数据加密理论及基础

开源开发实践-第四周

David & Cary

CONTENT
目 录

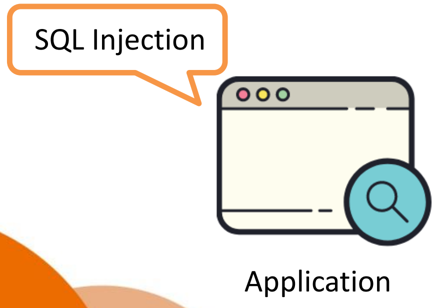
- Transparent Data Encryption in PostgreSQL
- Data Storage in PostgreSQL
- Accessing Buffer Manager
- Locking Strategies
- The Consideration – Homework

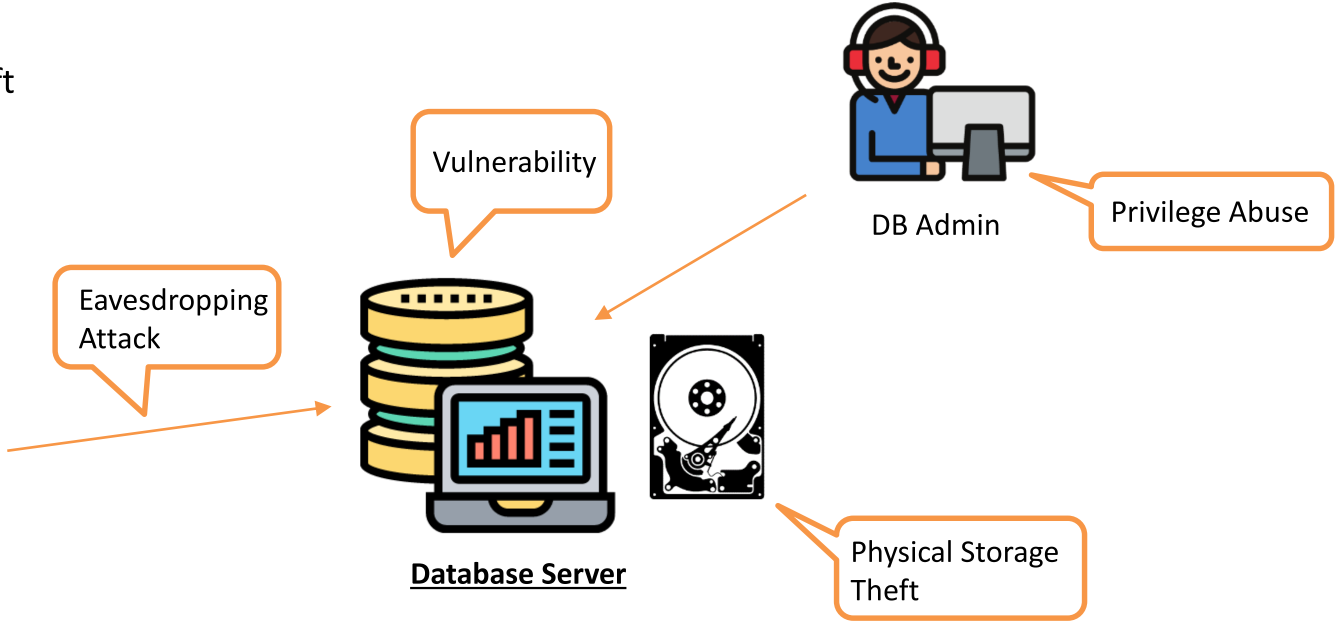
Transparent Data Encryption In PostgreSQL

Data Security

Database Security Threats

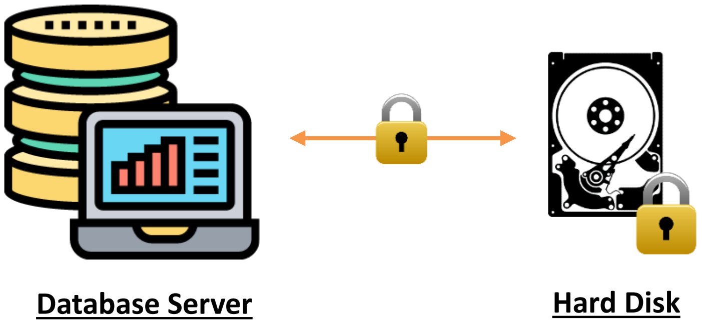
- Database server is often the primary target of the following types of attacks
  - Privilege abuse
  - SQL injection
  - Storage media theft
  - Eavesdropping
  - Man-in-the-middle





Data Security

Transparent Data Encryption Focus



Transparent Data Encryption

Also known as Data At Rest Encryption

TDE protects your database:

- From someone who can read your database file directly.
- From someone who can take a backup copy of your database
- From someone to access confidential data if the hard disk is stolen

Attention

- TDE encrypts data before it is stored on disk
- TDE decrypts data after it is read from disk
- Data is not encrypted while being used in shared memory, or over network (if TLS is not used)

TDE does NOT protect your database:

- From someone who is a malicious ‘privileged’ user from querying your data
- From man-in-the-middle attacks
- From malicious client connections

Transparent Data Encryption

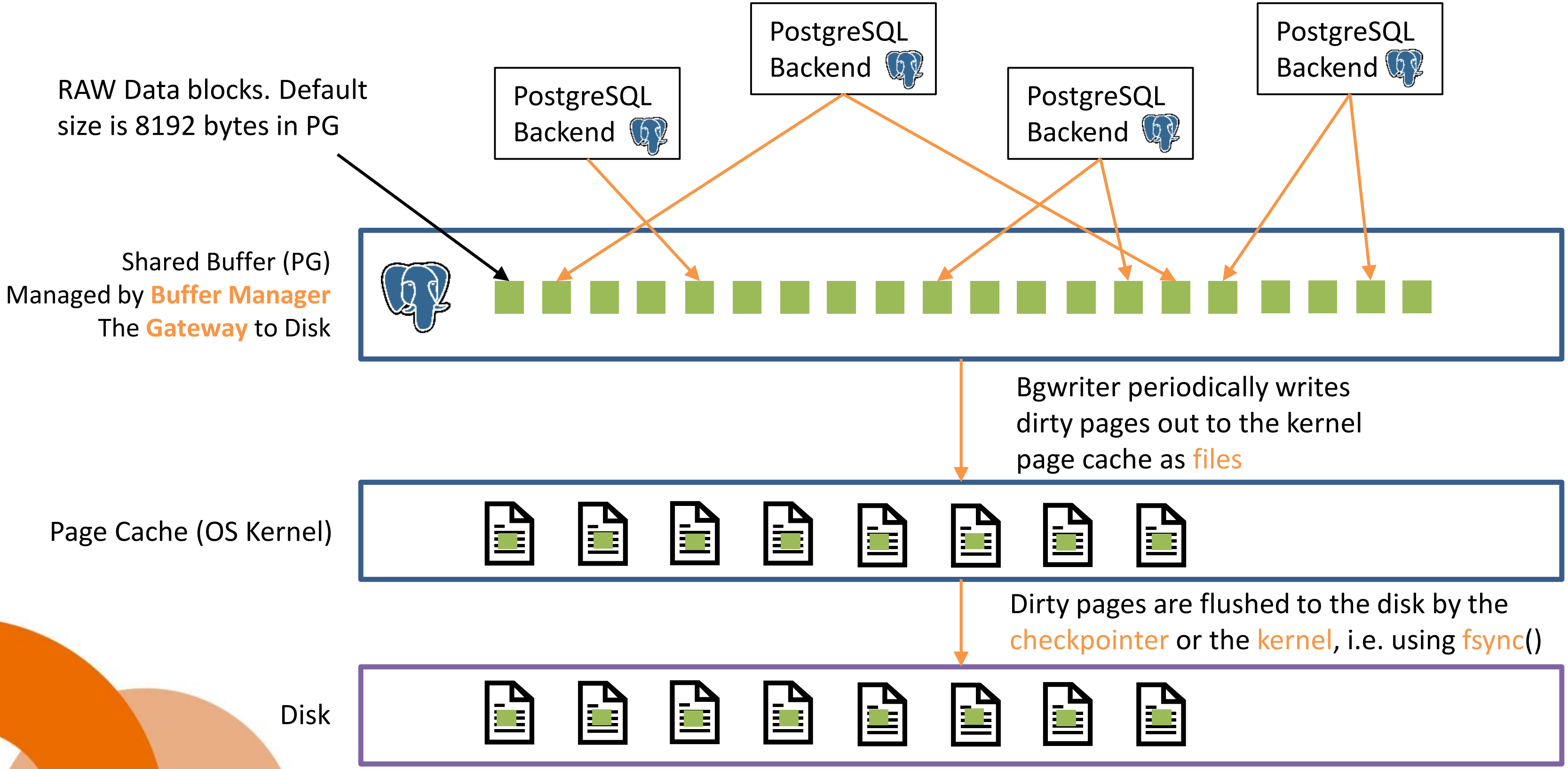
What can be encrypted?

- $\boxed{\text{Table}}$
- Indexes
- Data blocks
- TOAST Table
- WAL
- System catalog tables
- Temporary files

Data Storage in PostgreSQL

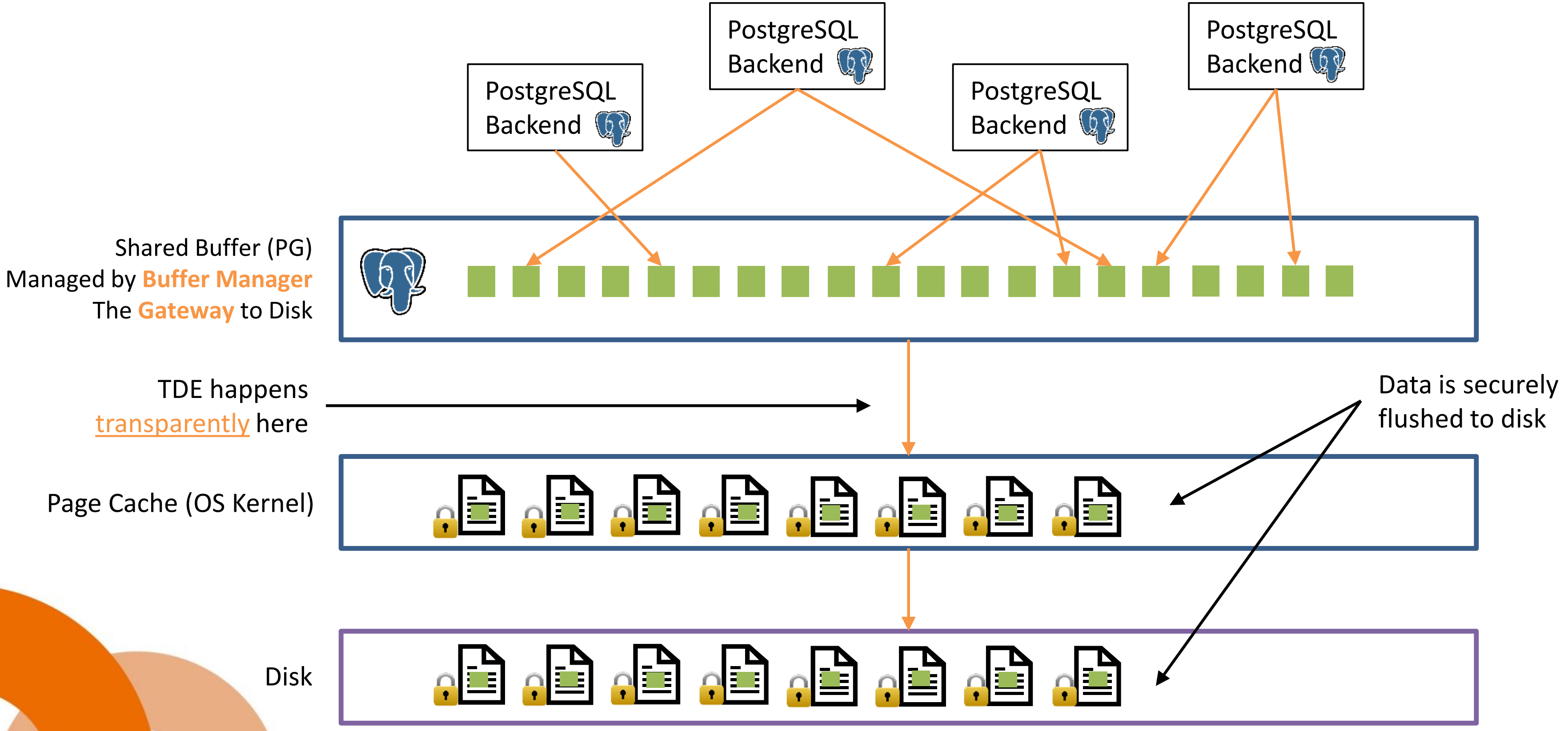
PostgreSQL I/O Architecture

Overview of PostgreSQL I/O



File Level Encryption

Adding TDE into the I/O Picture



File Level Encryption

Pros and Cons

PROS

- Relatively less execution of encryption and decryption
- No need to peek the files on disk as buffer manager manages the data placement already
- Relatively simple to implement

CONS

- Repeating encryption and decryption on the same block of data could happen quite frequently when the shared buffer size does not fit all the data of interest
- Could lead to performance issue in large SELECT queries

How PG Manages Files on Disk

Understand the Concept of Object Identifier (OID)

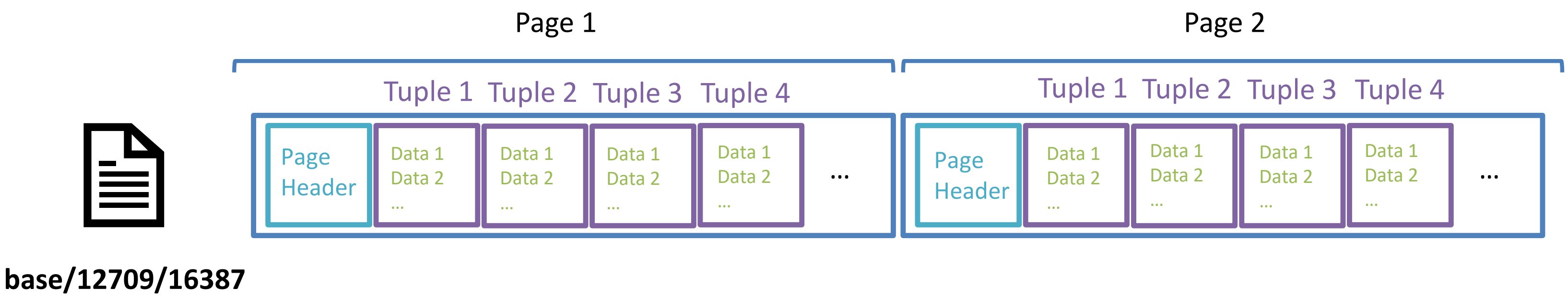
- Object identifiers (OIDs) were added to PostgreSQL to uniquely identify database objects
- **For example**, tables, indexes, functions, databases…etc
- It is still heavily used within PostgreSQL especially in system tables (catalog)
- It is important to understand the OID concept because the actual table data is stored on the disk using OID as file names.
- **For example**, a table named "**test2**" created under the database "**postgres**" will have its data stored at location "**base/12709/16387**" as seen from the output of "pg_relation_filepath".
    - 12709 is the OID of the database
    - 16387 is the OID of the table
- With the effect of TDE, the data file 16387 should be encrypted.

```
postgres=# create table test2 (a int, b char(20));
CREATE TABLE
postgres=# insert into test2 values(generate_series(1,2000),1);
INSERT 0 2000
postgres=# select pg_relation_filepath('test2');
 pg_relation_filepath
----------------------
 base/12709/16387
(1 row)

postgres=# select oid from pg_class where relname = 'test2';
  oid
-------
 16387
(1 row)

postgres=# select oid from pg_database where datname='postgres';
  oid
-------
 12709
(1 row)

postgres=#
```

How PG Manages Files on Disk

Examine Data File with 'hexdump'

- A datafile contains multiple "**blocks**" or "**pages**" of data. Each page by default is 8192 bytes. This is structured very similarly to the buffer manager's shared buffer
- Each page contains multiple "**tuples**" or "**rows**" of that table
- Each tuple contains one or more "**column data values**"



A table page layout

Page Header Data Layout

- These 2 tables are pulled from the official PostgreSQL documentation page explaining the page header data
- More details can be found in the corresponding source file at *src/include/storage/bufpage.h*
- Question to consider:

Should the page header data be encrypted by TDE?

Table 68.2. Overall Page Layout

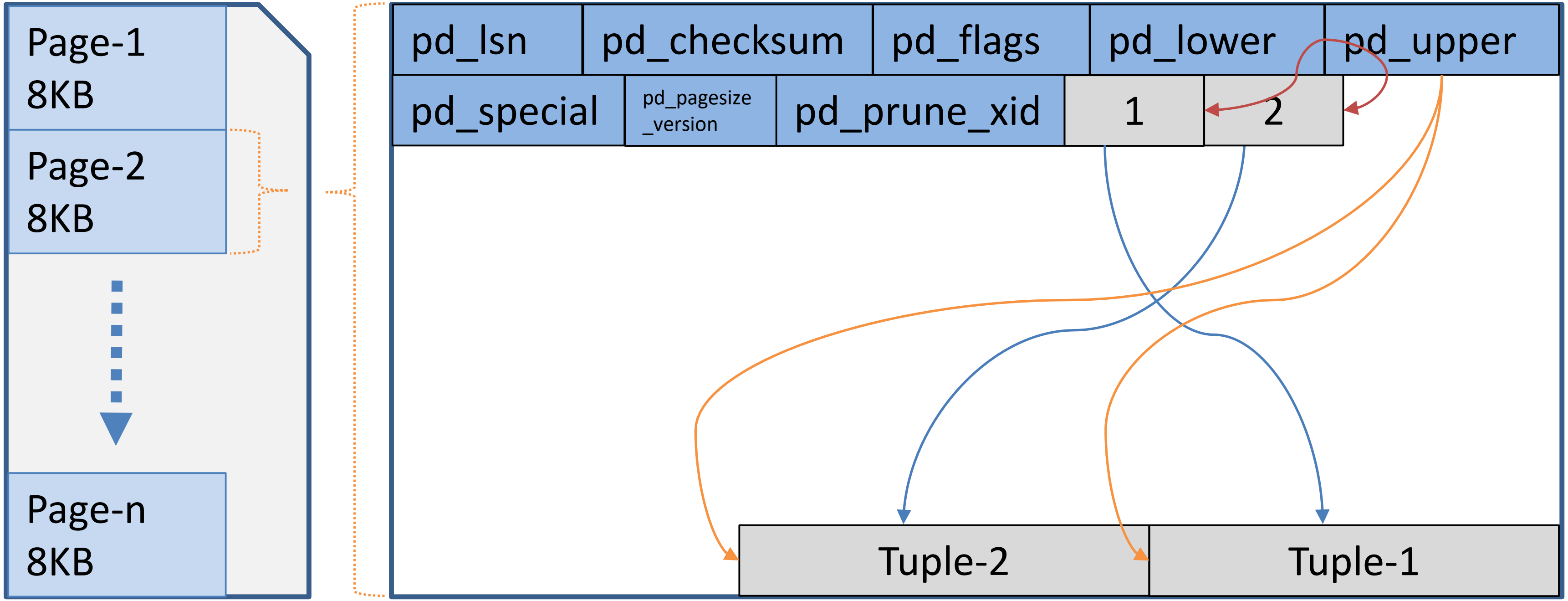
Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of item identifiers pointing to the actual items. Each entry is an (offset,length) pair. 4 bytes per item.
Free space	The unallocated space. New item identifiers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

Table 68.3. PageHeaderData Layout

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of WAL record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none

A table page layout

Insert a tuple into a page



A table page layout

Insert a tuple into a page

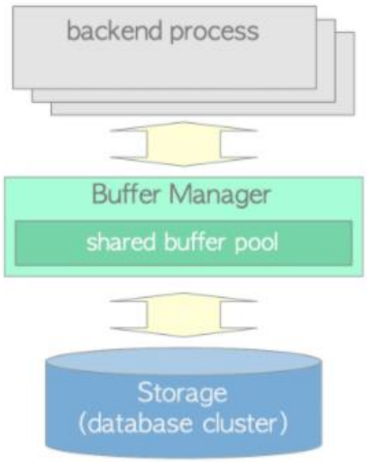
```
david@VB:~/sandbox$ psql -d postgres -p 5555
psql (13.2)
Type "help" for help.

postgres=# create table tbl(data text);
CREATE TABLE
postgres=# insert into tbl values('hello postgres week-1');
INSERT 0 1
postgres=# checkpoint ;
CHECKPOINT
postgres=# insert into tbl values('hello postgres week-2');
INSERT 0 1
postgres=# checkpoint ;
CHECKPOINT
postgres=#
```

```
david@VB: ~/sandbox/pgdata/base/12696 (ssh)
david@VB:~/sandbox/pgdata/base/12696$ ls -l 16384
-rw------- 1 david david 8192 Apr  9 14:57 16384
david@VB:~/sandbox/pgdata/base/12696$ hexdump -C 16384
00000000  00 00 00 00 90 b2 56 01  00 00 00 00 1c 00 d0 1f  |......V.........|
00000010  00 20 04 20 00 00 00 00  d0 9f 5c 00 00 00 00 00  |. . ......\.....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001fd0  e6 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001fe0  01 00 01 00 02 08 18 00  2d 68 65 6c 6c 6f 20 70  |........-hello p|
00001ff0  6f 73 74 67 72 65 73 20  77 65 65 6b 2d 31 00 00  |ostgres week-1..|
00002000
david@VB:~/sandbox/pgdata/base/12696$ hexdump -C 16384
00000000  00 00 00 00 90 b4 56 01  00 00 00 00 20 00 a0 1f  |......V..... ...|
00000010  00 20 04 20 00 00 00 00  d0 9f 5c 00 a0 9f 5c 00  |. . ......\...\.|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001fa0  e7 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001fb0  02 00 01 00 02 08 18 00  2d 68 65 6c 6c 6f 20 70  |........-hello p|
00001fc0  6f 73 74 67 72 65 73 20  77 65 65 6b 2d 32 00 00  |ostgres week-2..|
00001fd0  e6 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001fe0  01 00 01 00 02 08 18 00  2d 68 65 6c 6c 6f 20 70  |........-hello p|
00001ff0  6f 73 74 67 72 65 73 20  77 65 65 6b 2d 31 00 00  |ostgres week-1..|
00002000
```

The Almighty Buffer Manager

The heart of buffer I/O

- Buffer Manager is one of the core modules in PostgreSQL that is utilized by almost all other modules for their buffer needs
- Buffer manager manages data transfers between **shared memory** and **persistent storage** (disk) and can have a significant impact on the performance of the DBMS.
- The PostgreSQL buffer manager works very efficiently!
- It is important to understand how buffer manager works in general because it is related to the TDE feature we are about to add.



http://www.interdb.jp/pg/pgsql08.html

The Almighty Buffer Manager

The internals of buffer manager – buffer tag

- Each page can be identified by a **buffer tag**.
- A buffer tag consists of several OID values that describe the origin of a page

For example:

the buffer_tag '{(16821, 16384, 37721), 0, 7}

- identifies the page that is in the seventh block whose relation's OID and fork number are 37721 and 0
- the relation is contained in the database whose OID is 16384 under the tablespace whose OID is 16821

```
InvalidForkNumber = -1,
MAIN_FORKNUM = 0,
FSM_FORKNUM,
VISIBILITYMAP_FORKNUM,
INIT_FORKNUM
```

Can understood the type of this page
- Does page store user data?
- Does page store visibility information?
- Does page store free space information?

src/include/storage/buf_internals.h

```
typedef struct buftag
{
    RelFileNode rnode;        /* physical relation identifier */
    ForkNumber  forkNum;
    BlockNumber blockNum;     /* blknum relative to begin of reln */
} BufferTag;
```

The block number of this page with respect to **rnode**

src/include/storage/relfilenode.h

```
typedef struct RelFileNode
{
    Oid         spcNode;      /* tablespace */
    Oid         dbNode;       /* database */
    Oid         relNode;      /* relation */
} RelFileNode;
```

Describe the origin of a page:
- Belong to which database?
- Belong to Which table?
- Belong to which tablespace?

The Almighty Buffer Manager

The internals of buffer manager – buffer descriptor

- Each page can be described by a **buffer descriptor**.
- You can understand it as the meta data of each page
- contains the buffer tag structure described in previous slide and the buf_id, which is the index where the actual page data is located in the buffer pool
- Also contains a state variable, the content lock, and the pointer to the next free block in the freelist chain
- Buffer pool is just an array of actual data blocks that is by default 8192 bytes in size

```
typedef struct BufferDesc
{
    BufferTag   tag;            /* ID of page contained in buffer */
    int         buf_id;         /* buffer's index number (from 0) */

    /* state of the tag, containing flags, refcount and usagecount */
    pg_atomic_uint32 state;

    int         wait_backend_pid;   /* backend PID of pin-count waiter */
    int         freeNext;       /* link in freelist chain */

    LWLock      content_lock;   /* to lock access to buffer contents */
} BufferDesc;
```

The Almighty Buffer Manager

The internals of buffer manager – 3 layer design

Construct a buffer tag based on available information

↓

Using the buffer tag, do a hash lookup and find its buf_id (index)

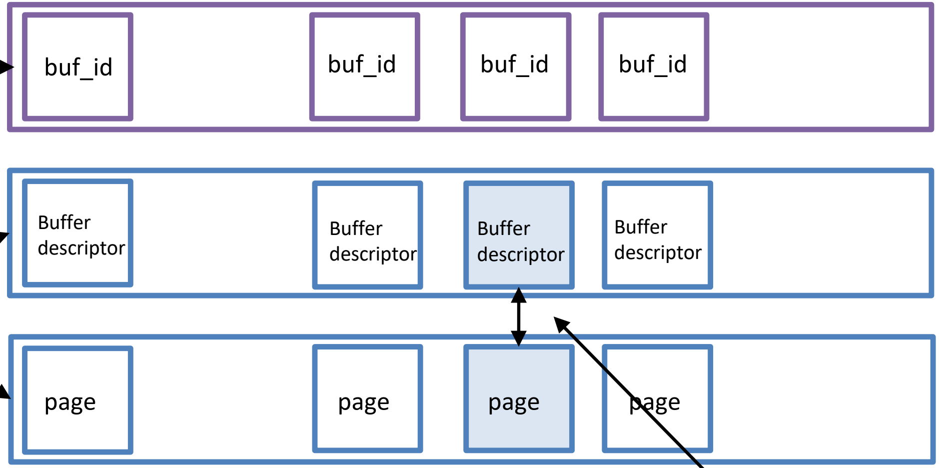
↓

With buf_id, PG can now access the page's actual data and buffer descriptor.

↓

Then make changes to the actual page data or change the buffer state

A hash table is data structure that stores key-value pairs and we use the key to look up value at instant time



Buffer Hash Table

- Key = buffer tag
- Value = buf_id (the index)

Buffer Descriptor Array

Buffer Pool Array

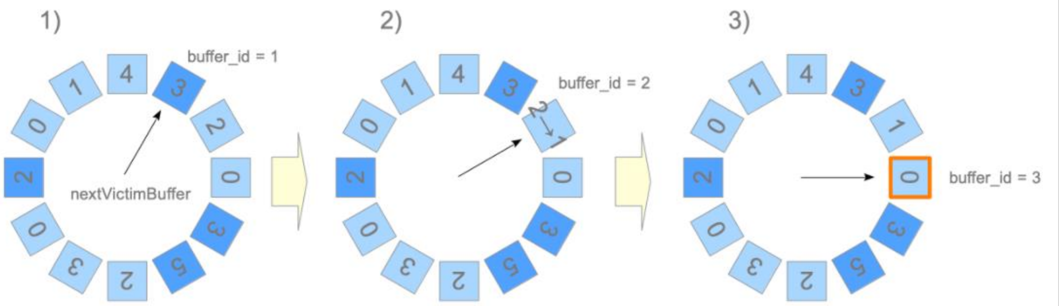
Buf_id represents the index of buffer descriptor and actual page in their respective arrays

The Almighty Buffer Manager

Page Replacement Algorithm – Clock Sweep

- What happens if the buffer is full when we need to insert new page to the buffer?
- We need to eject one old page from the buffer to free up space. The selected page to be ejected is called a "victim" .
- The answer is "clock sweep" algorithm. Imaging the buffer descriptor array as a circular buffer.
- The algorithm only works on buffer that is unpinned, meaning not being used by other backends currently.
- The algorithm loops around the circle clockwise and decrease the usage_count by 1 until a buffer whose usage_count = 0 is found. This is the buffer to be ejected

1. The nextVictimBuffer points to the first descriptor (buffer_id 1); however, this descriptor is skipped because it is pinned.
2. The nextVictimBuffer points to the second descriptor (buffer_id 2). This descriptor is unpinned but its usage_count is 2; thus, the usage_count is decreased by 1 and the nextVictimBuffer advances to the third candidate.
3. The nextVictimBuffer points to the third descriptor (buffer_id 3). This descriptor is unpinned and its usage_count is 0; thus, this is the victim in this round.



http://www.interdb.jp/pg/pgsql08.html

The Almighty Buffer Manager

Page Replacement Algorithm - Clock Sweep

```c
/* Nothing on the freelist, so run the "clock sweep" algorithm */
trycounter = NBuffers;
for (;;)
{
	buf = GetBufferDescriptor(ClockSweepTick());

	/*
	 * If the buffer is pinned or has a nonzero usage_count, we cannot use
	 * it; decrement the usage_count (unless pinned) and keep scanning.
	 */
	local_buf_state = LockBufHdr(buf);

	if (BUF_STATE_GET_REFCOUNT(local_buf_state) == 0)
	{
		if (BUF_STATE_GET_USAGECOUNT(local_buf_state) != 0)
		{
			local_buf_state -= BUF_USAGECOUNT_ONE;

			trycounter = NBuffers;
		}
		else
		{
			/* Found a usable buffer */
			if (strategy != NULL)
				AddBufferToRing(strategy, buf);
			*buf_state = local_buf_state;
			return buf;
		}
	}
	else if (--trycounter == 0)
	{
		/*
		 * We've scanned all the buffers without making any state changes,
		 * so all the buffers are pinned (or were when we looked at them).
		 * We could hope that someone will free one eventually, but it's
		 * probably better to fail than to risk getting stuck in an
		 * infinite loop.
		 */
		UnlockBufHdr(buf, local_buf_state);
		elog(ERROR, "no unpinned buffers available");
	}
	UnlockBufHdr(buf, local_buf_state);
}
```

The Almighty Buffer Manager

Dirty and Clean Page

- The concept of dirty or clean (not dirty) page is super important in buffer manager
- If a page is marked as ‘dirty’, it means that the contents of this page has been changed and it differs from the page that is physically stored on the disk.
- If a page is not ‘dirty’, it means that the contents of this page has not been changed and it is the same as the page that is physically stored on the disk.
- Why is this important?
    - This flag could prevent a lot of unnecessary writes to disk, which could be a time-consuming task.
    - 2 background processes “checkpointer” and “background writer” are responsible for “flushing dirty buffers” to disk periodically
    - Buffers not marked as “dirty” are not necessary to be flushed to disk
    - When dirty buffer is ejected by Clock Sweep algorithm in previous slide, it will also be flushed to disk
    - When a clean buffer is ejected by Clock Sweep algorithm in previous slide, it is not necessary to flush to disk

The Almighty Buffer Manager

The Ring Buffer

- The ring buffer is different from the clock sweep algorithm where we image the buffer descriptor buffer to be like a circle.
- Ring buffer is a temporary buffer area that is only allocated when there is a large read/write query. In this case, buffer pool, will not be used to process this large query
- Why is that?
    - We all know that reading from memory is much faster than reading from disk
    - Without ring buffer, the pages in buffer pool may all have to be ejected to accommodate the pages that the big read/write query requires.
    - This will reduce the cache hit ratio and affects the performance of the database
    - Having a ring buffer that is allocated on-demand when there is a huge read/write query can avoid this issue
    - The ring buffer is destroyed after use.

Accessing Buffer Manager

Accessing Buffer Manager

The Entry Point

- Most of the backend modules access the buffer manager via the "ReadBufferExtended()" function
- This function has 3 logical cases and we will describe all of them here in this module
- Takes 5 arguments
  - Relation
  - ForkNumber
  - BlockNumber
  - ReadBufferMode
  - BufferAccessStrategy

These 3 form a buffer tag

Controls how buffer manager handles error or eject data from buffer. Will not look in detail

- This function will return a buf_id to the caller.
- Remember, buf_id is simply an index number to the buffer pool and buffer descriptor arrays

src/backend/storage.bufmgr.c

```
Buffer
ReadBufferExtended(Relation reln, ForkNumber forkNum, BlockNumber blockNum,
                   ReadBufferMode mode, BufferAccessStrategy strategy)
{
    bool        hit;
    Buffer      buf;

    /* Open it at the smgr level if not already done */
    RelationOpenSmgr(reln);

    /*
     * Reject attempts to read non-local temporary relations; we would be
     * likely to get wrong data since we have no visibility into the owning
     * session's local buffers.
     */
    if (RELATION_IS_OTHER_TEMP(reln))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                 errmsg("cannot access temporary tables of other sessions")));

    /*
     * Read the buffer, and update pgstat counters to reflect a cache hit or
     * miss.
     */
    pgstat_count_buffer_read(reln);
    buf = ReadBuffer_common(reln->rd_smgr, reln->rd_rel->relpersistence,
                            forkNum, blockNum, mode, strategy, &hit);
    if (hit)
        pgstat_count_buffer_hit(reln);
    return buf;
}
```

Accessing Buffer Manager

Case 1: Accessing A Page in Buffer Pool

src/backend/storage/bufmgr.c

- This is done in a sub-routine called "**BufferAlloc**"
- First it creates a Buffer Tag using relation, block number and fork number.
- prepare the hash table and partition lock
- Do the hash lookup and try to find a **buf_id** from the hash table
- If an entry is found, that means the target block **exists** in shared buffer and it can be used right away
- So simply just look up its buffer descriptor, pin it and return it
- Remember, buffer descriptor contains everything about the data block, including the lock, status and location. So, knowing the buffer descriptor, the caller can find and make changes to a data block

```c
static BufferDesc *
BufferAlloc(SMgrRelation smgr, char relpersistence, ForkNumber forkNum,
            BlockNumber blockNum,
            BufferAccessStrategy strategy,
            bool *foundPtr)
{
    BufferTag   newTag;         /* identity of requested block */
    uint32      newHash;        /* hash value for newTag */
    LWLock     *newPartitionLock;   /* buffer partition lock for it */
    BufferTag   oldTag;         /* previous identity of selected buffer */
    uint32      oldHash;        /* hash value for oldTag */
    LWLock     *oldPartitionLock;   /* buffer partition lock for it */
    uint32      oldFlags;
    int         buf_id;
    BufferDesc *buf;
    bool        valid;
    uint32      buf_state;

    /* create a tag so we can lookup the buffer */
    INIT_BUFFERTAG(newTag, smgr->smgr_rnode.node, forkNum, blockNum);

    /* determine its hash code and partition lock ID */
    newHash = BufTableHashCode(&newTag);
    newPartitionLock = BufMappingPartitionLock(newHash);

    /* see if the block is in the buffer pool already */
    LWLockAcquire(newPartitionLock, LW_SHARED);
    buf_id = BufTableLookup(&newTag, newHash);

    if (buf_id >= 0)
    {
        /*
         * Found it.  Now, pin the buffer so no one can steal it from the
         * buffer pool, and check to see if the correct data has been loaded
         * into the buffer.
         */
        buf = GetBufferDescriptor(buf_id);

        valid = PinBuffer(buf, strategy);

        /* Can release the mapping lock as soon as we've pinned it */
        LWLockRelease(newPartitionLock);

        *foundPtr = true;

        return buf;
    }
```

Accessing Buffer Manager

Case 2: Page is not found in Buffer Pool

- If a data page is not found in case 1, then 2 things can happen
- (1) if a page is NEW and does not exist any where yet. Create it. This is also called “extend” and it will call storage manager’s **smgrextend()** routine to add one more page data
- (2) if a page exists already on disk, load it into shared buffer and it will call storage manager’s **smgrread()** routine to read the page’s data from disk to shared buffer

```c
bufBlock = isLocalBuf ? LocalBufHdrGetBlock(bufHdr) : BufHdrGetBlock(bufHdr);

if (isExtend)
{
    /* new buffers are zero-filled */
    MemSet((char *) bufBlock, 0, BLCKSZ);
    /* don't set checksum for all-zero page */
    smgrextend(smgr, forkNum, blockNum, (char *) bufBlock, false);

    /*
     * NB: we're *not* doing a ScheduleBufferTagForWriteback here;
     * although we're essentially performing a write. At least on linux
     * doing so defeats the 'delayed allocation' mechanism, leading to
     * increased file fragmentation.
     */
}
else
{
    /*
     * Read in the page, unless the caller intends to overwrite it and
     * just wants us to allocate a buffer.
     */
    if (mode == RBM_ZERO_AND_LOCK || mode == RBM_ZERO_AND_CLEANUP_LOCK)
        MemSet((char *) bufBlock, 0, BLCKSZ);
    else
    {
        instr_time  io_start,
                    io_time;

        if (track_io_timing)
            INSTR_TIME_SET_CURRENT(io_start);

        smgrread(smgr, forkNum, blockNum, (char *) bufBlock);

        if (track_io_timing)
        {
            INSTR_TIME_SET_CURRENT(io_time);
            INSTR_TIME_SUBTRACT(io_time, io_start);
            pgstat_count_buffer_read_time(INSTR_TIME_GET_MICROSEC(io_time));
            INSTR_TIME_ADD(pgBufferUsage.blk_read_time, io_time);
        }

        /* check for garbage data */
        if (!PageIsVerifiedExtended((Page) bufBlock, blockNum,
                                    PIV_LOG_WARNING | PIV_REPORT_STAT))
        {
            if (mode == RBM_ZERO_ON_ERROR || zero_damaged_pages)
            {
                ereport(WARNING,
                        (errcode(ERRCODE_DATA_CORRUPTED),
                         errmsg("invalid page in block %u of relation %s; zeroing out page",
                                blockNum,
                                relpath(smgr->smgr_rnode, forkNum))));
                MemSet((char *) bufBlock, 0, BLCKSZ);
            }
            else
                ereport(ERROR,
                        (errcode(ERRCODE_DATA_CORRUPTED),
                         errmsg("invalid page in block %u of relation %s",
                                blockNum,
                                relpath(smgr->smgr_rnode, forkNum))));
        }
    }
}
```

Accessing Buffer Manager

Case 3: Page is not Found in Buffer Pool And Buffer Pool is Full

- If a data page is not found in case 1, and it needs to load from the disk, but current buffer pool is **full**.
- Then it needs to eject one or more data pages (called **victims**) from shared buffer by using the **clock-sweep** algorithm

src/backend/storage/freelist.c

```c
/* Nothing on the freelist, so run the "clock sweep" algorithm */
trycounter = NBuffers;
for (;;)
{
    buf = GetBufferDescriptor(ClockSweepTick());

    /*
     * If the buffer is pinned or has a nonzero usage_count, we cannot use
     * it; decrement the usage_count (unless pinned) and keep scanning.
     */
    local_buf_state = LockBufHdr(buf);

    if (BUF_STATE_GET_REFCOUNT(local_buf_state) == 0)
    {
        if (BUF_STATE_GET_USAGECOUNT(local_buf_state) != 0)
        {
            local_buf_state -= BUF_USAGECOUNT_ONE;

            trycounter = NBuffers;
        }
        else
        {
            /* Found a usable buffer */
            if (strategy != NULL)
                AddBufferToRing(strategy, buf);
            *buf_state = local_buf_state;
            return buf;
        }
    }
    else if (--trycounter == 0)
    {
        /*
         * We've scanned all the buffers without making any state changes,
         * so all the buffers are pinned (or were when we looked at them).
         * We could hope that someone will free one eventually, but it's
         * probably better to fail than to risk getting stuck in an
         * infinite loop.
         */
        UnlockBufHdr(buf, local_buf_state);
        elog(ERROR, "no unpinned buffers available");
    }
    UnlockBufHdr(buf, local_buf_state);
}
```

Accessing Buffer Manager

Case 3: How to Eject a Page?

- As we have discussed, a page can either be marked as “dirty” or “not dirty”
- If “dirty”, we need to write and flush to the disk immediately by calling “FlushBuffer()”
- If “not dirty”, we simply remove it from the shared buffer.

```c
if (oldFlags & BM_DIRTY)
{
    /*
     * We need a share-lock on the buffer contents to write it out
     * (else we might write invalid data, eg because someone else is
     * compacting the page contents while we write).  We must use a
     * conditional lock acquisition here to avoid deadlock.  Even
     * though the buffer was not pinned (and therefore surely not
     * locked) when StrategyGetBuffer returned it, someone else could
     * have pinned and exclusive-locked it by the time we get here. If
     * we try to get the lock unconditionally, we'd block waiting for
     * them; if they later block waiting for us, deadlock ensues.
     * (This has been observed to happen when two backends are both
     * trying to split btree index pages, and the second one just
     * happens to be trying to split the page the first one got from
     * StrategyGetBuffer.)
     */
    if (LWLockConditionalAcquire(BufferDescriptorGetContentLock(buf),
                                 LW_SHARED))
    {
        /*
         * If using a nondefault strategy, and writing the buffer
         * would require a WAL flush, let the strategy decide whether
         * to go ahead and write/reuse the buffer or to choose another
         * victim.  We need lock to inspect the page LSN, so this
         * can't be done inside StrategyGetBuffer.
         */
        if (strategy != NULL)
        {
            XLogRecPtr  lsn;

            /* Read the LSN while holding buffer header lock */
            buf_state = LockBufHdr(buf);
            lsn = BufferGetLSN(buf);
            UnlockBufHdr(buf, buf_state);

            if (XLogNeedsFlush(lsn) &&
                StrategyRejectBuffer(strategy, buf))
            {
                /* Drop lock/pin and loop around for another buffer */
                LWLockRelease(BufferDescriptorGetContentLock(buf));
                UnpinBuffer(buf, true);
                continue;
            }
        }

        /* OK, do the I/O */
        TRACE_POSTGRESQL_BUFFER_WRITE_DIRTY_START(forkNum, blockNum,
                                                  smgr->smgr_rnode.node.spcNode,
                                                  smgr->smgr_rnode.node.dbNode,
                                                  smgr->smgr_rnode.node.relNode);

        FlushBuffer(buf, NULL);
        LWLockRelease(BufferDescriptorGetContentLock(buf));

        ScheduleBufferTagForWriteback(&BackendWritebackContext,
                                      &buf->tag);
```

Accessing Buffer Manager

Where Could TDE Take Place?

- We know that TDE is all about:
    - encrypting data before writing to disk.
    - decrypting data after reading from disk.
- Where is the code that does reading and writing from disk?
- We have "**smgrread**" to read from file
- And "**FlushBuffer**" to write to disk
- Start to think about how TDE can be applied here…
- Start to think about which cryptographic algorithm is more suitable for buffer manager's data structure…

```c
if (mode == RBM_ZERO_AND_LOCK || mode == RBM_ZERO_AND_CLEANUP_LOCK)
    MemSet((char *) bufBlock, 0, BLCKSZ);
else
{
    instr_time  io_start,
                io_time;

    if (track_io_timing)
        INSTR_TIME_SET_CURRENT(io_start);

    smgrread(smgr, forkNum, blockNum, (char *) bufBlock);

    if (track_io_timing)
    {
        INSTR_TIME_SET_CURRENT(io_time);
        INSTR_TIME_SUBTRACT(io_time, io_start);
        pgstat_count_buffer_read_time(INSTR_TIME_GET_MICROSEC(io_time));
        INSTR_TIME_ADD(pgBufferUsage.blk_read_time, io_time);
    }
```

```c
/*
 * FlushBuffer
 *      Physically write out a shared buffer.
 *
 * NOTE: this actually just passes the buffer contents to the kernel; the
 * real write to disk won't happen until the kernel feels like it.  This
 * is okay from our point of view since we can redo the changes from WAL.
 * However, we will need to force the changes to disk via fsync before
 * we can checkpoint WAL.
 *
 * The caller must hold a pin on the buffer and have share-locked the
 * buffer contents.  (Note: a share-lock does not prevent updates of
 * hint bits in the buffer, so the page could change while the write
 * is in progress, but we assume that that will not invalidate the data
 * written.)
 *
 * If the caller has an smgr reference for the buffer's relation, pass it
 * as the second parameter.  If not, pass NULL.
 */
static void
FlushBuffer(BufferDesc *buf, SMgrRelation reln)
{
    XLogRecPtr  recptr;
    ErrorContextCallback errcallback;
    instr_time  io_start,
                io_time;
    Block       bufBlock;
    char       *bufToWrite;
    uint32      buf_state;
```

Locking Strategies

Locking Strategies

Types of Software Locks

- Locking may not have any direct relation to the TDE feature we are going to add, but it is a very important topic, and it is important to have a fair understanding of different types of locks.
- Locking is sometimes referred as a **synchronization technique**.
- We can "**acquire**" a lock before accessing a resource and "**release**" a lock after.
- Common types of software locks:

Mutex (Mutual Exclusion)

  - A simple lockable object
  - Only one thread can acquire the lock at a time
  - Only the thread who have acquired the lock can release
  - Acquiring a locked mutex could result in failure or blocking

Locking Strategies

Types of Software Locks

Semaphore

- A semaphore is a very relaxed type of lockable object
- Has a predefined maximum count and current count.
- Acquire a semaphore by a “wait” operation and release with a “signal” operation
- When we do a “wait” operation, current count is decremented and “wait” will return and software can proceed. If the current count cannot be decremented (ie = 0), then “wait” will block until someone else increases the current count.
- When we do a “signal” operation, current count is incremented. A thread blocking in the “wait” operation will be woken up and proceed.
- If there are multiple threads blocking in the “wait” operation, a single “signal” operation will only wake up ONE thread.

Locking Strategies

Types of Software Locks

Spinlock

- It is a special type of Mutex that does not use OS synchronization functions when a lock operation has to wait.
- Instead, it just keeps “spinning” or “looping” until you have acquired what you need.
- If the lock is held for a very short time, it can be more efficient than regular mutex
- However, if the process has to be locked for a long time, then it is just wasting time doing nothing.
- In this case, mutex would do better.

src/backend/storage/bufmgr.c

```
uint32
LockBufHdr(BufferDesc *desc)
{
    SpinDelayStatus delayStatus;
    uint32      old_buf_state;

    init_local_spin_delay(&delayStatus);

    while (true)
    {
        /* set BM_LOCKED flag */
        old_buf_state = pg_atomic_fetch_or_u32(&desc->state, BM_LOCKED);
        /* if it wasn't set before we're OK */
        if (!(old_buf_state & BM_LOCKED))
            break;
        perform_spin_delay(&delayStatus);
    }
    finish_spin_delay(&delayStatus);
    return old_buf_state | BM_LOCKED;
}
```

Locking Strategies

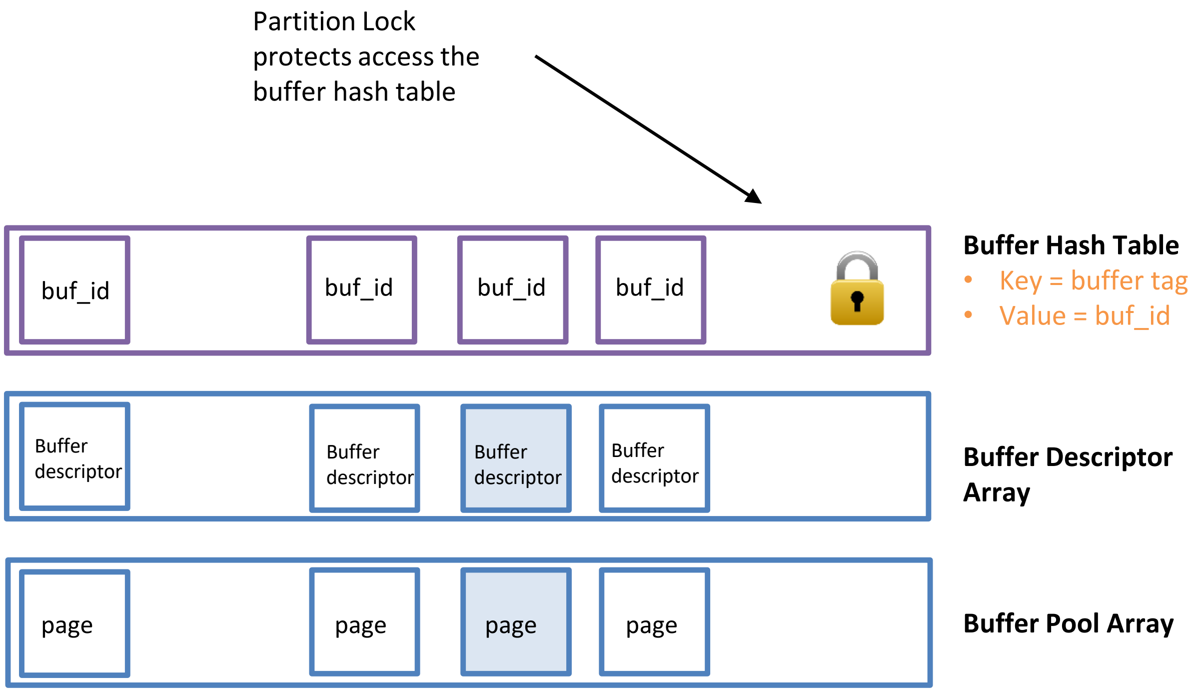
How Many Locks Are Used In PostgreSQL?

- All of them are used in PostgreSQL!
- The ability to ensure data can be correctly processed in a highly concurrent environment such as PostgreSQL is very, very important.
- It does not just apply to PostgreSQL
- It applies to probably almost all the software you will encounter in the future
- Buffer Manager uses 4 locks to protect the buffer data
  - Partition_lock (mutex)
  - Content_lock (mutex)
  - Io_in_progress_lock (mutex)
  - Spinlock

Locking Strategies

Partition Lock

- Also known as “BufMappingLock”
- It is a type of mutex lock that is used to protect “Buffer Hash Table” that backends use to insert or look up a buffer block location (buf_id)
- Can be declared as SHARED or EXLUSIVE.



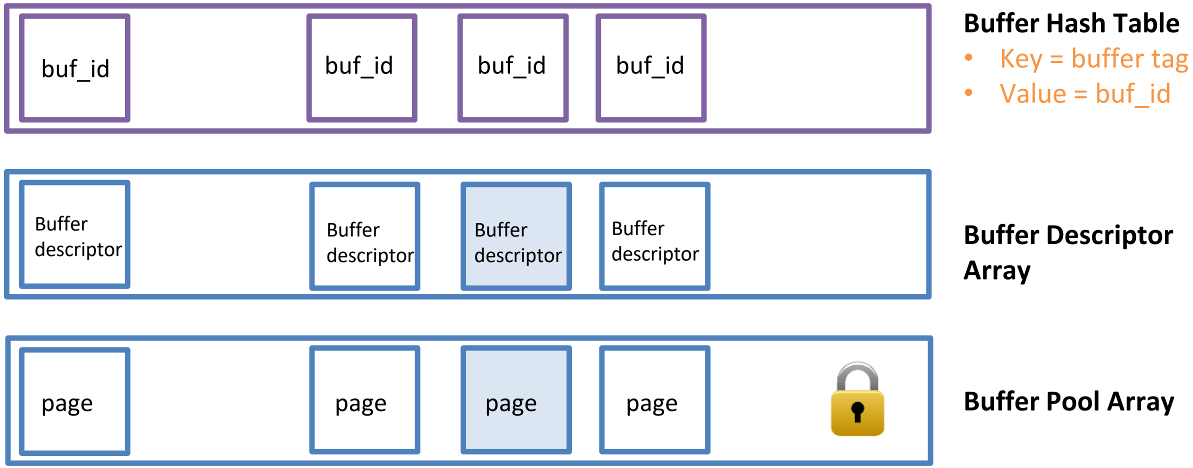
Locking Strategies

Content Lock

- It is a type of mutex lock that is used to protect “Buffer Pool” to synchronize read and write to each buffer block.
- Can be declared as SHARED or EXLUSIVE.
- Each block has its own content lock stored in buffer descriptor
- Exclusive lock is acquired when doing the following:
    - Inserting new tuples
    - Removing tuples

```
typedef struct BufferDesc
{
    BufferTag   tag;            /* ID of page contained in buffer */
    int         buf_id;         /* buffer's index number (from 0) */

    /* state of the tag, containing flags, refcount and usagecount */
    pg_atomic_uint32 state;

    int         wait_backend_pid;  /* backend PID of pin-count waiter */
    int         freeNext;       /* link in freelist chain */

    LWLock      content_lock;   /* to lock access to buffer contents */
} BufferDesc;
```

Content lock protects backend access to the actual data block



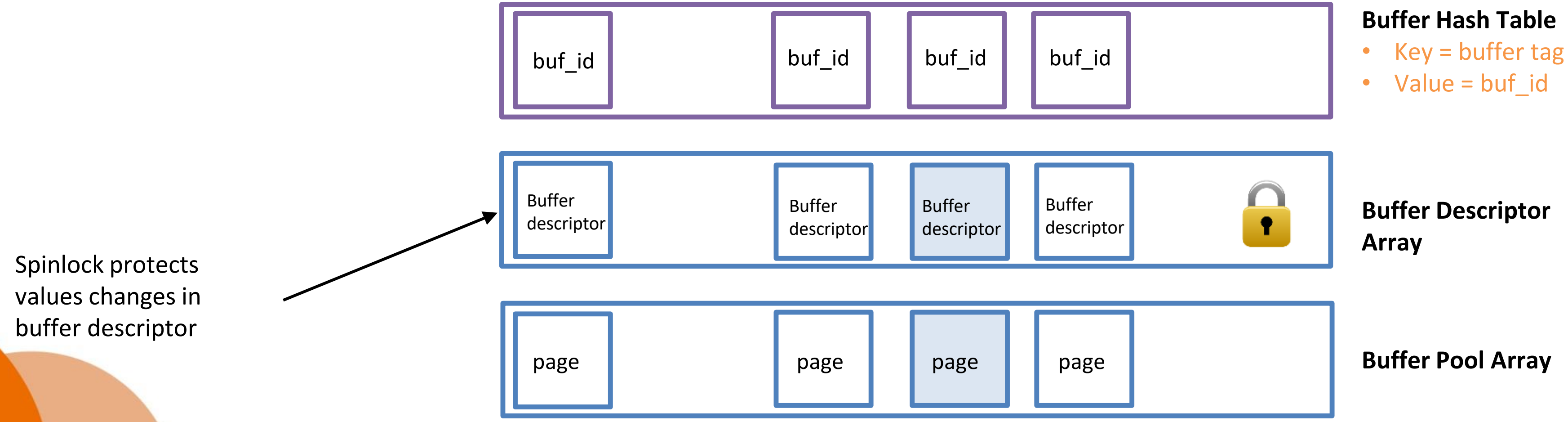
Locking Strategies

Spinlock

- Spinlock is used to protect some of the most commonly updated/checked values in Buffer descriptor layer.
- When these values are checked or updated, a spinlock is used
- The lock duration is very short

```
LockBufHdr(bufferdesc);    /* Acquire a spinlock */
bufferdesc->refcont++;
bufferdesc->usage_count++;
UnlockBufHdr(bufferdesc); /* Release the spinlock */
```

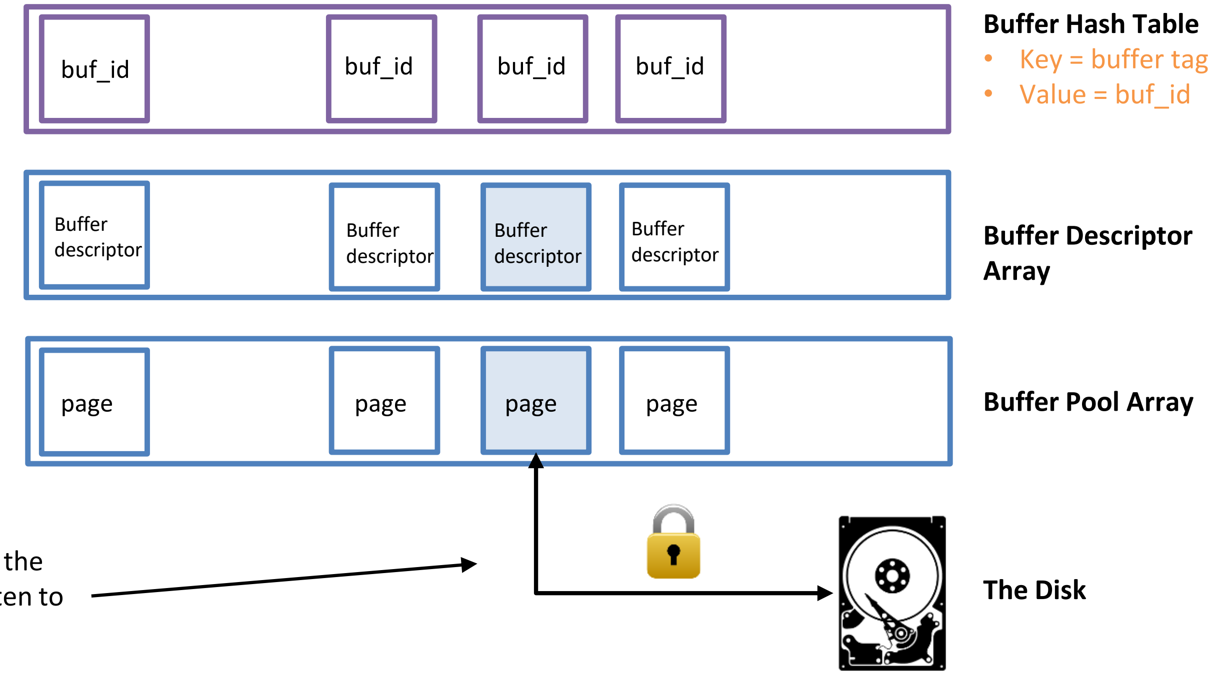
```
LockBufHdr(bufferdesc);
bufferdesc->flags |= BM_DIRTY;
UnlockBufHdr(bufferdesc);
```



Locking Strategies

I/O_In_Progress_Lock

- I/O in progress lock is a mutext lock that protects the read/write operation from shared buffer and the disk
- It must be acquired as EXCLUSIVE lock, to prevent other backend processes to intervene with the I/O
- Each block has its own I/O in progress lock declared.



TDE Consideration – Homework

瀚高
HIGH GO

融知与行 瀚且高远

THANKS