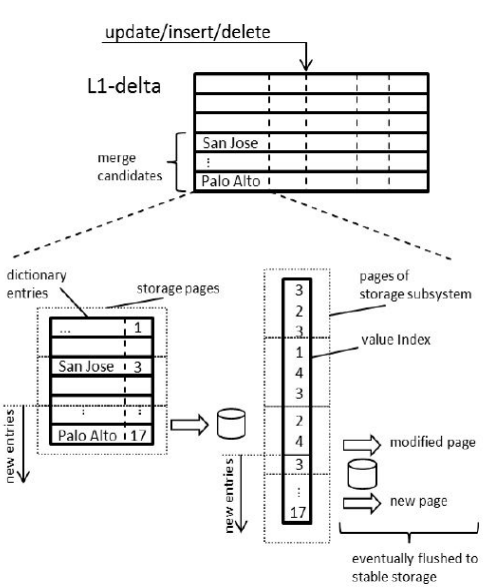
Merge frequency: Based on data size

3.3.3. L2-delta-2-Main Merge

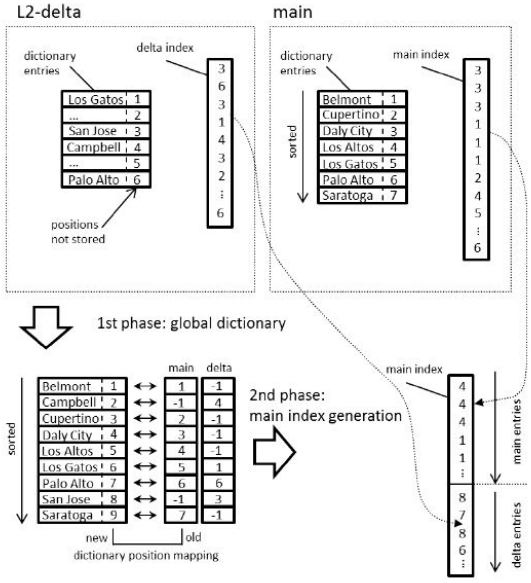
This is a resource intensive task. The old L2-delta is closed for updates and a new empty L2-delta is created. This is followed by creation of main structure by merging and if in case of any failure, the merge operation is retried.

Merge type: partial/ full merge

Merge frequency: Based on data size



L1-delta to L2-delta merge



L2-delta to Main store merge

During merge operation, lock management is handled internally by locking the read/write operation in stores affected.

3.4. Log management

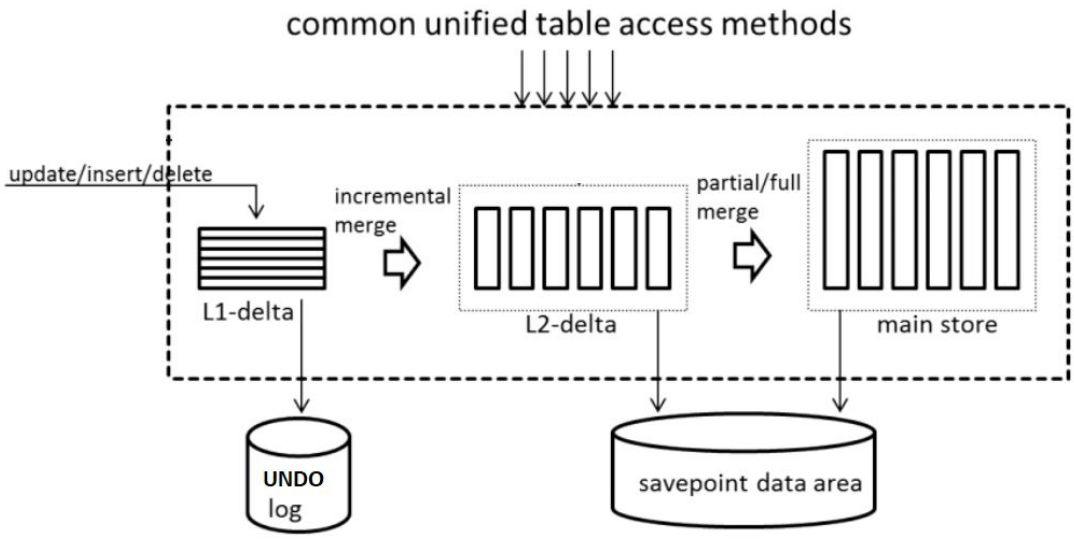
The DM maintains one log per store (L1-Delta, L2-Delta and Main). The log for L1-Delta and Main are maintained in the Main memory while for L2-delta, we store the log file in the hard drive.

Implementation Method: “live” blockchains

In this approach, the block hashes will be used for fast identification of the propagated commits amongst the stores. Once the changes from given block get populated all the way to Main store, the part of the chains (one chain per store) up to this block can be truncated. It is to be noted, however, that the notion of “live” blockchains allows for them to be maintained in memory.

Data Structure: List
Token format: <timestamp, operation, table, field, value>
Attributes: <block_number, nonce, list_of_tokens, hash_of_previous_block, hash> for each block.

Recovery Strategy: It ensures transaction atomicity by adopting an UNDO recovery strategy where all BEFORE-images are kept in the buffer and they are written to the file on the disk at runtime. Any abort due to deadlock are ignored.



Evaluation

We will maintain a mirror row store of Delta-1 which stores the tuples in Directed files, one per table. Within a file, records are stored in slotted pages whose size is 512 bytes.

Development Language

We will use Java for implementing every modules in windows/linux environment.