Faster and Durable Hash Indexes

- Amit Kapila | 2017.05.25

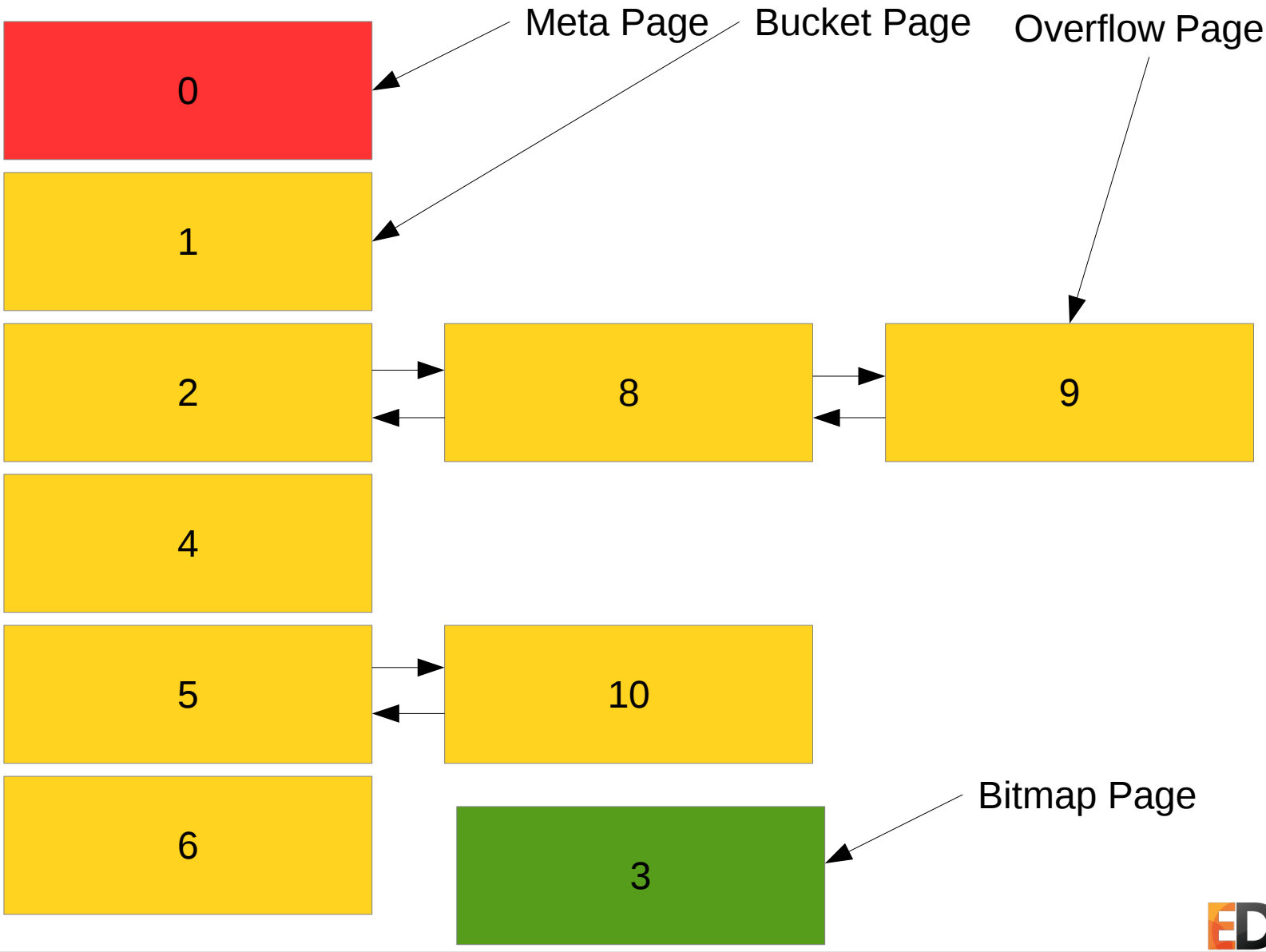
Contents

- Hash indexes
- Locking improvements
- Improved usage of space
- Code re-factoring to enable write-ahead-logging
- Write-ahead-logging
- Hash utility functions
- Further improvements

Hash indexes

- Hash indexes store 32-bit hash code for each indexed item.
- By definition, any operation in the hash index in O(1), however with duplicates that is not true.
- They are primarily used in equality search conditions (Select c1 From tab1 Where c2='xx';).
- Hash indexes are preferred to be used for unique columns.

Hash indexes



Contents

- Hash indexes
- Locking improvements
- Improved usage of space
- Code re-factoring to enable write-ahead-logging
- Write-ahead-logging
- Hash utility functions
- Further improvements

Locking improvements

- Changed the heavyweight locks (locks that are used for logical database objects to ensure the database ACID properties) to lightweight locks (locks to protect shared data structures) for scanning the bucket pages.
- Acquiring the heavyweight lock was costlier as compare to lightweight locks.
- Scans and Inserts can happen in parallel to split of a bucket.

Locking improvements

- Concurrency between Vacuum and Scans is also improved to some extent.

- Scans can precede Vacuum on the same bucket. This can be helpful in situations where bucket contains many overflow pages.

- The split operation can be interrupted and can be completed at a later time.

Locking improvements

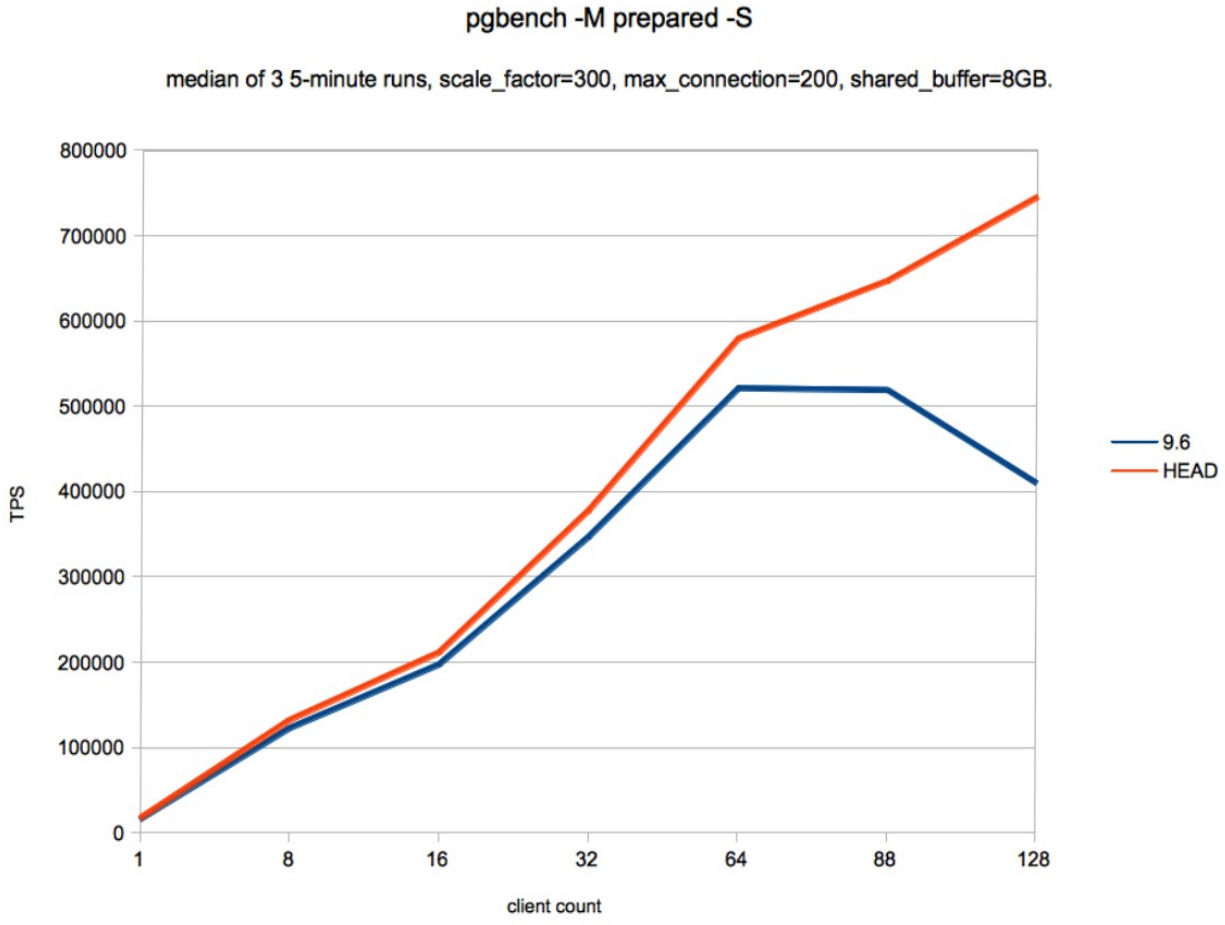
- Each search operation needs to access metapage to find the bucket that contains tuple being searched which leads to high contention around metapage.
- metapage is cached in backend local cache.
- Avoids a significant amount of buffer manager traffic and contention for accessing metapage.

Locking improvements (test setup)

- Performance comparison of hash index and btree.
- pgbench read-only workload where data fits in shared buffers.
- IBM POWER-8 having 24 cores, 192 hardware threads, 492GB RAM.

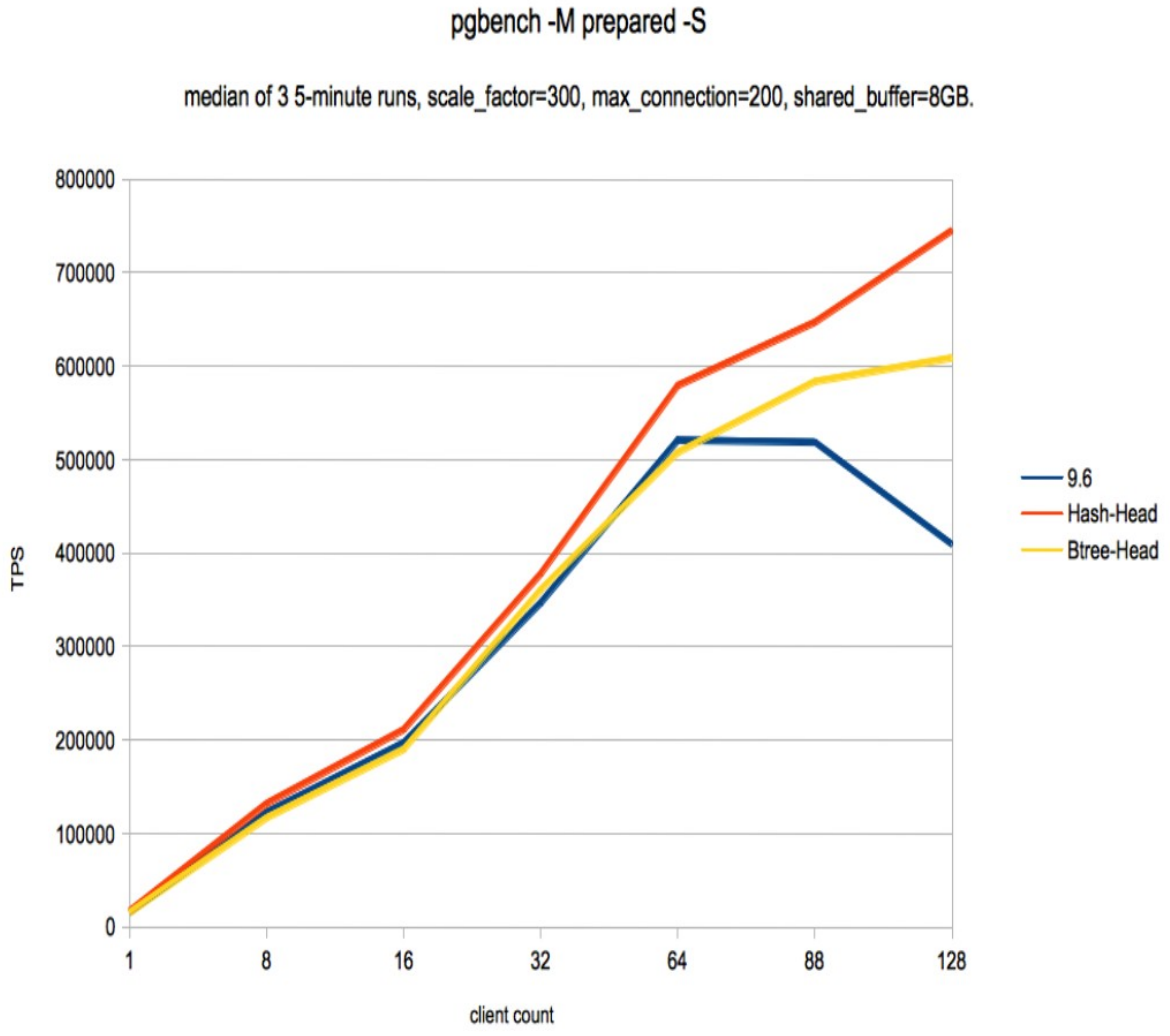
Locking improvements

- The performance of hash index has increased at all client counts in the range of 7% to 81%.
- The impact is more pronounced at higher client counts.



Locking improvements

- The hash index performs better than btree in the range of 10~22%.
- An improvement in the range of 40-60% for some other workloads has been observed when hash indexes are used for unique index columns.



Contents

- Hash indexes
- Locking improvements
- Improved usage of space
- Code re-factoring to enable write-ahead-logging
- Write-ahead-logging
- Hash utility functions
- Further improvements

Improved usage of space

- Prior to PostgreSQL-10, each split in hash index generally doubles the size of the hash index.
- To mitigate this problem, we now divide the larger splitpoints into four equal phases.
- Instead of growing from 4GB to 8GB all at once, a hash index will now grow from 4GB to 5GB to 6GB to 7GB to 8GB.

Improved usage of space

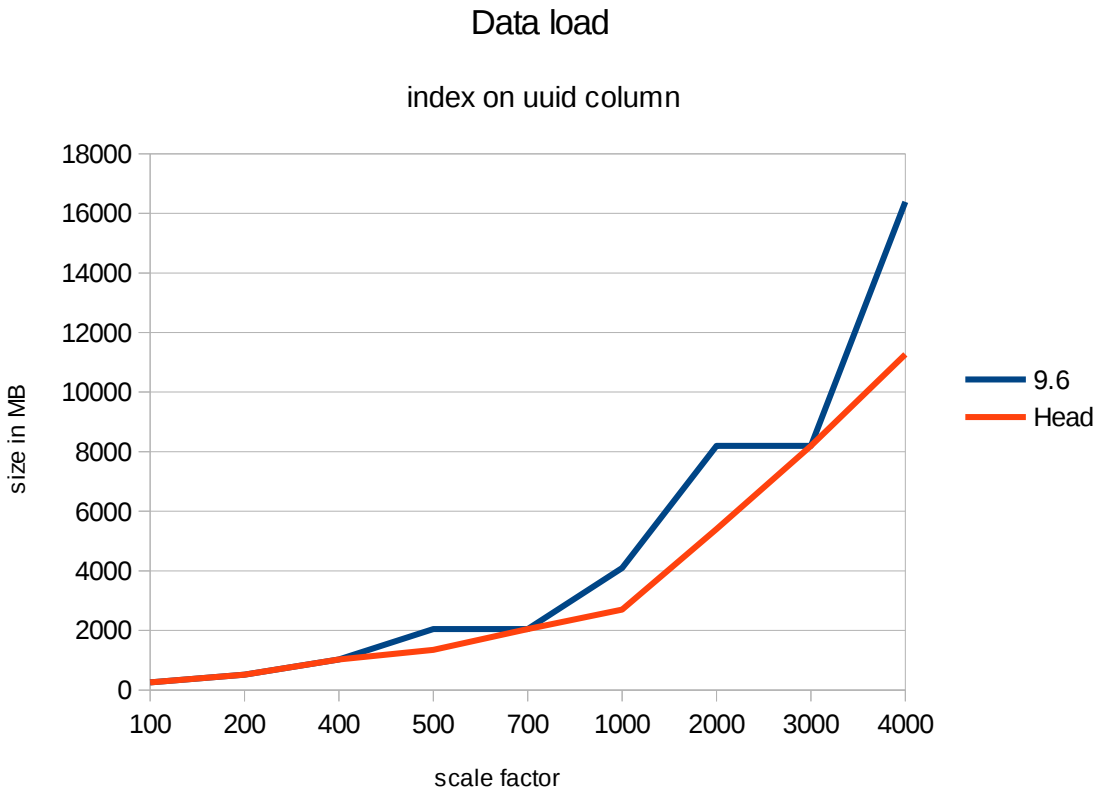
- Single page vacuum.
- Mark the items as dead page-at-a-time.
- If the space on the page is full, try to clean any dead items on the page and use the cleaned up space.
- This accelerates space recycling and reduces bloat.

Improved usage of space (test setup)

- Size comparison of hash and btree indexes on UUID column.
- Schema
  - CREATE TABLE hash_index_table (uuid_type UUID, bid int, abalance int, filler char(84));
  - CREATE INDEX hash_index on hash_index_table USING hash (uuid_type) with (fillfactor = 80);
- Scale factor = 10 means 1000000 tuples in tables

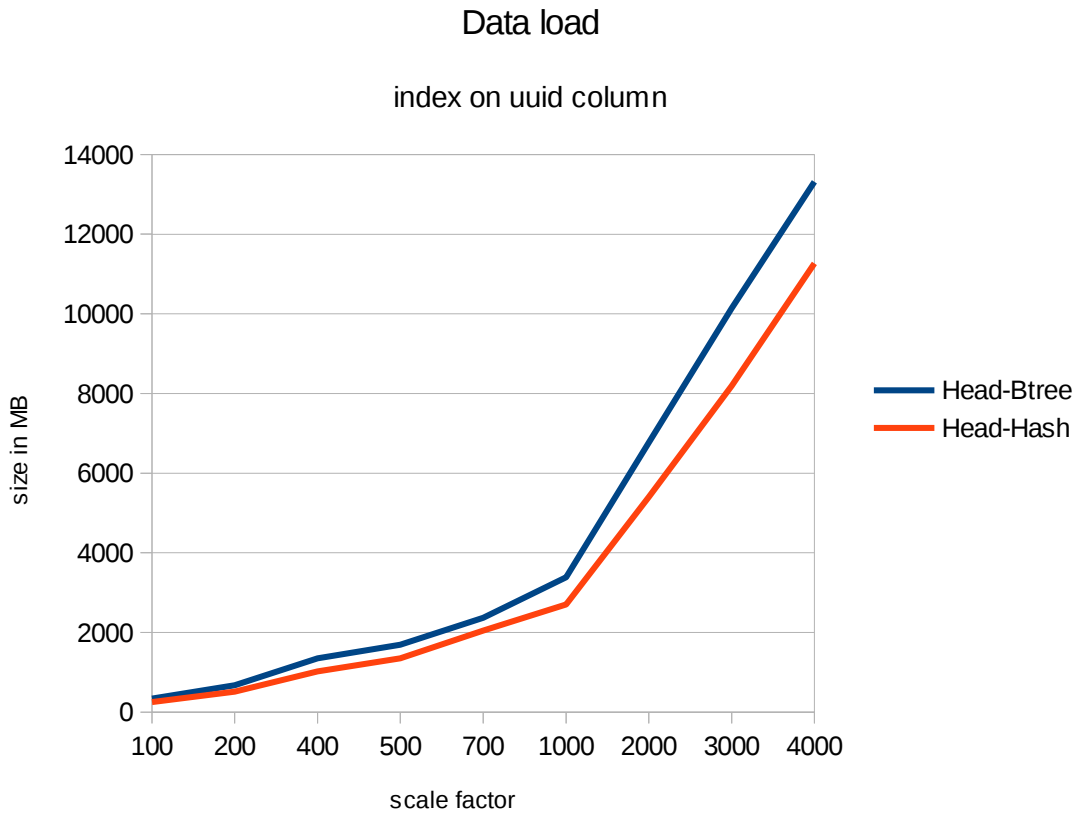
Improved usage of space

- The size of the hash index on Head is less than or equal to its size in 9.6.
- The size of the hash index in 9.6 is 45~50% larger than Head.



Improved usage of space

- The size of the hash index is less than the btree index on all scale factors.
- The size of the btree index is 15~32% larger than the hash index.



Contents

- Hash indexes
- Locking improvements
- Improved usage of space
- Code re-factoring to enable write-ahead-logging
- Write-ahead-logging
- Hash utility functions
- Further improvements

Code re-factoring to enable write-ahead-logging

- There are many places in the code of hash indexes where the operations that are atomic were not performed together.

- Reorganize the code to perform those operations together, so that they can be WAL-logged.

Code re-factoring (Squeeze bucket operation)

- We were moving (adding) tuples one-by-one from the bucket later in the chain to the bucket prior in the chain.
- This could easily lead to the same tuple returned twice if we would have logged it that way.
- Ensure that all the tuples from one bucket page are moved to another page in one-go.

Code re-factoring (Overflow page allocation operation)

- Allocation of new page and addition of the same in overflow chain are done separately.

- Adding separate WAL for each of those operations could waste newly allocated page if the system crashes after allocation of the new page but before adding it to overflow chain.

- Ensure allocation of new overflow page and addition of same in the overflow chain is performed as one atomic operation.

Code re-factoring (Split bucket operation)

- The start of split operation involves updating metapage and mark old/new bucket to indicate split operation is in progress.
- These operations were performed separately and logging them separately could lead to wrong query results after a crash or on standby.
- Ensure that these operations are performed atomically.

Code re-factoring (Create index operation)

- It involves the creation of a different type of pages in index and initialization of same.

- Re-organize code such that DO and REDO routines can share a common part of the code and all the related page modifications are done in a single section of code.

- This allows writing WAL record for each of the sub-operations. We don't try to write a single WAL record for all of the sub-operations as any error in create index operation will rollback the whole operation.

Contents

- Hash indexes
- Locking improvements
- Improved usage of space
- Code re-factoring to enable write-ahead-logging
- Write-ahead-logging
- Hash utility functions
- Further improvements

Write-ahead-logging

```
CREATE TABLE test_hash (a int, b text);

INSERT INTO test_hash VALUES (1, 'one');

CREATE INDEX test_hash_a_idx ON test_hash USING
hash (a);

WARNING:  hash indexes are not WAL-logged and their use
is discouraged
```

- WARNING is removed now, hurray!

Write-ahead-logging

- Hash indexes are crash-safe and usable on standby's which will enable its usage on production databases.
- “snapshot too old” is now supported for tables with hash indexes.
- The hash index operations like create index, insert, delete, bucket split, allocate overflow page, and squeeze in themselves don't guarantee hash index consistency after a crash.
- To provide robustness, we write WAL for each of these operations.

Write-ahead-logging

- CREATE INDEX writes multiple WAL records. First, we write a record to cover the initialization of the metapage, followed by one for each new bucket created, followed by one for the initial bitmap page.

- It's not important for index creation to appear atomic, because the index isn't yet visible to any other transaction, and the creating transaction will roll back in the event of a crash.

Write-ahead-logging

- An insertion that causes split constitutes of multiple atomic actions (insertion itself, allocation of a new bucket, overflow bucket pages, update meta information to indicate split is complete).

- If the system crashes in-between multiple atomic operations after recovery old and new buckets are marked with flags to indicate split is in progress.

- The split will be completed at next insert or split from the old bucket.

Write-ahead-logging

- A deletion operation constitutes of multiple atomic operations like the removal of tuples, mark the bucket to indicate no removable items remain, update the metapage with the reduced live tuple count.

- If the system crashes in the middle of operations, it can lead to next vacuum attempts to clean the already clean bucket but overall operations will work fine.

Write-ahead-logging

- A squeeze operation moves tuples from one of the buckets later in the chain to one of the buckets earlier in the chain.
- It writes WAL record when either the bucket to which it is writing tuples is filled or bucket from which it is removing the tuples becomes empty.
- If the system crashes in the middle of this operation, after recovery, the operations will work correctly, but the index will remain bloated until the next vacuum squeeze the bucket completely.

Write-ahead-logging (verification)

- The verification for each of the operation is done with the help of WAL consistency checker (guc: wal_consistency_checking).

- This guc variable allows full page image along with WAL record for an action to be recorded and subsequently when the WAL record is replayed, it first applies the WAL and then test whether the buffers modified by the record match the stored images.

- Manual crash-recovery has been tested by crashing the system at different stages such that pages written are torn-ed.

Contents

- Hash indexes
- Locking improvements
- Improved usage of space
- Code re-factoring to enable write-ahead-logging
- Write-ahead-logging
- Hash utility functions
- Further improvements

Hash utility functions

- pgstathashindex - It provides information about hash index.

  - This function is present in pgstattuple extension.

Column	Description
version	HASH version number
bucket_pages	Number of bucket pages
overflow_pages	Number of overflow pages
bitmap_pages	Number of bitmap pages
unused_pages	Number of unused pages
live_items	Number of live tuples
dead_tuples	Number of dead tuples
free_percent	Percentage of free space

Hash utility functions

- These functions are part of pageinspect extension.
- These functions provide information about different type of hash index pages.

Function name	Description
hash_page_type	returns page type of the given HASH index page.
hash_page_stats	returns information about a bucket or overflow page of a HASH index.
hash_page_items	returns information about the data stored in a bucket or overflow page of a HASH index page.
hash_bitmap_info	shows the status of a bit in the bitmap page for a particular overflow page of HASH index.
hash_metapage_info	returns information stored in meta page of a HASH index.

Contents

- Hash indexes
- Locking improvements
- Improved usage of space
- Code re-factoring to enable write-ahead-logging
- Write-ahead-logging
- Hash utility functions
- Further improvements

Further Improvements

- Add UNIQUE capability to hash indexes.
- Speed up Create Index operation – Bypass buffer manager layer and perform insertion page-at-a-time as we do for btree indexes.
- Write performance – Each insert operation updates meta-page for which it needs to acquire a lock on meta-page.

Further Improvements

- Squeeze operation performance – We always need to take a lock on next overflow page before releasing the previous page lock in bucket chain. This has a bad impact on concurrency with other operations.

- As we don't store the key in hash index, supporting multi-column index, push down of scan keys and index-only-scan are not straight forward.

- Thanks to Ashutosh Sharma and Mithun C Y who have helped me in getting performance data for this presentation.

Thanks!