The PrimeBase XT Transactional Engine

A WHITE PAPER

by Paul McCullagh

Version 1.02
20 March 2006

INTRODUCTION

The PrimeBase XT Engine (PBXT) is a transactional database engine designed for high concurrency. PBXT uses a number of new techniques to achieve its goals. In this white paper I describe the PBXT design and various features of the implementation.

THE ACID TEST

To say that PBXT is transactional engine means that it passes the ACID test. Briefly, this means:

- **Atomic:** All or nothing: all of a transaction is committed or none of it.
- **Consistency:** Maintaining integrity: a transaction may not violate database constraints such as a unique index.

- **Isolation:** The illusion of being alone: transactions are not affected by activity of others on the database. This means that transactions are serializable.

- **Durability:** Once committed, always committed: the program has a mechanism to guarantee persistence.

Where relevant I make reference to these constraints in the discussion of the design aspects of PBXT in the following sections.

ONCE WRITTEN TWICE SHY

One of the major differences between XT and a conventional transaction engine is that the standard engine writes all data twice: once to the log file and once to the database table. This is required for a number of reasons:

> **Durability:** writing a transaction first to the log ensures that on a crash the information written can be recovered and written to the database table(s) as far as it is missing.
>
> **Atomic:** if an error occurs we have a record in the log of what was done, in order to undo changes.

PBXT is transactional, but only writes data once namely, to the log. This has a number of consequences:

- Firstly the obvious: writing less, saves times.

- The database table data actually resides in the logs (how we get to the table data is explained below in the section "Getting a Handle on the Problem").

- Since table data is not stored in any particular table file, the size of a table is unlimited.

- Since log files are written sequentially, PBXT only ever writes transaction data sequentially.

- Since data is only written sequentially, variable sized rows pose no problem to PBXT. In fact, there is actually no need to specify a maximum size for column types such as VARCHAR, VARBIN or BLOBs.

- Since log file data is written immediately, records in the cache are never "dirty" (i.e. must be flushed before being freed). This greatly speeds up and simplifies cache management.

- Since log files are never updated, PBXT never modifies table data in place. This means that instead of updating a record, PBXT writes a new record.

- Since database table data is never updated rollback never has to undo a change.

- Since data is never overwritten, the internal data structure is inherently more stable than a standard implementation.

PBXT takes this idea a step further. In a standard implementation one transaction log is used. To reduce contention in a high concurrency environment, PBXT uses one log per thread of execution. This effectively means each transaction has its own log.

Which in turn means that the order of commit can no longer be determined by the log file. Since this information is important for serializability it is stored elsewhere (see section "To Commit or Not to commit").

When a log is full, i.e. reaches a predetermined threshold, the thread automatically creates a new log. However, it is still possible that a log grow beyond this threshold because a single transaction may not span more than one log.

TAKING OUT THE TRASH

In general, databases spend a fair amount of time managing the disk space provided. When rows are deleted or moved holes appear in the underlying files that store the database table data. Log files become full and need to be removed or archived. If this job is done "on-the-fly", then the speed of transactions is affected accordingly.

PBXT takes an unusual (for database systems) approach to this problem: it uses a "Garbage Collector". This means that unused space is not collected on the fly, but by a background process which is able to recognize that file space is no longer referenced.

It is clear from the discussion in the section "Write Once not Twice", that if a record is updated, once the change has been committed, the old record is garbage. This is also the case if a record is deleted.

Since all records are in log files, the Garbage Collector cannot simple free the record. Instead it waits until the amount of garbage in a log reaches a certain threshold. At this

time it reads the log sequentially and copies the valid records to a new log. When this job is complete the old log can be deleted.

In this way unused space is reclaimed independently of the rest of the system. This job is also done as a low-priority background task so as not to affect the speed of the user transactions running in the foreground.

GETTING A HANDLE ON THE PROBLEM

So how is it possible that a record is copied from one log to another without affecting the rest of the system? And how do we find the records that belong to a table anyway?

The solution to this problem is probably obvious: by using one more level of indirection, i.e. a handle.

PBXT makes extensive use of handles. References to records are stored as 8-byte handles. The first 4 bytes are a reference to the "handle file", and the second 4 bytes are a "handle index". Handle files are memory mapped for quick access.

The actual reference to the record is stored in the handle as a 10-byte value: a 4-byte "data log file" number, and a 6-byte log offset. The 4-byte record size is also stored in the handle so that the system knows how big the buffer must be before it reads the record.

To reduce contention when allocating handles, each thread has its own handle file.

So all the Garbage Collector has to do is update the handle of each record as it is copied to the new record. The record in the old log file is then no longer referenced and this makes it garbage. So you could say the job of the Garbage Collector is to turn the entire log file into garbage, at which point the file can be deleted.

Now if we consider the problem of finding the records belonging to a table then handles are only part of the solution, the "Row ID" concept is the other part, and this is discussed in the next section.

RETURN OF THE ROW ID

A PBXT table is represented by a file. The file contains a list of handles, which reference the records of the table. The index of the handle provides a unique ID for the row: the Row ID.

Access to the row handle is optimized by memory mapping the table file.

Unlike implementations that identify a row by its offset in a table data file, PBXT row IDs do not change if a record is updated. Conventionally this is required when a row is updated, and due to variable length columns it no longer fits into the slot previously occupied. This means the record must be move to a new location which changes the row offset.

This referential stability simplifies the implementation in a number of instances. In addition, by declaring an alias for the Row ID, a user can use the row ID in place of a standard "auto increment" type column.

READING FROM THE PAST

By now it is clear that the previous version of a record is available, at the very least, until a modifying transaction has been committed. So the logical consequence of this is to make access to these records available to reading transactions. This, I refer to as "reading from the past", and besides providing a necessary mechanism to guarantee isolation of transactions it has many applications.

The first, and most important, is eliminating the need for read locks. Since read locks are the single, greatest source of deadlocks in transactional systems, this is a great advantage.

Further applications include replication, producing a snapshot backup and restoration of a previous point in time.

Storing this information is a simple extension to the structure described so far. Instead of a row ID referencing a single record, it references a chain of records. As we follow the chain further away from the reference in the table file we go further into the past. Each record instance is called a "variation" of the row.

For quick access, the link from one variation to the next is stored in the record handle. This, plus the transaction ID complete the information stored in a record handle. In order to read a record, a transaction follows the chain until it finds the first record variation that is visible to the transaction. If none is found, then the record does not exist (as far as that particular transaction is concerned).

How much of the past is kept is a system settable parameter. Cleaning up the past is done by the "Sweeper", and this is discussed in the section "Making a Clean Sweep".

TO COMMIT OR NOT TO COMMIT

To commit or not to commit, is actually not the question, far more, how efficiently can it be done? Inevitably, at the end of a transaction the system must commit or abort. The problem with conventional transactional systems is that, either way it often requires quite a bit of work.

For example, if the system caches "before images" (the image of a record before update, used to speed up rollback), then these have to be freed on commit. On rollback, changes to records have to be undone. In general resources, such as locks, need to be freed.

A high commit overhead forces users to batch updates which, from an application point of view, is not always convenient. So the faster this operation is, the better.

The design of PBXT allows the system to perform an instant commit or rollback. To do this, it tracks transactions in a series of transaction list files. Just like the handle files and the table files, transaction files are memory-mapped. Each transaction writes 2 records in the transaction files. The first records is a begin record and the second is an end record.

The transaction ID of the transaction is determined by the begin record. The ID is an 8-byte number. The first 4 bytes is the number of the transaction file, and the second 4 bytes is the index of the begin record in the file.

To complete a transaction PBXT just writes an end record, indicating in the record if the transaction was committed or aborted. The rest of the system reacts instantly to this information. In particular, if a transaction is committed, variations that are already present on the variation chain of a record become instantly visible to other transactions. This is due to the fact that the reader checks the transaction status of each variation as it moves down the variation chain.

Variation belonging to aborted transactions remain invisible until they are eventually cleaned up by the "Sweeper" (see below).

In addition to record type (begin, commit or abort), transactions records contain a timestamp, and a reference to the starting point of the transaction (number and offset of a particular data log).

MAKING A CLEAN SWEEP

In general, the "Sweeper" is a process that cleans up after transactions. So far we have mentioned two functions of the Sweeper:

- **trimming** variations which have reached a certain age (of course, not the last committed variation - no matter how old it is),

- **removing** variations belonging to aborted transactions.

As its name suggests, the Sweeper prepares records for removal by the Garbage Collector. Just like the Garbage Collector it runs in the background, without affecting the user transactions running in the foreground.

It performs its job by marking records as garbage, and incrementing a garbage count (in bytes) in the data log header. In this way the Garbage Collector can determine the garbage ratio be dividing the garbage count by the overall file size. Based on the ratio it decides whether it is time to collect the garbage in the data file.

The Sweeper works by reading the transaction files following on after the oldest terminated transaction. Once a transaction has terminated the Sweeper can begin its work. This is done by reading the transaction as it was written to the data log.

Records that need to be removed (for one of the two reasons mentioned above) are removed by:

- freeing the handles referencing the records,

- removing the records from the variation chain of the table row,

- removing the variation from any indices that reference it (see "Past, Present and Future Indexed" below),

- and marking them as garbage.

If, as a result of this, a table row no longer has any variations in its chain, the row itself is freed.

Once the Sweeper has completed its work on a transaction it is referred to as "clean". A terminated transaction that has not yet been cleaned is said to be "unclean". To avoid the Sweeper conflicting with the Garbage Collector we make the following rule: as long as a data log contains unclean transactions it may not be garbage collected.

Finally, the Sweeper removes a transaction file once it has cleaned all transactions contained in the file.

PAST, PRESENT AND FUTURE INDEXED

As mentioned briefly above, PBXT indexes record variations, not rows. As a result, indexes contain references to:

> **the past:** old variations of records that have been superseded by a newer committed variation,
>
> **the present:** the standard case - the current committed variation of a record,
>
> **the future:** uncommitted variation, only visible to the owner transaction.

So when doing an index scan or search, a transaction could potentially see more than one variation per row. This does not occur because for each transaction, only one variation per record is valid (or none, for example in the case of an uncommitted insert).

When reading an index, each variation found is checked to to see if it is valid in the context of the reading transaction. If it is valid it is returned as usual. If not the index search continues to the next node.

RECOVERY IS GOOD, NO RECOVERY IS BETTER

Conventional transactional engine performs a process know as "recovery" on startup. This is done to guarantee Atomicity and Durability. The process usually works as follows:

The system reads the log from the last checkpoint forward to gather information about transaction that were in progress at the end of the last run. It then reads the log backwards, undoing any transaction that were not committed. After this it reads the log forward again, writing any committed transactions from the database tables.

All this can take quite a bit of time, particularly if the last checkpoint was quite a while ago.

PBXT does not require any recovery. Startup is instant, in the light of the following facts:

- Since data is only written immediately and only to the data logs, as the transaction is running, no additional work is required on startup to ensure that a committed transaction is durable.
- Any data that was not committed is automatically identified as garbage and cleaned up by the Sweeper and the Garbage Collector.

LITTLE ENDIANS AND BIG ENDIANS

These days a user of a database expects to be able to copy a database from one machine to another and be up and running without a second thought. This is only possible if the system commits itself to using one of the two byte endings in use today, either Little Endian or Big Endian.

PBXT stores data on disk in Little Endian format for the following reasons:

- Little Endian based PC hardware is the dominating architecture. Even more, now that Apple has announced that they will be basing future hardware on Intel chips.

- Little Endian chips are based on the CISC architecture. This means that they can access unaligned data directly. A RISC chip has the overhead of accessing packed (unaligned) data byte-wise, so it makes sense to do byte swapping at the same time.

CONCLUSION: THE PROOF IS IN THE EATING

Bearing in mind the fact that there are many existing transactional database engines, the standards (and expectation) are high for any new implementation. In other words, PBXT needs to prove that in practice it performs at least as well, but preferably better, than other transaction engines.

From the results of my initial performance test against two other transactional engines, I can confirm that this is the case. Considering that these engines have been in development for far longer than PBXT leads to the conclusion that the design of the system is largely responsible for the improvement.