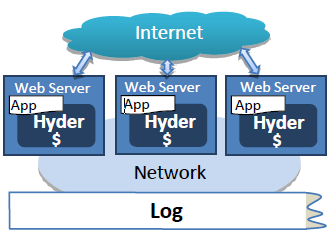
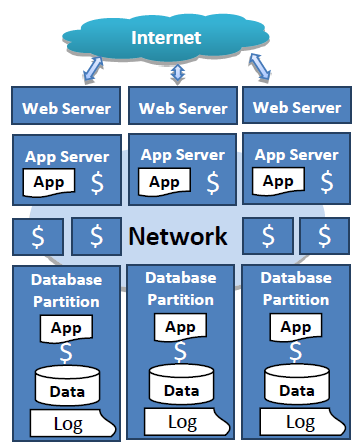
**3. Optimizing Online Transaction Processing (OLTP)**

OLTP is a kind of transaction-oriented information system typically used to handle a very large amount of transactions. Examples of uses of OLTP includes e-commerce transaction system, bank transaction system, etc. In such systems, a single database usually need to execute hundreds of thousands of transactions per second, as well as thousands of SQL queries per second. Thus when evaluating the performance of OLTP, data access throughput per second is an important benchmark. In addition, since the number of users of an OLTP is very large, operations like insert, update, and delete happen at the same time, how to properly handle concurrency control and how to scale out the system without significant slowdown in systems become a challenge.

Throughput of modern OLTP systems is often limited by various factors.

* **Disk access speed**. In OLTP, 90 percent of the time is spent on random or sequential data read/write, thus throughput can be improved by reducing disk I/Os.
* **CPU** **overhead**. When high speed storage, such as SSDs and flash memory, becomes less expensive, limitation by disk read/write speed can be reduced to minimal. Thus eliminating CPT overhead has become a major challenge in modern OLTP systems. Especially when it comes to distributed systems, overhead of handling locking protocol can have huge impact on transactional throughput.

In the following, we will discuss how different high-end systems exploit a various techniques to optimize OLTP performance. Then we are going to compare their improved performance using several benchmarks.

The Hyder architecture Partitioning architecture

**Figure 1**

**3.1 Hyder – A Transactional Record Manager for Shared Flash**

**3.1.1 Architecture**

Unlike many sharing-nothing systems, Hyder has the feature of scaling out without partitioning the database across multiple servers. It is a log-structured multiversion database, who is stored in raw flash chips and whose transaction commits are handled by optimistic concurrency control [1].

Figure 1 shows the architectural difference between Hyder and normal partitioning systems. As we can see, instead of partitioning all the database, application, log, and cache into different servers in multiple location, Hyder support data-sharing across multiple servers by allowing access to a shared pool of Log (which is a database) [1].

Traditional partitioning architecture has some limits on scaling out. Since caches, which are not shared, are on different servers, other servers cannot benefit from hot-data access on another server. It also means that wear-leveling cannot be achieved, a server that has frequently access data is more likely to wear out than those partitions contains less frequently access data. Though it is possible to design a good partitioning strategy that evenly partitions hot data and thus achieve load-balancing across all server, it is not easy to do so.

**3.1.2 Roll forward and Meld**

With Hyder, partitioning and distribute programming are not needed, since it shares a single log among all servers. Whenever a transaction is executed, it updates the single shared log instead of the log file within its own application. Then the updated log is broadcast to other servers, a roll forward algorithm is used to keep data up-to-date across servers. As every server can process any transaction, transactions can be spread across servers, thus load-balancing can be easily achieved.

Even though Hyder eliminates two-phase commit, it implements a procedure called “meld” to determine which transactions should be commit or abort. When every a transaction updates a record, the intention updated record is appended to the log. Note that at this time, the transaction is not treated as committed. A transaction is actually committed or aborted during the meld procedure, which occurs during rolling forward. The log keeps track of all transactions’ intensions. When a new intention is add to the end of the log, if it is conflict with other committed transactions, this transaction will be aborted, otherwise committed.

Although the overhead of two-phase commit is reduced, there are some other points that add cost to overhead, such as appending contention to log, broadcasting log to other server, rolling forward, meld procedure, aborted transactions. However, Hyder’s still performs better than most traditional distributed systems in terms of throughput [1].

**3.1.3 Storage**

Hyder uses raw flash memory as storage, which offers ~104 more I/O operation per second per gigabytes (GB) than hard disks. One of characteristics of raw flash chips is that it does not benefit from sequential access of data, though it performs better in the case of random access of data. While SSDs can be a good alternative of raw flash chips, SSDs are not append-only. Though SSDs can be implemented to turn random page-write into page-appends and automatically level wearing on the flash chip, it adds costs for this kind of transformation. Hyder can be also build on hard disk, if a server has large enough memory cache to avoid frequently operation on hard disk. Hyder is best supported by raw flash chips although it has the feasibility to be implemented with different storage media [1].

**3.1.4 Performance and Scalability**

Hyder can scale out linearly without partitioning by sharing a single log across all servers. Under simulation, when offered throughput increase up to 80K, we can see Hyder’s throughput also increases linearly while the abort rate is still negligibly small. However, its scalability is limited by the network, log and meld, which are thereby limited by hardware. Even though Hyder’s structure and roll forward algorithm eliminate the communication between servers about locking and commit, it has to keep collecting logs from the shared log as well as broadcasting logs to others, thus overhead can be very high once the number of distributed transactions reaches a tremendous high level. In addition, overhead may be higher if the loss rate of log messages becomes critically high because of network problem. Another potential problem with Hyder is its roll forward and meld algorithm. Though rolling forward and meld procedure eliminate the high overhead of two-phase locking, it cannot efficiently avoid aborting transaction.

The best part of this optimization is that it abandons the partitioning structure that most modern distributed systems use. This not only reduces the physical cost of additional devices, but also minimizes maintenance cost, since the most important component of it is the central shared log, which should be located in a single place. However, this also raises some other issues. For example, as there is only one copy of the log, which is also the database, data lost would be more vulnerable due to physical damaged, natural disaster, power outage.

This optimization has implemented a new way to improve distributed system performance, however, it is not fully ready yet. One critical problem it needs to solve to how to break the limits that are imposed by hardware and meld procedure. Limit by log structure can be improved with faster storage, and network speed is also increasing. Thus the most difficult part left is roll forward and meld procedure. We are expecting to see improvements on this part.

Though it has limitations, Hyder’s relatively lower cost makes it a good choice for users who don’t have much budget. Its easiness to implement and maintain doesn’t mean worse performance; instead, it performs fairly well if you take into account its hardware specifications.

**3.2 Calvin: Fast Distributed Transactions for Partitioned Database Systems**

Modern systems tend to achieve high scalability with the techniques of partitioning and replication, which is usually at the cost of system consistency. To make sure database can handle large amount of concurrent transactions, systems are designed to achieve 24/7 global availability by replication, thus lower system consistency. Examples include Amazon’s Dynamo, MongoDB, CouchDB, all of them provides highly flexible scale-out ability but supports very limited traditional ACID features [2]. While these distributed system achieve high scalability, but the reducing transactional support increase the design and implementation difficulty for programmers. For the increasing code complexity, development time grows and the performance of the system might be lowered due to the burden of ensuring consistency.

Calvin is designed to provide high scalability and full ACID transaction support by providing a transaction scheduling and data replication layer to transform a non-transactional system into a linearly scalable shared-nothing system [2].

**3.2.1 Architecture**

As shown in Figure 2, Calvin can be broken into three layers: sequencing layer, scheduling layer, and storage layer.

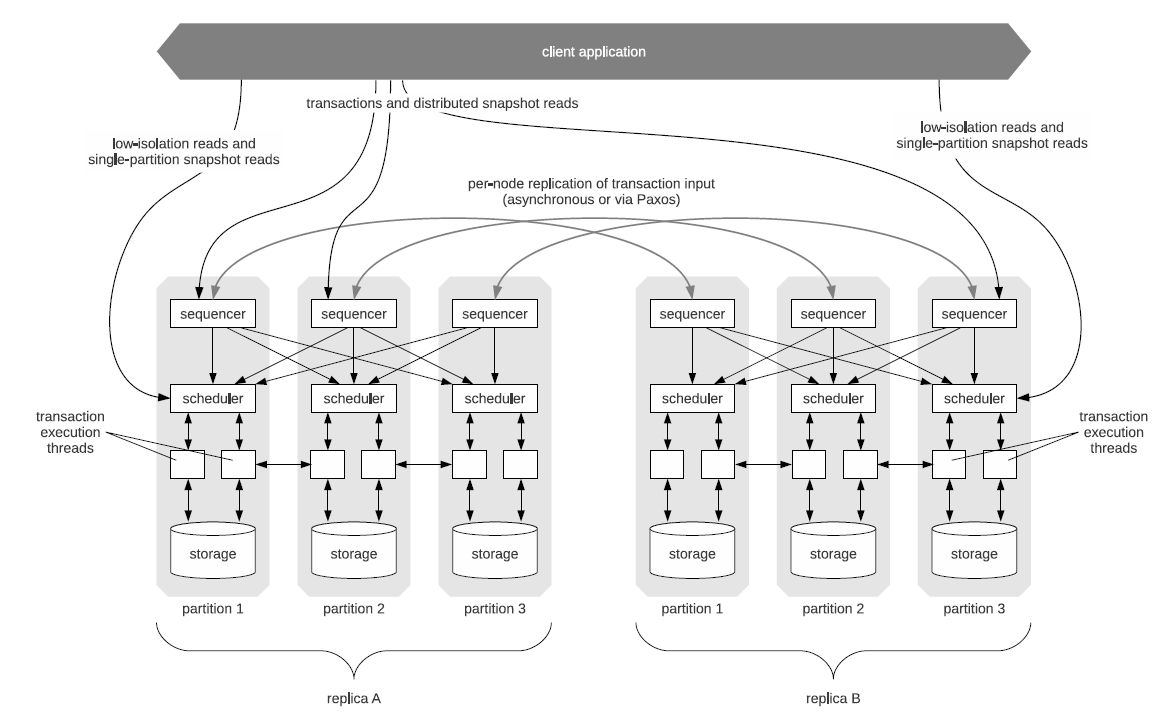
**3.2.1.1 Sequencing Layer**

The sequencing layer keeps a global transaction input sequence, which will be ordered in a way that satisfy serializability among all replicas. For normal distributed systems, sequencing might suffer a single point of failure, for all transaction inputs are stored in a single sequencer node. Calvin avoids this failure by compiling input transactions into a batch, which is then partitioned and distributed across replicas. In this way, every replica has its own copy of the sequence and thus avoids a single point failure. Also since each replica has a full copy of transaction inputs, it does not need to communication with others to make sure transactions are executed in a serial order, thus it reduces overhead overall.

**3.2.1.2 Scheduling Layer**

The scheduling layer schedules transactions with a deterministic locking scheme to guarantee equivalent serializability. It uses a modified version of two-phase strict locking. For normal strict two phase locking, transactions acquire lock before they start reading or write data; however in Calvin, a transaction needs to acquire all locks before it is executed, what it mean by this is that a transaction needs to declare all locks it will need in advance. And locks are guaranteed to be granted to transactions in the order in which those transactions request the locks [2].

The scheduling scheme that Calvin uses seems problematic, since transactions must be executed in the order that it get from the sequencer, even those later transactions have no conflict with earlier ones, later transactions have to wait until earlier ones complete. Calvin avoids this disadvantage by exploiting disk-based storage. Calvin actually prefetches data that are needed for transactions that are still in the queue. This is the so-called “warm-up” process that Calvin uses to do heavy lifting jobs for a transaction before it is executed [2]. In this way, when a transaction being executed, it should have all the necessary data. Hence, processing time of such a transaction should be minimal, which is claimed to take no more time than it takes in normal OLTP systems. However, it a transaction is “large”, it may need a lot of preprocessing work and take up a large amount of memory, which is likely to affect the overall throughput if the flash memory is not large enough. The system should limit the amount of resources that can be used for prefetching data.



**Figure 2: Architecture of Calvin**

**3.2.2 Recovery**

Calvin’s recovery strategy is efficient thanks to some characteristics of deterministic database system. First, because the transaction input sequence is determined and is replicated all over replicas, once a replica is failed, client can switch to another replicas without any delay. Second, when recovering from crash, it only needs to recover the transaction inputs to the point of failure, for the sequence of transaction input is unique. Calvin only needs to log transaction inputs rather than doing traditional logging, thus part of overhead can be saved.

**3.2.3 Performance and Scalability**

One bottleneck of this technique is the speed of disk [2]. The system only benefits from prefetching only when it can be done before a transaction is actually being executed, otherwise, the overall throughput would be significantly affected by the delay of prefetching data. It is also means that by improving the performance of disk (reducing I/O, disk latency), system throughput will also be boosted.

Under test, when system machines (nodes) increases up to 100, throughput per node stay constant if contention is very low (index = 0.0001); however, if contention is high, such is 0.01, we can see a gradual drop of throughput per node. Since real-world workloads often has very low contention, this disadvantage of Calvin handing high contention workload is “forgivable”.

Calvin is designed for those who wants high speed systems to handle large amount of distributed transaction without scarifying the fully support for traditional ACID. But Calvin’s performance very much depends on hardware, especially the speed of disk storage since the prefetching, and it also depends on the data type. Once prefetching rate drops, its performance drops as well. Thus for those business that will have a large amount of transaction operating on common sets of data, Calvin performs fairly well.

**3.3 Very Lightweight Locking for Main Memory Database**

As main memory as well as other fast storage media such as SSDs is becoming less and less expensive, disk I/O becomes a less important bottleneck for OLTP throughput improvement; by contrast, locking protocol appears to put limits on OLTP performance [3]. Thus the technique of very lightweight locking (VLL) is proposed as one of the solution to this problem. The basic idea of very lightweight locking is to use a protocol call selective contention analysis (SCA) to achieve high transactional throughput under high contention workloads.

VLL is designed under the framework of Calvin, which is a sharing-nothing distributed system that uses a deterministic locking scheme to improve transactional throughput as we describe in the last section. VLL improve throughput in a several ways.

**3.3.1 Locking**

Instead of using a central locking data structure (which is a traditional way), VLL stores the lock information along with the raw data, which means overhead can be reduce since both data and lock information can be obtained in a single retrieval. However, without a traditional lock data structure, it is difficult to track the sequence of transaction in the order of lock request/release. Thus, VLL maintains a global transaction order, which is similar to Calvin, and it also requires all transactions to acquire all locks they will need before they can be executed; but it implement a different algorithm that can perform even better than Calvin.

Lock information is stored as a pair of integers (Cx, Cs), which means the number of requests for exclusive lock and shared lock for a particular item. Other than this, there is a transaction queue for each item. Whenever a transaction request a lock for an item, the Cx or Cs value increments by 1. A transaction can have two status, free and blocked. If a transaction can acquire all needed locks immediate, which mean the lock information for accessed item shows Cx=1 and Cs=0 or Cx=0 depending on whether requesting a shared lock or exclusive lock, it is treated as free and it can be executed immediate; otherwise, it is labeled as blocked and placed in the transaction queue waiting for locks. Once a transaction is labeled blocked, it has to make it to the front of the queue to be executed. When a transaction is in the front of the queue, it means it becomes the first one who request the lock and no other transaction is holding the lock, since a transaction releases all locks and is removed from the transaction queue once it is completed. In this way, deadlock can also be avoid.

However, this raises a problem. What if it comes into such situation that transaction A is in the front of the queue, and transaction B is blocked previously, but now B should be good to execute because A is not holding any locks that B need; however, B still needs to wait until A completes so that it can make it to the front of the queue, thus being executed. The waiting of B is unnecessary and thus reduce throughput if this situation happens frequently (and it does if contention is high). VLL is able to handle this with Selection Contention Analysis (SCA).

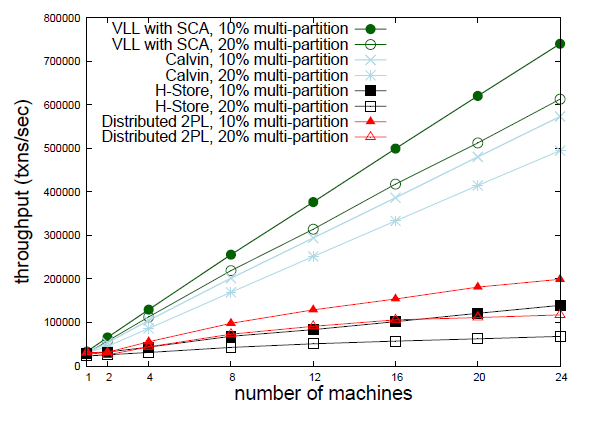
**3.3.2 Selective Contention Analysis**

SCA scan from the head of the queue through a certain position, which is depended on the contention, to check if a transaction is ready to be executed. In this way, the “waiting” situation described above could be eliminated. SCA uses a heuristic that the closer a transaction to the front, the more likely it is ready to run (can immediate acquire all locks). The analysis of the heuristic is that suppose a transaction is in position i, then it may conflict with up to i-1 prior transactions; as it get closer to the front, i decreases, thus the probability of conflict also decreases [3].

On one hand, SCA can reduce CPU idle time due to “waiting transactions”; on the other hand, it also adds cost to CPU overhead because it is doing a scan, although the scan is limited to a range. Therefore, SCA is activated only when it is needed; simply put, SCA is only activated when its gain outweighs its cost.

**3.3.3 Performance and Scalability**

Though under different configuration, VLL has different performance. VLL generally performs better than Calvin and H-Store, and it scales linearly with number of machines from 1 to 24. As shown in Figure 3, with SCA and 10% multi-partition, VLL’s throughput achieves 750K with 24 machines, whereas Calvin is 570K and H-Store is100K. The performance gap between these different systems are mainly due the locking protocol, which on the other hand admits that VLL has fewer locking overhead and thus improve transactional throughput.



**Figure 3: Scalability**

**3.4 Benchmark**

OLTP benchmarks are important tools that are used to evaluate OLTP system performance by simulating real-world workloads. OLTP benchmark is important and helpful for developers and researchers who want to test an OLTP system but are limited by having accessing to real-world data. The performance of an OLTP system is measured as throughput, which is the number of transaction per second. A good OLTP benchmark should (1) enable to user to validate alternative options by applying to different systems or using different configurations, (2) be able to perform performance metrics and compare them with real-world requirements, (3) be able to analyze the causes of performance bottleneck [4].

These features are important, even though every OLTP has similar functionality, they may focus on different aspect. For example, OLTP for e-commerce website focus on insertion transactions due to large amount of new order placed by users; banking systems may emphasize on update and read operations due the frequent update balance of banking accounts. Therefore, a good benchmark should provide the flexibility of configuration for users so that the benchmark can actually reflect real-world environment. In addition, performance metrics is important for users to evaluate the performance of a system in detail, which can also provide useful information about the relationship between performance and investment. Finally, suggestion from benchmark should be made to user as a reference to analyze performance bottleneck.

There are several benchmarks that can be used to evaluate OLTP system performance.

**3.4.1 TPC-C**

TPC-C benchmark is currently the industrial standard for evaluating OLTP systems [4]. It is comprised of ten warehouses, nine tables, 92 columns, 3 indexes, and five different kinds of transaction. Noted that 92% of its transactions is new-order transactions, thus only 8% is read-only transactions.

TPC-C benchmark is treated as industrial standard for several reasons. First, it includes all basic components of OLTP benchmarks, and was designed to carry over many characteristics of TPC-A. Some major characteristics are also added, such as more complex and realistic types of transactions, online and deferred execution of transactions, higher levels of contention for data access and update [5].

Second, TPC-C model simulates a real business model for an order-entry environment, like wholesale business, e-commerce system, where users perform various operations. Not only normal operations, such as placing new order, check order status, process payment, monitoring stock level, are supported, TPC-C can also forge error inputs that force transaction to be cancelled. Since there are 10 warehouses in total, part of transactions may requesting items in other warehouse, and this kind of transactions is also supported. The mix of transactions that TPC-C provides is complex, and they are all issued in a way that is close to the real world. Thus in my point of view, the various type of workloads supported by TPC-C a good simulation of the real-world workload.

Third, thanks to a wide range of data types and heavy workloads, it is able to resemble concurrent transactions very well. Transactions are issued one right after another without delay, which increases the stress on OLTP for handling concurrency [3].

TPC-C can provide two kind of performance metrics. One is the performance metrics in number of executed transactions per minute (tpm-C), another one is the price/performance metrics that shows you the unit price of tpm-C. Which one matters more depends on a user’s purpose. If a user’s application is supposed to have critical large amount of transaction, which requires a OLTP system to have very good support for it, performance metrics would have higher value to the users; on the other hand, if a user’s application requires only moderate support of heavy concurrent transactions, or a user has more concern about the cost, price/performance metrics is probably what the user need to take a closer look.

**3.4.2 SEATS**

The SEATS benchmark simulates an airline ticketing system where customers search for flights, check seat availability, make a reservation, update and check reservation information. The system consists of 60% of read-only transactions, which is searching for flights and check for open seats [4].

It claims to emulate a ticketing system, however, I don’t think this benchmark can effectively reflect real world transactions. First of all, for an airline ticketing system, there should be a larger amount, which should be greater than 60%, of transactions is read-only. Normally, most customers spends much more time on checking flight schedule and comparing every options than placing an order. Once a reservation is made, seldom does a customer change or update the reservation. In this perspective, I think this benchmark should increase the weight of read-only transactions in order to reflect real world customer behaviors.

**3.4.3 Other benchmarks**

There are other benchmarks that models different business environments. AuctionMark models the workload characteristics of an online auction site; TATP benchmark emulates a caller location system used by telecommunication providers; Voter benchmark is used to evaluate a voting system. Web-Oriented Benchmarks, such as LinkBench, Twitter, Wikipedia, etc., are used to model web-based applications like social networking, cloud sourcing sites.

These are all possible benchmarks for OLTP system, users should conduct a detailed research on these benchmarks and pick those that most matches their systems for performance evaluation.

[1] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder – A Transactional Record Manager for Shared Flash. In CIDR, paper 2, 2011.

[2] [Alexander Thomson](http://dl.acm.org/author_page.cfm?id=81477643334&coll=DL&dl=ACM&trk=0&cfid=427942483&cftoken=91548260), [Thaddeus Diamond](http://dl.acm.org/author_page.cfm?id=81502721299&coll=DL&dl=ACM&trk=0&cfid=427942483&cftoken=91548260), [Shu-Chun Weng](http://dl.acm.org/author_page.cfm?id=81392592140&coll=DL&dl=ACM&trk=0&cfid=427942483&cftoken=91548260), [Kun Ren](http://dl.acm.org/author_page.cfm?id=81502720488&coll=DL&dl=ACM&trk=0&cfid=427942483&cftoken=91548260), [Philip Shao](http://dl.acm.org/author_page.cfm?id=81502713854&coll=DL&dl=ACM&trk=0&cfid=427942483&cftoken=91548260), and [Daniel J. Abadi](http://dl.acm.org/author_page.cfm?id=81100545599&coll=DL&dl=ACM&trk=0&cfid=427942483&cftoken=91548260). Calvin: fast distributed transactions for partitioned database systems. In SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pages 1-12, 2012.

[3] Kun Ren, Kun Ren, and Daniel J. Abadi. Lightweight Locking for Main Memory Database Systems. In VLDB, vol6, p145.

[4] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe CudreMauroux. OLTPBench: An Extensible Testbed for Benchmarking Relational Databases. In VLDB, vol 7, p277.

[5] Transaction Processing Performance Council. http://www.tpc.org/tpcc/detail.asp