Modularity and Scalability in Calvin

Alexander Thomson
Google[‡]
agt@google.com

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Abstract

Calvin is a transaction scheduling and replication management layer for distributed storage systems. By first writing transaction requests to a durable, replicated log, and then using a concurrency control mechanism that emulates a deterministic serial execution of the log's transaction requests, Calvin supports strongly consistent replication and fully ACID distributed transactions while incurring significantly lower inter-partition transaction coordination costs than traditional distributed database systems.

Furthermore, Calvin's declarative specification of target concurrency-control behavior allows system components to avoid interacting with actual transaction scheduling mechanisms—whereas in traditional DBMSs, the analogous components often have to explicitly observe concurrency control modules' (highly nondeterministic) procedural behaviors in order to function correctly.

1 Introduction

The task of building parallel and distributed systems designed for *embarrassingly parallel* computational problems is relatively straightforward. Data-parallel tasks with reasonably sparse and coarse-grained dependencies between computational tasks—such as many analytics applications—fall nicely into this category. In particular, parallelizing a workload across many machines is easiest when the amount of real-time coordination required between machines is lowest.

The general problem of online transaction processing, however, does *not* fall into this category. OLTP systems generally guarantee isolation between transactions using row-level locks, and any transaction that incurs synchronous communication between multiple machines (e.g. by performing a remote read or running a distributed commit protocol) must hold its locks for the full duration of this communication. As more transactions span multiple machines, lock contention becomes a bigger barrier to achieving high transactional throughput.

For systems that synchronously replicate transaction effects at commit time via Paxos [11] (e.g. Spanner, Megastore, Spinnaker), the problem is further exacerbated, since transactions' contention footprints[1] extend for the full Paxos replication latency [8, 1, 14].

[*]This material is supported by the National Science Foundation under Grant Number IIS-1249722.

[‡]Work done while the author was at Yale.

[1]The "contention footprint" of a transaction is the period of time during which it may limit concurrent execution of conflicting transactions. For systems that use locking, this corresponds directly to the time spent holding locks. In optimistic systems like Megastore, it is the time from when the transaction begins executing until the validation phase is complete, since other commits during this time window may cause the transaction to abort.

One approach to overcoming this limitation is to reduce transactional support. For example, Microsoft's SQL Azure [4], Google's Bigtable [6], and Oracle's NoSQL Database [17] limit each transaction to accessing records within a single small data partition. For certain "embarrassingly partitionable" OLTP applications, solutions that provide limited transactional support are useful. For many applications, however, this approach is a poor fit: if a transaction ever spans data on multiple partitions, the developer is forced to split it into multiple separate transactions—and therefore inherits the burden of reasoning about data placement, atomicity, and isolation.

Calvin takes a different approach to ameliorating this scalability bottleneck: it reduces the amount of coordination required while executing distributed transactions. Calvin rearranges the transaction execution pipeline so that all distributed agreement that is not directly data-dependent (i.e. all inter-partition communication except for serving remote reads) occurs in efficient batch operations *before* transactions begin acquiring locks and executing.

Specifically, Calvin divides the transaction execution pipeline into three stages—logging, scheduling, and execution. When a client submits a request to Calvin to execute a transaction, this transaction *request* (as opposed to ARIES-style physical actions) is immediately appended to a durable log, before any actual execution begins. Calvin's scheduling mechanism then processes this request log, deciding when each transaction should be executed in a way that maintains an invariant slightly stronger than serializable isolation: transaction execution may be parallelized but must be equivalent to a serial execution (a) in the order that transaction requests were logged and (b) that does not non-deterministically cause transactions to abort. Then, once a transaction begins executing, it is disallowed from performing any non-deterministic behaviors[2].

Calvin's processing of a series of transaction requests is therefore equivalent to a deterministic serial execution in log-specified order. As a result, the log of transaction requests itself serves as an ultimate "source of truth" about the state of a database. Even if all other information about a the database were discarded, its state at any point in history could be recovered by deterministically replaying the operation log[3].

This architecture provides several advantages over traditional (nondeterministic) database system designs:

- **Near-linear scalability.** Since all distributed agreement (both between replicas and between partitions within a replica) needed to commit a transaction occurs *before* Calvin executes the transaction, no distributed commit protocols are required, ameliorating the primary barrier to scaling ACID systems. We have demonstrated this scalability with a Calvin deployment that executed half a million TPC-C transactions per second[4] while partitioning data across one hundred commodity EC2 machines [21].
- **Strongly consistent replication.** Multiple replicas of a Calvin deployment remain perfectly consistent as long as they use the same operation request log to drive transaction execution. Once a transaction is written to the log, no further inter-replica coordination is required to synchronize replicas' states. This allows strongly consistent replication—even across geographically separated data centers—at no cost to transaction throughput [21].
- **Highly modular system architecture.** By explicitly describing the emulated serial execution in the form of the operation request log, Calvin provides other system components with a *declarative* specification of the concurrency control mechanisms' behavior. Components that traditionally interact closely with the transaction scheduling system in standard nondeterministic database systems (e.g. logging, replication, storage) are thus completely decoupled from Calvin's concurrency control implementation. Standard interfaces for each Calvin component make it easy to swap in different implementations of each component to quickly prototype systems with different properties.
- **Main-memory OLTP performance over disk-resident data.** When Calvin begins processing a transaction request by adding it to the log, it may also perform a cursory analysis of the transaction request and send a

[2]Calls to functions such as GetTime() and Random() are intercepted by Calvin, and replaced by deterministic functions that use the time at which the client submitted the transaction, rather than the local system time of the server executing the transaction.

[3]Calvin uses this property (along with frequent checkpointing) for durability and disaster recovery.

[4]Note that TPC-C New Order transactions frequently access records spanning multiple data partitions *and* incur high lock contention.

prefetch "hint" to each storage backend component that the transaction will access once it begins executing. The storage engine can therefore begin bringing the transaction's read- and write-sets into memory even before the request is appended to the log, so that all relevant data will be memory-resident once the transaction begins executing.

2 Calvin's Modular Implementation

Database management systems are notoriously monolithic pieces of software, largely because they use complex concurrency control mechanisms to orchestrate the simultaneous execution of many transactions [9]. Many attempts have been made—with varying success—to build clean interfaces between various components, decoupling transaction coordination, buffer pool management, logging/recovery mechanisms, data storage structures, replication coordination, query optimization, and other processes from one another [2, 5, 7, 13, 12, 16].

We believe that the fundamental difficulties in unbundling database components lies in the way concurrency control protocols are traditionally described. Besides being highly non-deterministic, concurrency control algorithms are usually framed (and specified and implemented) in a very *procedural* way. This means that system components must often explicitly observe internal state of the concurrency control module to interact with it correctly. These internal dependencies (particularly for logging and recovery) become extremely apparent in modular systems that are otherwise successful at separating database system components [13, 16].

Whereas serializable isolation requires equivalence to *some* serial execution of transactions (but is agnostic as to what particular order is emulated), Calvin's determinism invariant restricts concurrency control mechanisms to maintain equivalence to a specific, pre-determined order. This design decision was originally motivated by the need to reduce coordination costs between participants in a transaction, but we found that establishing an a priori transaction ordering is also useful as a *declarative* specification of concurrency control behavior.

In Calvin, database system components that traditionally interact closely with the concurrency control manager can instead gain the same information simply by reading from the (immutable) transaction request log.

Thus, in addition to reducing coordination costs between instantiations of the same component on different machines, Calvin's log simplified the coordination between components. This section gives an overview of the three independent parts of every Calvin deployment—the log, scheduler, and storage backend—and their individual interfaces and subcomponents.

2.1 Log

A Calvin deployment appends new transaction *requests* to a global log, which is then readable by other system components. Implementations of the log interface may vary from very simple to quite sophisticated:

- **Local log.** The simplest possible implementation of a log component appends new transaction requests to a logfile on the local filesystem to which other components have read-only access. This implementation is neither fault-tolerant nor scalable, but it works well for single-server storage system deployments.
- **Sharded, Paxos-replicated log.** At the opposite extreme from the local log is a scalable, fault-tolerant log implementation that we designed for extremely high throughput and strongly consistent WAN replication. Here, a group of front-end servers collect client requests into batches. Each batch is assigned a globally unique ID and written to an independent, asynchronously replicated block storage service such as Voldemort or Cassandra [19, 10]. Once a batch of transaction requests is durable on enough replicas, its GUID is appended to a Paxos "MetaLog". To achieve essentially unbounded throughput scalability, Paxos proposers may also group many request batch GUIDs into each proposal. Readers can then reassemble the global transaction request log by concatenating batches in the order that their GUIDs appear in the Paxos MetaLog.

2.1.1 Log-forwarding

Distributed log implementations may (a) forward the read- and write-sets of new (not-yet committed) transaction requests to the storage backends that they will access (so that the data can be prefetched into memory *before* transaction execution begins) and (b) analyze a committed transaction request and, if it can determine which schedulers will need to process the request, actively forwards the request to those specific schedulers (otherwise it forwards it to all schedulers). Our implementation provides a subcomponent that can be plugged into any actual log implementation to automatically handle these forwarding tasks.

2.1.2 Recovery Manager

Recovery managers in database systems are notoriously cross-dependent with concurrency control managers and data storage backends. Popular recovery protocols such as ARIES log every change to index data structures (including, for example, the internal details of each B+ tree page split), which means that they must understand what inconsistent states could arise in every index data structure update. Furthermore, recovery managers commonly rely on direct knowledge of record and page identifiers in the storage layer in order to generate log records, and may store their own data structures (e.g. LSNs) inside the data pages themselves.

Calvin leverages its deterministic execution invariant to replace physical logging with logical logging. Recovery is performed by loading database state from a recent checkpoint, then deterministically replaying all transactions in the log after this point, which will bring the recovering machine to the same state as any non-crashed replica. Calvin's recovery manager is therefore very simple and entirely agnostic to implementation details of the log, scheduler, and storage backend—so long as they respect Calvin's determinism invariant.

2.2 Separating Transaction Scheduling from Data Storage

In traditional database systems, actively executing transactions request locks immediately before each read and write operation. There are thus direct calls to the concurrency control code from within the access methods and storage manager. Calvin's concurrency control mechanisms diverge from this design: a Scheduler component examines a transaction before it begins executing and decides when it is safe to execute the *whole* transaction, then hands the transaction request off to the storage backend for execution with no additional oversight. The storage backend therefore does not need to have any knowledge of the concurrency control mechanism or implementation. Once a transaction begins running, the storage backend can be sure that it can process it to completion without worrying about concurrent transactions accessing the same data.

Since the storage manager does not make any direct calls to the concurrency control manager, it becomes straightforward to plug any data store implementation into Calvin, from simple key-value stores to more sophisticated NoSQL stores such as LevelDB, to full relational stores (Section 4 describes some of the storage backends that we have implemented over the course of creating various Calvin configurations). However, each storage backend must provide enough information prior to transactional execution in order for the scheduler to make a well-informed decision about when each transaction can safely (from a concurrency control perspective) execute. This is done by building a wrapper around every storage backend that implements two methods:

- **`RUN(action)`** receives a command in a language that the data store understands (e.g. `get(k)` or `put(k,v)` for a key-value store, or a SQL command for a relational store) and executes it on the underlying data store.
- **`ANALYZE(action)`** takes the same command, but instead of executing it, returns its *read- and write-sets*—collections of logical objects that will be read or created/updated when `RUN` is called on the command. These objects can be individual entities (such as key-value pairs or relational tuples) or ranges of entities. The scheduler leverages the `ANALYZE` method to determine which transactions can be run concurrently by the storage backend. In some cases, the `ANALYZE` method can derive the read and write set of the command using static analysis. In other cases, it may have to perform a 'dry run' of the command (at only one replica,

with no isolation) to see what data items it accesses. Then, when the same command is eventually `RUN`, the storage backend must double-check that the initial observation of what would be read or written to was still accurate. (Since the `RUN` method is called in deterministic execution mode, all replicas will perform this same double-check step and arrive at the same result.)

This wrapper around the storage backend enables the clean separation of transaction scheduling from storage management. This approach deviates significantly from previous attempts to separate database systems' transactional components from their data components [13, 12, 16] in that Calvin does not separate transaction *execution* from the data component at all—it only separates transaction *scheduling*.

Just as the storage backend is pluggable in Calvin, so is the transaction scheduler. Our prototype currently supports three different schedulers that use transactions' read-/write-sets in different ways:

- **Serial execution à la H-Store/VoltDB [18].** This scheduler simply ignores the read and write set information about each transaction and never allows two transactions to be sent to the storage backend at the same time. However, if data is partitioned across different storage backends, it allows different transactions to be sent to different storage backends at the same time.
- **Deterministic locking.** This scheduler uses a standard lock manager to lock the logical entities that are received from calling the ANALYZE method. If the lock manager determines that the logical entities of two transactions do not conflict, they can by executed by the storage backend concurrently. In order to avoid nondeterministic behavior, the ANALYZE method is called for each transaction in the order that the transactions appear in the transaction log, and the lock manager locks entities in this order. This ensures concurrent execution equivalent to the serial transaction order in the log, and also makes deadlock impossible (and the associated nondeterministic transaction aborts) [20, 21].
- **Very lightweight locking (VLL).** VLL is similar to the deterministic locking scheduler described above, in that it requests locks in the order of the transactional log, and is therefore deterministic and deadlock free. However, it uses per-data-item semaphores instead of a hash table in order to implement the lock manager. We originally proposed storing these semaphores alongside the data items that they control [15] for improved cache locality. However, this violates our modular implementation goal, so Calvin supports two versions of VLL—one which co-locates semaphores with data, and a more modular one in which the scheduler tracks all semaphores in a separate data structure.

Despite major differences between the internals of these transaction scheduling protocols, each implements a common interface and adheres to Calvin's deterministic scheduling invariant, so switching between them is as simple as changing a command line flag with which the Calvin binary is invoked. A single Calvin deployment could even use different schedulers for different replicas or even partitions within the same replica.

3 Transactional Interface

Clients of a Calvin system deployment may invoke transactions using three distinct mechanisms:

- **Built-in Stored Procedures.** Any storage backend API call can be invoked as a stored procedure with transactional semantics. For example, we implemented the TPC-C transactions that we benchmarked in [21, 15] as built-in API calls to a TPC-C-specific storage backend implementation. This is the transaction execution method that yields the best performance. However, while we made every effort to make it easy to extend or re-implement Calvin's storage backend components, it is by far the most development-intensive way for an application to run complex multi-operation transactions.
- **Lua Snippet Transactions.** Alternatively Clients can submit transactions that bundle together multiple storage API calls using client-specified logic as implemented in Lua code. The client simply constructs a Lua code snippet and sends it as a string to Calvin, which performs some analysis to make sure the code is well-behaved. Thereafter, it can be scheduled and executed as if it were a single stored procedure.

- **Optimistic Client-side Transactions.** Calvin also allows client-side transaction execution in which an OCC validation phase is executed as a deterministic stored procedure at commit time. Here, a client application starts a transaction session, performs any set of storage API calls (writes are buffered locally), and then submits a commit request for the session. Upon receiving the commit request, Calvin checks version timestamps for each record read and written by the transaction. If all of them precede the timestamp at which the client's session started, then this validation phase succeeds and the buffered writes are applied—otherwise the transaction "aborts"[5] and the client must start it again using a new session.

We expect that most application developers will choose to mostly use optimistic client-side transactions while building new applications. Specific performance-sensitive transactions can then be replaced with Lua snippets or built-in stored procedures as needed.

4 Beyond Main-Memory OLTP

Although we originally developed Calvin as a transaction scheduling and replication layer explicitly for main-memory OLTP database systems—and with specific log, scheduler, and storage implementations in mind—we have found it very easy to create Calvin configurations with a broad array of characteristics simply by swapping out implementations of various components.

4.1 CalvinDB

The distributed OLTP database system described in [21]—which we now refer to as CalvinDB—provided a basic CRUD interface to a memory-resident key-value store. Our early prefetching experiments used a very rudimentary disk-based backend that performed poorly. Newer versions are backed by a LevelDB-based store and provide stronger prefetching support, so Calvin can reliably process transactional workloads that sometimes access disk-resident data with main-memory-only throughput performance.

In addition, we have extended CalvinDB's interface to support secondary indexes and a subset of SQL for declarative data manipulation[6]. We found that Calvin's deterministic semantics makes it surprisingly painless to implement atomic database *schema* changes with no system or application downtime whatsoever—a challenge that is frequently a source of headache for architects of distributed database systems.

4.2 CalvinFS

Scalable metadata management and WAN replication for distributed file systems are hot research topics in the systems community due to the ever-increasing scalability and availability requirements of modern applications.

We therefore created CalvinFS, a distributed file system that supports strongly consistent WAN replication. CalvinFS stores data blocks that make up file contents in an easy-to-scale, non-transactional block store, and uses Calvin for transactional storage of file *metadata*. This turned out to be an extremely good fit for Calvin, because real-time metadata updates for distributed file systems in many ways resembles a *not*-embarrassingly-partitionable OLTP workload in two important ways:

- File systems traditionally support linearizable updates—which strongly resemble ACID transactions—and recent snapshot reads.
- Although most operations involve updating metadata for a single file (e.g. reading, writing to, or appending to a file), certain operations (such as creating and deleting files) involve atomically updating metadata entries

[5]Note that since this OCC validation phase is itself logged and executed as a standard built-in transaction, its outcome is entirely deterministic and proceeds identically at every system replica.

[6]Specifically, Calvin currently requires SQL statements that perform updates to identify all updated rows by primary key, and queries' WHERE clauses are restricted to predicates on columns on which a secondary index is maintained.

for both the file and its parent directory. Since these entries may not be stored on the same metadata server, these represent distributed transaction of exactly the type that Calvin processes very efficiently.

We demonstrated that CalvinFS can scale to billions of files and process hundreds of thousands of updates and millions of reads per second while maintaining consistently low read latencies. Like CalvinDB, it can survive entire data center failures with only small performance hiccups and no loss of availability.

4.3 Calvin in Costume: Mimicking Hyder and H-Store

To further demonstrate the flexibility of Calvin's modular implementation, we created configurations that duplicate the transaction execution protocols, logging mechanisms, and data storage structures of Hyder [3] and H-Store [18]. Both configurations involved modifying or extending only a few of Calvin's components; a surprisingly small amount of additional code was required to reproduce these two distinct architectures.

4.3.1 Calyder

Hyder is a transaction processing system for shared flash storage. In Hyder, an **executor** server runs each transaction based on a current snapshot view of the database, then adds an after-image of the transaction's updates (called an **intention**) to a distributed flash-resident log. Intentions are then applied to the database state by a deterministic **meld** process that performs optimistic validation checks, applying any transaction whose read-/write-sets have not been updated since the snapshot that the executor used to execute the transaction [3].

We created Calyder, a Calvin instantiation that uses a shared-flash log implementation based on Hyder's, and where a simple "executor" component at each Calvin server accepts transaction requests from clients and executes them using our optimistic client-side transaction mechanism (in which the "executor" acts as the client). Each scheduler then deterministically applies optimistic validation checks, exactly like Hyder's meld procedure.

Although our flash log implementation was somewhat simpler and slower than Hyder's, and our data storage mechanism involved additional copying of inserted record values, we were able to recreate performance measurements that resembled Hyder's, but with slightly higher transaction abort rates due to our implementation's somewhat longer delays between the execution and meld (optimistic validation) phases.

4.3.2 H-Cal

H-Store is a main-memory OLTP database system that executes transactions serially using a single thread per partition, eschewing all logging, locking, and latching and relying on in-memory replication for durability [18].

We created H-Cal, which uses a "fake" log component implementation in that doesn't actually record any durable state, but simply forwards transaction requests to the partitions at which they will execute, then executes each partition's transactions serially in a single thread using the H-Store scheduler discussed above.

Like H-Store, H-Cal achieves extremely high throughput for embarrassingly partitionable workloads, but its performance degrades steeply when transactions may span multiple partitions[7].

5 Conclusion

Four years of work on deterministic transaction scheduling in distributed storage systems have culminated in a robust prototype with excellent scalability and replication characteristics. Its modular and extensible implemen-

[7]H-Cal's performance actually degrades more gracefully than H-Store's in the presence of distributed transactions, since H-Store does not use a deterministic execution invariant to avoid distributed commit protocols. However, the current implementation of VoltDB, a commercialization of the H-Store project, implements a logical logging mechanism and deterministic execution invariant nearly identical to Calvin's [22].

tation allows new ideas to be quickly integrated and instantly tested in deployments that partition data across hundreds of machines and consistently replicate state across geographically disparate data centers.

We plan to release the Calvin codebase under an open-source license, with the hope that researchers and system architects will find Calvin useful, both in our provided configurations and as a common substrate for quickly prototyping new systems by swapping in novel implementations of individual components.

References

[1] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[2] D. Batoory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering*, 1988.

[3] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, 2011.

[4] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In *SIGMOD*, 2010.

[5] M. J. Carey, D. J. Dewitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. V. The exodus extensible dbms project: An overview. In *Readings in Object-Oriented Database Systems*, 1990.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[7] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, 2000.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[9] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a Database System.* 2007.

[10] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 2010.

[11] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[12] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. *VLDB*, 2009.

[13] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[14] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *CoRR*, 2011.

[15] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 145–156. VLDB Endowment, 2013.

[16] R. C. Sears. *Stasis: Flexible Transactional Storage*. PhD thesis, EECS Department, UC Berkeley, 2010.

[17] M. Seltzer. Oracle nosql database. In *Oracle White Paper*, 2011.

[18] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[19] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *FAST*, 2012.

[20] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

[21] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[22] VoltDB. Website. voltdb.com.