The Aurora Storage Manager

Christian Convey Master's Project
April 2003

Abstract

This paper gives an architectural overview of the Aurora storage manager, and details the more interesting issues that arose during its design and implementation.

I assume in this paper that the reader is familiar with data stream management systems in general, and with Aurora in particular.

Introduction

At its heart, the Aurora data stream management system (DSMS) is a data processing system. To be taken seriously, Aurora must quickly process the volumes of data associated with robust RDBMS's. The Aurora Storage Manage (ASM) is designed to enable Aurora to perform in that league.

ASM manages all of the tuples that rest on the arcs, waiting for processing by the next box in the query network.

ASM also manages all of the tuples stored at Connection Points.

Requirements

ASM's initially stated requirements were to provide:

- a tuple-oriented view of data, despite any underlying paging mechanisms employed,
- durable data, available after a planned restart of the system,
- high-performance, low-latency data access,
- support for over 100 GB of concurrently stored data, and
- freedom from the need to pre-calculate a database's total storage size. (I.e., allow dynamic growth of the database.)

Moreover, whatever data access was needed for the ever-changing concept of Connection Points must be provided efficiently.

I make a point of referring these as ASM's *initially stated* goals because as Aurora has evolved, additional data storage needs have appeared. We'll discuss those new needs later in this paper.

This initial version of ASM also has specific *non*-requirements:

- Atomic transactions
- Recoverability after a crash / media failure.

In the original uses foreseen for Aurora, no need was anticipated for recoverability. The assumption was that after a system crash, the data which had been in Aurora would be too stale to be of interest, and could thus be discarded. As of this writing, however, those assumptions are being questioned.

General Design Goals

Storage management in a DSMS can differ significantly from that of a RDBMS. The most obvious difference is that tuples are arranged into queues rather than random-access relations. We'd like to capitalize on that knowledge in our design.

Aurora's scheduler produces explicit execution plans describing when tuples will be enqueued / dequeued from various arcs. Again, we'd like to put this information to use ASM. The hope is that foreknowledge of data accesses will permit unique opportunities for efficient pre-caching and cache-eviction policies in the buffer cache.

Aurora allows various priorities to be assigned to a query network's outputs. This could ultimately lead to the storage manager operations for one queue being prioritized above those that are pending on another queue. If possible, we'd like to take all reasonable steps to ensure ASM's internal processing is sensitive to those relative priorities.

Common wisdom warns against excessive copying every tuple's contents from one

memory location to another. With all of the layers of software in Aurora, there's a temptation to copy tuples' content between layers just to gain a simple interface. We must therefore be vigilant and only copy tuple data when absolutely necessary.

Finally, at any moment ASM may have a large set of pages that must be read from or written to disk. Linux uses an elevator-seeking algorithm that may reward concurrent, rather than sequential, requests for file operations by an application. This is an important design consideration as we try to maximize low-level I/O rates.

Architectural Overview

ASM is organized into three layers of functionality:

- IOLIB
- Buffer Cache
- Queue Management

IOLIB provides a low-level, page-oriented view of the underlying filesystem.

Buffer Cache manages the buffer pool, pre-caches data pages, and lazy-writes dirty pages.

The Queue Management system provides all queue operations and Connection Point operations, as well as admission control. It's the lowest-level system in Aurora that has the concept of tuples and of queues; the Buffer Cache and IOLIB think entirely in terms of pages and frames.

IOLIB

The Unix filesystem API falls short in some significant ways compared to the needs of a DSMS like Aurora:

- File accesses are byte-oriented, even though page-sized operations are known to exhibit better performance.
- On some filesystems no single file may be later than 2 GB.
- In order to have concurrently pending requests for *n* different regions of a file(s), one must have *n* application threads, each concurrently invoking the filesystem API.
- Support for easy experimentation with the performance ramifications of different page sizes.

IOLIB is designed to mask the higher levels of Aurora from these specific problems. Additionally, IOLIB provides internal page management so that most page allocation/free operations avoid translation into filesystem operations.

IOLIB provides the following kind of interface to its users:

- Initialize(max-concurrency, bytes-per-page, page-file-directory-path)
- Operations are requested with a submitOp(...) function, and their results are discovered with a receiveCompletedOp(...) function.
- The operations are: read-page, write-page, allocate-page, free-page

Single-filesystem support

For simplicity, an instance of IOLIB only manages one body of data at a time, all of which is stored in a single filesystem directory. Contrast this to sophisticated RDBMS's that allow different groups of data to appear on different filesystems for reasons such as load balancing and making the best use of high-performance storage.

Catering to elevator-seeking

It seems like a good idea to cater to the Linux kernel's elevator-seeking disk management technique. To do so, an application cannot present all of its read/write operations sequentially to the Unix filesystem API; they must be concurrently pending. However, we'd like to shelter the users of IOLIB from having to create many threads merely to improve I/O throughput.

A performance mistake that we could make when attempting highly concurrent I/O would be to issue an *open(...)* call every time a new page read / write operation needed to be executed. Therefore, some way must be found to re-use file descriptors (FDs) as much as possible.

To accomplish these goals, IOLIB keeps an internal thread pool and an FD pool, and provides an asynchronous I/O (AIO) interface to IOLIB's users. Each thread, when executing an I/O operation, uses one FD. Because of Linux's limit on the number of threads and FDs (1024) a process can have, IOLIB imposes a user-defined limit on the number of concurrent filesystem operations it will support.

This AIO functionality works as follows:

1. The user submits an I/O Control Block (IOCB) which describes the kind of operation desired. The IOCB is placed into a queue of all not-yet-executed IOCBs.
2. Each IOLIB worker thread, upon completing its previous IOCB operation, pulls a new IOCB from the head of the queue and executes it.
3. When an IOLIB worker thread completes its current IOCB-specified operation, it places a return status into the IOCB and puts the IOCN into a response queue that's monitored by the user.

This scheme permits a user to have just one or two threads for interacting with IOLIB, and yet achieve great concurrency of I/O operations.

One feature of the FD pool is worth noting. As we'll see later, IOLIB may have to access many distinct files in the filesystem at nearly the same time. However, even when using multiple files, IOLIB must not violate its constraint on the total number of FDs it may have open at one time.

In order to support multiple data files without resorting to frequent open(...) / close(...) system calls, the FD pool uses the following logic. For each FD that's in the pool but not presently leased out, the pool remembers which particular file the FD is tied to. When a lease-request is made on the FD pool, the pool first tries to provide a FD that's already open on the file specified in the request. If no such FD is available, the pool will close one of the available FDs, open(...) a new one on the specified file, and return it to the caller.

File structure

As the set of pages stored by IOLIB grows, a linear-fill strategy is applied to file growth as follows.

IOLIB presumes that the underlying filesystem doesn't support files larger than 2 GB. As storage needs grow, IOLIB will grow its current file until approximately the 2 GB mark. Once additional growth is needed, a new file will be created, and that file in turn will experience the same growth pattern that older files did.

For example, if IOLIB is managing 13.2 GB of data, there will be 7 data files: 6 files that are 2 GB in size, and one file (the most recently created one) that's 1.2 GB in size.

The internal structure of an IOLIB data file is a function of the page size, but the same basic approach is always employed.

Each file contains a sequence of allocation regions (ARs). The first page in every AR is a bitmap giving the allocation status of all the other pages in that AR. One bitmap bit is used for each page, simply describing whether or not that page is currently in use by the application. Unused pages are available for allocation.

An AR contains as many pages as its bitmap can describe. For instance, if page sizes are 64KB, then the bitmap page will have 65,536 bytes, which is 524,288 bits. That means in a 64KB-page system, each AR will contain 524,289 pages (1 bitmap page + the 524,288 data pages whose statuses are recorded in the bitmap page).

Two factors may prevent the currently last AR in a file from being as large as its bitmap's potential.

If the AR size doesn't evenly divide into 2GB, the final AR in a full sized file is necessarily smaller than the other ARs in the file.

Secondly, files aren't necessarily grown in integer multiples of AR sizes, as we'll discuss later. Therefore, the current length of a file may limit the number of data pages contained in that file's last AR.

Page allocations and file growth

IOLIB keeps all of the AR allocation bitmaps in virtual memory at all times, for simplicity. When a page allocation request is made, the bitmaps are scanned for an available page. If an available page is found, its bitmap bit is set and the page's logical address is returned. If a no page in the entire system is available, the allocation request will be held up pending file growth.

IOLIB has a high-water mark for triggering file growth. For example, if the high-water mark was 90% and IOLIB had a disk footprint of 20GB, then file growth would be triggered when 18GB were actually allocated by the application.

The following describes how an individual file is grown. File growth is not done in constant-sized chunks. In most cases, it's done as a proportion of the current file size. For example, if the growth ratio is set to 10%, each attempt to grow a file would be by 10% of the file's current size.

This growth ratio is bounded, however. In order to prevent many distinct growths on a small database, a constant lower limit can be established for file growth sizes. Any file growth attempt whose ratio would lead to a smaller growth than this lower limit, is automatically adjusted to grow the file at the lower limit. At the upper bound, each file growth attempt is reduced, if necessary, to prevent the file from growing larger than 2 GB.

If file growth is needed but the last file in the file set is already at 2 GB, a new file must be allocated. Each new file starts out with a modest size, without any attempt to immediately make the file large enough to satisfy the IOLIB's high-water mark.

Because IOLIB uses linear fill on files, there's no obvious reason to support the concept of concurrent growth operations; there's only one "end" of the file set that can be grown. Therefore, IOLIB permits at most one file growth operation at a time.

Open issues and future work for IOLIB

Two features of the Linux 2.6 kernel are expected to obviate much of IOLIB's code. One feature is large file support in many filesystems, and the other is native support for the Posix Asynchronous I/O standard. These two features would let IOLIB use merely one file, and would take away the need for IOLIB's thread pool and FD pool.

Another open issue is that IOLIB has a weakness in the way page allocation / page free requests are made. The can normally be satisfied extremely quickly, because an internal bitmap consultation is all that's needed in most cases. However, in such cases we still require the user to use the AIO machinery: formulate an IOCB for a page allocation request, submit the IOCB, and await the response. IOLIB would be simpler to use if page allocations/frees were simply blocking function calls.

Another issue is whether or not IOLIB should be extended to let each IOCB be tagged by the user with a priority level. If we had more outstanding IOCB requests than we could handle, then some IOCBs would have to wait for execution. We could ensure that IOCBs awaiting execution were fully executed according to priority order, rather than in first-come-first-serve order.

Finally, we have the issue of disk location for pages. Experience teaches us that great performance benefits are found when pages that are accessed around the same time also happen to be near each other on disk.

However, the standard Unix filesystem API gives no explicit control over disk location. Some filesystem-specific APIs, such as SGI's XFS, provide a measure of control in this regard, but not all Linux systems support XFS. The only sure way of achieving specific data placement on disk is to have the IOLIB store data on a raw partition. The open question is whether or not such steps are worthwhile.

Buffer Cache

ASM's buffer cache is different than traditional demand-paging buffer caches. Its users don't simple request pages and await their framing. Instead, the users maintain a table of page priorities, which the buffer cache consults to perform pre-caching and during cache evictions. The user knows that as long as certain restrictions are adhered to, any page

whose user priority is set to 10 will eventually become framed.

The Aurora scheduler can plan ahead its page accesses by tens or hundreds of pages at a time. The buffer cache's unusual design was created to capitalize on this special foreknowledge held by the scheduler.

The public interface provided by the buffer cache is conceptually as follows:

- To allocate or free a page, the user directly operates on a pool of available pages.
- For any page of interest: The user can set the caching priority between 0 and 10, indicating how much the user wants the page to be framed. Once framed, a page with priority 10 will remain framed until the user lowers its priority. (The user is responsible for not setting more pages to priority 10 than there are frames.)
- A user can wait for a particular page to be framed, or for an already-framed page to be lockable by the user.
- For any framed page: lock-for-read, lock-for-write, mark-dirty, and unlock the page.

An application that wanted to allocate a new page and write to it would take the following steps:

1. Acquire a new page address out of the available-pages pool. (Block if the pool is currently empty.)
2. Set the page's priority to 10, so that it will definitely be framed eventually. Await the page's framing.
3. Wait until a write-lock on that page can be granted to the user. (The buffer cache's lazy writer may compete with the user over a write lock on the page.)
4. Modify the contents of the page's frame buffer. Mark the page as dirty, and relinquish the write lock.

Using this interface can be much more complicated than using a traditional demand-paging system. This complexity has proven to be one of this buffer cache's biggest drawbacks.

Pages, Frames, and Priorities

Every framed page in the database has two different priorities: A *user priority*, and an *effective priority*. (For space efficiency, a user priority may remain unstated and is then presumed to be 0.)

The effective priority of a frame is a function of (a) the user priority of the page occupying the frame, and (b) the lock status and I/O status of the frame. Differentiating between user priority and effective priority lets us represent a temporary boost of a page's priority, even beyond what the user requested. While a page's user priority is in the range [0, 10], a page's effective priority is in the range [0, 11].

In order to keep the pre-caching / page-eviction logic clean, the buffer cache applies two simple rules:

- A page with a higher effective priority always gets preferential framing over a page with lower effective priority.
- A page currently holding a frame will not be evicted to make room for another, not-yet-framed page with the same effective priority.

These rules provide a somewhat clean way for the buffer cache to offer pinning when necessary.

In order for a page frame to be pinned, its effective priority must be 11. The following frame states will cause a frame's effective priority to be 11:

- The user holds a read-lock or a write-lock on the page frame.
- The page frame is being lazy-written to disk.
- The page frame is currently being populated from disk.

A page that isn't pinned will just have an effective priority equal to its current user priority. With this scheme, all page frames that meet any of the three pin-causing conditions will have a higher effective priority than any page frame that lacks all those conditions. This mechanism is how we prevent the eviction of pinned frames.

Internal Design

Three main data structures in the buffer cache contain information about pages and/or frames:

- Available Page Set
- Frameless Pages Map
- Framed Pages Map

Available Page Set is a cache of available pages. This cache exists because acquiring a new page from IOLIB can involve a lot of latency. When a user asks the buffer cache to allocate a new page, the page is drawn out of this set. If the cache is getting too low on pages, requests are issued to IOLIB for additional pages. Then IOLIB satisfies those page-allocation requests, the resulting pages are placed into the Available Page Set.

Frameless Pages Map tracks the user priorities for pages that the user has explicitly set the priority on, but which aren't currently framed. In the simplest sense, this map just remembers, per page, the user priority that the user has specified.

This map also has logic for quickly finding *n* pages in the map that have the highest user priority. This is helpful when frames become available, and the buffer cache must quickly select the pages that are to be granted frames.

Framed Page Map has one entry for each framed page. It carries information regarding the page and the state of the frame. Each entry in this map records: Page address, user priority, effective priority, locking status, and I/O status.

The buffer cache uses a single dedicated thread to perform pre-caching, lazy-writing, and to handle all results from IOLIB operations. A single thread is able to accomplish all these tasks because of IOLIB's asynchronous interface. This seems to validate the choice of using AIO at the IOLIB level. It lets higher levels avoid the complexity of heavy multithreading.

Engineering Challenges

The buffer cache's complex interface proved very difficult to implement and to use. While the priority-based map still seems sound conceptually, it's unclear whether or not it can provide enough performance boost (or any boost for that matter) to warrant its code complexity.

Another issue was specifically a result of the buffer cache not offering demand paging. When a user's thread wants to access or lock a page, the buffer cache must provide an efficient way to sleep that thread until the page is in the desired state

To implement this, the buffer cache keeps a map of {page addresses → signal semaphores}. Any user thread that wants to await the framing, or the lockability, of a page puts a n entry into this map. The user thread then waits on the semaphore, and the buffer cache will signal it when the page reaches the desired state.

This machinery seems overly complex, and an open question exists regarding if/how this mechanism can be improved.

Finally, using many different data structures to track the states of pages and frames may contribute to the general CPU-boundness of ASM. ASM relies on C++'s Standard Template Library extensively for providing maps, sets, etc. Unfortunately, the C++ STL can lead to a programmer carelessly create too many container structures, because it takes so little programming time to do so. One must wonder exactly how much performance loss is attributed to my willingness to instantiate maps, sets, etc. whenever it was convenient.

Queue Management

The ultimate goal of ASM is to present queues of tuples, and Connection Points. The Queue management is the layer that offers that interface.

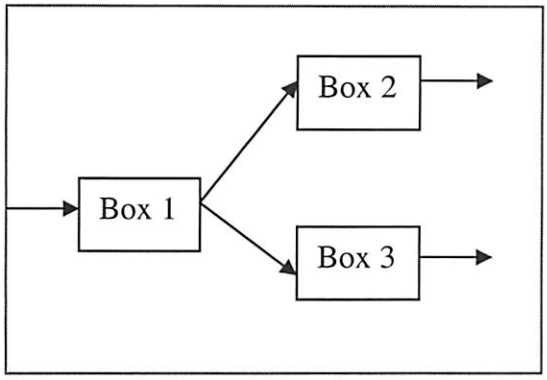
While the lower implementation layers in ASM work in terms of pages, frames, and files, the Queue managements builds on those to provide the language of tuples, QBases, QViews, Connection Points, and Superbox Plans.

QBases and **QViews** represent the two different ends of a tuple-containing queue. Tuples are inserted into QBases and removed from QViews.

It seems natural to use a single object, perhaps called a Queue, to represent the data structure

just described, rather than two objects, a QBase and a QView. We'll now discuss the motivation and the power of the QBase / QView scheme.

In an Aurora flow diagram, queues are drawn from the output(s) of one box to the input(s) of one (or more) boxes. One possible configuration, the one that motivates the QBase/QView model, is shown in below.



The meaning of a flow diagram like the one above is that any tuple placed onto the Box1→Box2 arc, will also have a copy placed on the Box1→Box3 arc.

A naïve implementation of this topology would actually duplicate the storage of each tuple leaving Box 1. One copy would be placed into the backing store of the Box1→Box2 arc, and the other copy would be placed in a distinct backing store location for the Box1→Box3 arc. Such an implementation seems wasteful, because we *know* that the two arcs pass the same data. The QBase / QView model is how we address this scenario.

When a QView is created, it's indelibly associated with some particular QBase. In our example, a QBase would be defined at the output of Box1, and two QViews would be defined: one at the input of Box2, and the other at the input of Box3.

A QBase manages the backing storage associated with the tuples inserted into it. It keeps the stream of still-interesting tuples in a sequence of pages, where a given page will hold as many tuples as it can fit. The QViews simply store pointers into that page sequence.

When a tuple is removed from a QView, the QView's pointer into the page sequence is advanced by as many tuples as were removed. Once all of the QViews have "removed" the tuples on one of the pages, the page is no longer needed by the QBase. At that point, the QBase can deallocate the defunct page.

(Actually, that's a simplification. A QBase keeps a *circular queue* of pages, to avoid excessive page allocations and frees. A page can be freed when it's empty, but its appearance in a circular queue means it will likely get reused and never actually become empty. As of this writing, the functionality for returning an empty page to the free-page store hasn't been implemented.)

Any QBase can be a **Connection Point**. When a QBase is a Connection Point, it can retain tuples that are no longer of interest to any of the currently attached QViews. This is to permit other QViews, attached at some *later time*, to have access to the history of the stream of tuples that passed through the QBase.

Each Connection Point has a *history specification* defining how far back into history the QBase retains old tuples. The history specification is in terms of maximum tuple age (elapsed wall-clock time since inserted), the number of tuples that the queue will hold at once, or both.

To scan the history stored in a Connection Point, the user attaches a new QView to that Connection Point. In the act of attaching, he specifies how much history is to be replayed through that QView. The user can specify how many tuples to return in terms of the maximum number of tuples, the maximum age of tuples, or both.

A QView that has been attached to a Connection Point will present the historical tuples in the same sequence in which they were inserted. When all of those historical tuples have been removed from the QView, the QView will proceed to present all newly inserted tuples. That is to say, a QView attached to a Connection Point smoothly transitions from presenting historical data to presenting live data.

Once a page of tuples is no longer needed according to this history specification, *and* now currently attached QView still needs to present tuples on that page, the page can finally be freed from the QBase. So in general,

a Connection Point might hold onto data pages longer than a non-Connection Point QBase would have.

The Queue management requires the Aurora scheduler to provide it all active **execution plans**. In Aurora's runtime, there are many *worker threads*, each of which is single-handedly responsible for processing a sequence of flow diagram boxes. Since there are many worker threads in the runtime, there can be many concurrently active execution plans.

There are two reasons that these execution plans must be pre-declared to ASM: admission control, and page prioritization.

Admission control is necessary to prevent excessive competition for page frames. Every box that runs, in order to have the necessary data access, will concurrently pin one or twp pages from *every arc* going into or out of that box. If we allowed an arbitrary number of boxes to run at the same time, they could collectively need to pin more pages at one time than there are page frames. Avoiding pinning deadlock in such a situation would be difficult.

To solve this problem, ASM actually controls when a given worker thread may proceed to process the next box in its execution plan. By doing so ASM ensures that there are enough page frames for all the worker threads' needs.

One might think that such throttling of the worker threads would best be left to the scheduler, not ASM. However, only ASM has all of the information needed to know exactly how many frames are available at a particular moment. This is because dirty frames aren't available for re-use until the buffer cache's lazy writer commits the frame to disk. That information should, and does, remain private to ASM.

Recall that for each execution plan, at most one step is currently active. Each execution plan step is associated with a single box that can operate on two or more queues. For each execution plan step we can know which will be accessed and whether the accesses will be insertions or removals.

ASM's elaborate **page prioritization** scheme is the other reason that ASM needs foreknowledge of the scheduler's execution plans. It uses the following heuristics to estimate the global importance of any particular page being framed.

- The pages associated with currently active execution plan steps deserve frames more than do pages associated with not-yet-active execution plan steps.
- Some of the pages in a queue are closer to the insert point / removal points than other pages in the queue. We use the knowledge of which pages are closest to those insert / remove points, along with the knowledge of what insertions / removals a particular execution plan step is likely to perform, to give varying priorities to the pages.

Finally, the Queue management makes some statistics and metadata available to its users. These include internal performance measurements, the total number of tuples in a QView, the number of tuples in a QView that can be immediately removed without a page fault, and the average timestamp of all tuples currently in a QView.

Open Issues and Future Work for Queue Management

The **QBase / QView model** has been a mixed blessing. On the one hand, it allowed a very easy implementation of the Connection Point page-retention logic. Pages are kept in a QBase's page ring whenever a QView hasn't yet presented that page's data. A Connection Point just has an extra, internal QView that's properly positioned to force the QBase to retain all of the appropriate historical tuples.

On the other hand, people find the QBase / QView model confusing. The implementation was also very complicated; I spent over a week getting the page-ring logic correct for the case of multiple QViews.

The ultimate justification for the QBase / QView model would be the existence of flow networks, which solve real problems, containing a lot of boxes whose outputs are sent to multiple destinations. Whether or not this will happen remains to be seen.

Performance is another problem. During informal profiling of an ASM test program, I

found that 70% of the CPU time was spent recalculating page priorities.

While the page prioritization scheme in ASM is interesting, it may just be too computationally expensive to ever justify itself. If nothing else, ways must be found to reduce the costs of this scheme.

Another possibility is that we should simply ignore the tantalizing information we have on future probable page-access-order. Perhaps demand-paging with LRU or MRU eviction policies would serve us well. This is a very important research topic for ASM's future viability. If ASM doesn't improve in this area, Aurora has little chance of performing acceptably.

Atomic operations may be necessary if Aurora is every to be recoverable after a crash. One difficulty, though, is deciding exactly what the granularity of atomicity ought to be.

For the future, we're considering an alternative representation for storing a Connection Point's purely historical data. Instead of simply having Connection Points hold on to data pages longer than their non-Connection Point counterparts, we may migrate such historical information into a relational database, such as **Berkeley DB**. The motivation is the hope that Connection Points could more efficiently offer selections and/or projections of the historical tuples in a way not currently possible.

Finally, in the future ASM may need to provide **per-box state storage**. Various box types may need significant amounts of durable private state. That state information only needs to be in main memory while its associated box executes. Since ASM already has a paging system in place, it may be the natural mechanism for storing box states.

State of Integration

Aurora still uses its old prototype storage manager. This is largely because we've lacked the time to integrate this new ASM. We hope to begin the integration in late April of 2003.

The integration will be somewhat complicated by the different services provided by the two storage managers

Acknowledgements

Few of the ideas for ASM came only from me. Many of the ideas were originated and improved by the various people attending Aurora meetings over the past few years. Thanks to all.