LSM TREE INDEX

CS 631 Project

Paras Garg
203050004
Praneeth Reddy
203050019

15 December 2020

Github Link : `https://git.cse.iitb.ac.in/parasgarg/lsm`

Introduction

Every minute 600,000 visits on amazon, Facebook server, and more than 100,000 writes and read operations are sent to the database. With such a high amount of data being stored, viewed and analysed, a demand for high performance comes as a must. Hence, the need of the hour is that the data should be stored and retrieved quite efficiently without the performance being compromised.

Objective

In this project, our aim is to modify the index access layer of PostgreSQL to add a LSM Tree Based Index over existing Btree based index access "src/backend/access/nbtree" so that our LSM Tree Based index optimize write performance over existing btree based index implementation.

Key idea

1. Creating a new extension in the existing postgres database.
2. By creating two levels of index called **L0** and **L1**. Initial inserts are made into **L0**, when **L0** exceeds it's size, flush **L0** into **L1**.
3. **L0** and **L1** are modified Existing Btree indexes.
4. We implemented searching using custom index access method.

System Architecture

Front-end

It is just the existing psql interface that PostgreSQL provide.

Back-end

Implemented a structure similar to Log structured Merge trees to organize the incoming data on the basis of clustering by search key.

- Worked in single-user mode
- Not handled concurrency control issues.
- Implemented in C language
- Used visual studio code IDE for building code and GCC-GDB for debugging and PG_CONFIG for building and installing the extension.

Steps to Install the LSM Extension

- Set the following variables appropriately:
  - export PATH=/usr/local/pgsql/bin:$PATH
  - export PG_CONFIG=POSTGRES_INSTAlLDIR/bin/pg_config
- Move to the lsm(Extension folder) directory
  - cd lsm/*extension folder*/
- To use the PGXS infrastructure for your extension run the following commands
  - make USE_PGXS=1
  - make install USE_PGXS=1
- Start the Postgres Server(restart the server if it's already running)
- To create extension use
  - Create Extension lsm

Implementation Details

We have created a LSM TREE Based index called LSM. LSM index consists of two levels, **L0** and **L1** where **L0** is used to insert from the user and when **L0** exceeds it's size, flush **L0** into **L1**.

Our Main goals

1. Store metadata for the index. We found two ways to store it.
   (a) Create a datastructure called shared memory hash table in shared memory of postgres in extension _PG_init() and use hashsearch to retrieve and store our metadata.
   (b) Each Btree index called nbtree in postgres store its metadata in a separate page (PAGE 0) and store datastructire Btree Metadata. We use the same approach, appended our metadata to end of metadata page (page 0) of btree, which is represented in the form of LSM Metadata . To avoid metadata loss when page shrink occurs we make Page Structure lower address to point to the end of metadata.
2. Index access methods
   (a) Each index access method is described by a row in the pg_am system catalog. The pg_am entry specifies a name and a handler function for the index access method. These entries can be created and deleted using the CREATE ACCESS METHOD and DROP ACCESS METHOD SQL commands.

(b) CREATE ACCESS METHOD register a handler function which accepts a IndexAmRoutine which will contains pointer to all the functionality and address of function to our index.

(c) The IndexAmRoutine struct, also called the access method's API struct, includes fields specifying assorted fixed properties of the access method.

3. Creating index support class, operator support function and operator family.

   (a) The routines for an index method do not directly know anything about the data types that the index method will operate on. Instead, an operator class identifies the set of operations that the index method needs to use to work with a particular data type.

   (b) Since Btree support 5 operator strategies and support function 5 . We create LSM opertors and operator family using existing operator support functions used by nbtree implementation.

   (c) Operator classes are used to compare the operands of same data type.

   (d) To compare cross data type operators operator families are created.In operator family left operand is always taken as index operators.

4. Index Build
   Implemented in a function lsmbuild . lsmbuild create the L0 level index by calling internally btree build which will create btree index and add our metadata to Btree Metadata.

5. Insert

   (a) Initial N (fixed) insert is made into **L0** when **L0** reaches its maximum size

   (b) If **L1** is not created it will be created and its OID is inserted into lo metadata.

   (c) **L1** is created internally using index_concurrently_create_copy method using existing Index Info from **L0** and same name as **L0** by prepending **L1** to its name such that both **L0** and **L1** are unique in the database.

6. Merging : Merging is done in 3 steps.

   (a) Creating indexscan on **L0** btree using index_beginscan method.Scanning over entire Btree **L0** using btreescan and inserting each node from scan to **L1** node. Instead of deleting tuple from **L0** we truncate **L0** to single page using RelationTruncate

   (b) Rebuilding **L0** tree using index_build and existing index_info structure from **L0** index

(c) Rewriting metadata to **L0** node since all page including Page 0 has been deleted except index_info metadata maintained by system catalog.

7. Functions created

   (a) Datum lsm_handler(PG_FUNCTION_ARGS)
   Create and return a am_handler

   (b) Static bool lsm_insert(Relation rel, Datum *values, bool *isnull,ItemPointer ht_ctid, Relation heapRel,IndexUniqueCheck checkUnique,IndexInfo *indexInfo)
   Handles insert into Lsm index. internally maintains **L0** and **L1** and insert data in **L0** initially.If required merge **L0** into **L1**.

   (c) void lsm_create_L1_if_not_exits(Relation heap,Relation index,LsmMetaData* lsmMetaCopy)
   Create l1 index if not exist in LSM index

   (d) lsmbuild(Relation heap, Relation index, IndexInfo *indexInfo)
   Used to build a lsm tree internally call btree function to create btree index. If heap relation already contains data than it will be inserted into **L0** . When next insert occurs if it exceeds size, then **L0** is flushed into **L1**.

   (e) void lsmbuildempty(Relation index)
   Used to create a empty btree.

   (f) static void lsm_merge_indexes(Oid dst_oid, Relation top_index, Oid heap_oid)
   Used to merge **L0** into **L1** created on given heap relation.

   (g) static void lsm_truncate_index(Relation index, Oid heap_oid)
   Used to truncate **L0** to zero buffer and rebuild **L0**.

   (h) static void lsm_init_entry(LsmMetaData* entry, Relation index)
   Used to initialize the lsmMetaData entry

Files created

1. lsm.h
   Contains function prototype and struct for LSMMetaData

2. lsm.control
   Parameter used by pgsx to install the extension

3. lsm.c
   contains function used by lsm to handle the index implementation

4. lsm–1.0.sql
   Contains handler definition, operator class and operator families used by lsm index.

5. Makefile
   Make file to build the shared library using the postgres source code.

6. lsm.o
   Object file generated for the compiled c code.

7. lsm.so
   Shared library generated for installation.

Files Modified

Files modified due to make install

1. POSTGRES_INSTDIR/share/extension/lsm.control
2. POSTGRES_INSTDIR/share/extension/lsm–1.0.sql
3. POSTGRES_INSTDIR/lib/lsm.so

Run Through

Two tree levels **L0** and **L1** with n=2

- create extension lsm; /*creates a new extension*/
- create table t(k bigint, val bigint); /* create a table t */
- create index lsm_index on t using lsm(k); /* create a lsm based index lsm_index on table t*/
- insert into t values (1,10);
  /* inserted into **L0**. Now **L1** contains 1 entry*/
- insert into t values (2,10);
  /* inserted into **L0** .Now,**L0** contains 2 enties*/
- insert into t values (3,10);
  /*inserted into **L0**. Now **L0** exceeds max size, since **L1** is not created **L1** is created, **L0** is flushed into **L1**, **L0** become empty **L1** contains 3 entries*/
- insert into t values (2,10); /* insert into **L0** . now lo size become 0 and **L1** contains 3 tuples*/

Further work that can be implemented

1. Implementation of locking strategies to make it multithreaded using RowExclusiveLock
2. Handling unique insertion by scanning both **L0** and **L1** to check whether given entry
3. Deletion of tuple from index by maintaining extra column in index tuple to maintain whether tuple has been deleted.When scanning returning those tuple only whose entry is not marked deleted.

Learning

1. Page layout mechanism of database.
2. How database manage multiple index access method in the catalog, by keeping entries in pg_am then defining generic index methods in index.c which will internally call index related method using pg_am handler registered.
3. yao and lehman btree code.
4. Vacuum processing.
5. How postgres scan the index using ambeginscan and amgetbitmap handler in am entry.
6. How to access locking mechanism.
7. Buffer management by database ,pinning and unpinning of buffer using rel_buf
8. Metadata storage.

Conclusion

Postgres provide a optimized version of btree called nbtree based on lehman paper. But btree may suffer when insertion in index are happen in random order and lot of buffer scan in random order is required. To optimize write we use a LSM based index using existing btree.Our LSM Based index can optimize further to reduce searching time because now we have to search in multiple trees simultaneous.Postgres provide a good knowledge how database works and manage all its data.

References

1. Postgresql 13 Documentation: Database page Layout
   https://www.postgresql.org/docs/current/storage-page-layout.html
2. Postgresql 13 Documentation :c-language functions
   https://www.postgresql.org/docs/current/xfunc-c.html
3. Postgresql 13 Documentation : Packaging Related Objects into an Extension
   https://www.postgresql.org/docs/current/extend-extensions.html
4. Postgres13 documenation : Operator Optimization Information
   https://www.postgresql.org/docs/current/xoper-optimization.html
5. Postgres13 Documentation : Index Access Method Interface Definition
   https://www.postgresql.org/docs/13/indexam.html
6. Discovering the Computer Science Behind Postgres Indexes
   http://patshaughnessy.net/2014/11/11/discovering-the-computer-science-behind-postgres-