Technical Disclosure Commons

Defensive Publications Series

October 28, 2016

Bloom Filter Locking

Adam Blinkinsop

Follow this and additional works at: http://www.tdcommons.org/dpubs_series

Recommended Citation

Blinkinsop, Adam, "Bloom Filter Locking", Technical Disclosure Commons, (October 28, 2016)
http://www.tdcommons.org/dpubs_series/308

This work is licensed under a Creative Commons Attribution 4.0 License.

This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Bloom Filter Locking

ABSTRACT

Database implementations typically use either fine-grained locking or coarse-grained locking, but not both in the same system. This can lead to greater contention and less efficient operation, e.g., when an application generates queries that have a wide variation. This document describes Bloom filter based locking techniques that are a compromise between coarse-grained and fine-grained locks. The techniques can efficiently handle queries from applications that generate queries that have a wide variation.

KEYWORDS

- Database locks
- Concurrency control
- Bloom filter
- Adaptive locking

BACKGROUND

Concurrent access to resources, e.g., database tuples, is implemented by using concurrency control techniques such as locks. Locks may be implemented, e.g., as fine-grained locks that temporarily restrict access of a process (e.g., program thread) to individual (or a small number) of database tuples, e.g., when such database entries are being edited by another process, or as coarse-grained locks that restrict access to a large set of database tuples.

Fine-grained locks enable specifically locking a limited set of tuples, while enabling access to other tuples in the database. However, implementing such locks may be computationally expensive, e.g., when a large number of tuples are to be locked, since each lock may cover only an individual tuple or a small set of tuples.

Coarse-grained locks that lock a large number of tuples may be computationally less expensive, e.g., since one lock restricts access to the large number of tuples. However, coarse-grained locks can cause too much contention, e.g., if the system receives requests to access tuples that are locked by a coarse-grained lock.

DESCRIPTION

Bloom filters

A bloom filter is a space efficient probabilistic data structure that is used to test whether an element is a member of a set. This data structure is probabilistic because it can return a false positive, e.g., when a particular element isn't a member of the set, but the bloom filter indicates otherwise. However, a bloom filter does not return a false negative. For example, when a particular element is a member, the bloom filter always returns confirmation of the membership. To add to a bloom filter, an input key is hashed multiple times, and the result indices in the bloom filter are set (e.g., "True"). To check membership, the input key is hashed and the result is compared with the corresponding indices in the bloom filter. If any of the indices are false, the input key is confirmed as not a member. Bloom filters enable inexpensive checks of membership in a set for an input element.

Locking techniques

Database implementations typically use either fine-grained locking or coarse-grained locking, but not both in the same system. This can lead to greater contention and less efficient operation, e.g., when an application generates queries that have a wide variation. For example, an application can generate a large number of small queries (e.g., small changes in rapid succession) that affect a small number of tuples of a database and also generate large queries (e.g., that require hundreds of changes to the database). This document describes bloom filter based locking techniques that are a compromise between coarse-grained and fine-grained locks. The techniques can efficiently handle queries from applications that generate queries that have a wide variation.

Implementing bloom filter locks

A bloom filter locking relation is created, with a specified constant size m and number of hash functions k. To update optimistic concurrency control (OCC) locks for a mutation, an input lock (e.g., a fine-grained input lock) is treated as a key and is hashed k times, modulo m. If the resultant value indicates that the bloom filter is unchanged, e.g., when updating a backend (such as a database), it is determined that the input lock was not updated. In implementing bloom filter locks, integer values may be used and incremented in the bloom filter corresponding each index, e.g., instead of using bit flipping, e.g., between "True" and "False" values. For example, the integer values may be based on a timestamp.

Single object operations

To convert a fine-grained lock for touch, k bloom filter keys are obtained based on k hash functions (modulo m). Each key determines a different value in a set. In the event of an update to an individual object (e.g., a backend database object), at most the k lock tuples of the bloom filter (plus a tuple corresponding to empty set) need to be updated. To convert a fine-grained lock for check, it is fed to each of the k hash functions to get k keys. If any of these keys is unchanged in the bloom filter, the object is unchanged.

Multiple object operations

To determine the locks that are to be checked, a smallest common set of bloom filter locks are computed from the input locks. This process is carried out in two steps.

- **Step 1:** In a first step, all bloom filter locks are computed and inserted into a multiset.

- **Step 2:** In a second step, for each input lock, the bloom filter lock that has the most multiset entries is identified. This bloom filter lock is inserted into a check set.

In some implementations, the second step may result in a tie, e.g., two different bloom filter locks that each have the most multiset entries. In order to handle ties, in some implementations, a seed is chosen (e.g., randomly) before performing the second step. If a tie occurs, the tied lock values plus the seed modulo m is sorted. This ensures that the check set is not biased, e.g., towards one end of the bloom filter relation.

In a case of a small query, this technique acts similarly to a fine-grained lock, such that individual queries have a low likelihood of experiencing a collision. With large values of m and k equal to 1, the bloom filter lock is very similar to a fine-grained lock. With larger queries, or with smaller values of m, the bloom filter lock acts more like a coarse-grained lock. As the query grows, the bloom filter is closer to a coarse-grained lock.

Advantages

Bloom filter locks as described in this disclosure are advantageous in many applications. For example, advantages are provided for classes of systems that restrict the number of conditions that could exist on a final mutation, along with a large variation in transactions, e.g., include both transactions that impact a small number of objects and transactions that impact a large number of objects. By implementing bloom filter locks, a cap can be set on the number of conditions and advantages of fine-grained locking can still be achieved for a small-query case.

Bloom filter locks can be used for different types of databases. For example, a graph database that stores data about graph nodes and edges. A node-specific lock acts as a coarse-grained lock such that edges to the node are locked with the node-specific lock. An edge lock acts as a fine-grained lock specific to the edge. In some implementations, bloom filter locks are partitioned, e.g., for different classes of information in the database, and a relatively low condition limit is enforced.

Bloom filter locks as described herein can be used in any computational context that requires efficient concurrency control for wide variation query types. For example, the techniques can be used in cache management, e.g., to implement responses to cache staleness queries. In another example, the techniques can be applied in implementing software transactional memory (STM), where multiple threads access a common set of data.

CONCLUSION

Techniques of this disclosure can be utilized to manage concurrency in large, high traffic databases. By implementing bloom filter locks, a compromise between fine-grained locks and coarse-grained locks is achieved. For small queries, the bloom filter lock functions similar to a fine-grained lock, while for large queries, the bloom filter lock functions similar to a coarse-grained lock. The techniques enable handling queries that cause significant contention when coarse-grained locks are implemented but also access a large number of individual objects such that fine-grained locks are computationally expensive.