Fastore System Architecture

# Core Concepts

The core concepts for Fastore resulted from consideration of how a columnar store, such as that proposed by the TransRelational Model as described by C.J. Date, could be provided in a way that performed well under an OLTP environment.

## Columnar Stores

Columnar stores provide desirable characteristics for certain problems because they group locality around columns rather than tuples. For instance, if one wishes to determine if given value is in a certain column of a given table, columnar storage might provide a more direct answer without the need to skip over potentially vast amounts of irrelevant tuple data. Another benefit is that, depending on the storage strategy the columns may each be independently sorted, which opens up potential for rapid search and sort operations as well as the compression options.

The main problem with columnar stores is that query results often involve retrieval of many data columns and thus the lack of tuple locality introduces a random access requirement for row reconstruction. Another problem is the difficulty in actually performing row reconstruction, especially if the columns are ordered in some fashion other than some coordinated table ordering.

## Main Problem

We realized when designing Fastore that it is desirable to have each column sorted independently, but coordinate those values through a surrogate row identifier. This could conceptually be provided by 2 B-Trees per column, one by value, containing the associated row IDs, the other by row ID having the associated value. This essentially degenerates to the situation in a classic RDBMS with a secondary index defined on every column. Not only does this duplicate a large amount of data (well offsetting any benefits of column compression), it significantly increases the cost to update all that data.

In considering this, we realized that a technique we developed a couple years ago for another project we were working on, Kanvix, could be applied to solving this problem. Specifically, given a B-Tree of values, how to provide a secondary identifier which could quickly reference back to a given value without needing to duplicate that value. In Kanvix we solved this with a hash table structure which stored value tree leaf node pointers by row ID. This way, given a row ID, the value could be found by scanning just a single leaf. On the tree side, when a node was split or merged, those entries within the added or removed node were maintained in the secondary structure by re-pointing them to the appropriate node. We realized that this same technique could be applied to this problem.

## Multiple Rows per Value

In Kanvix, each Row ID constituted a unique entry whereas in this case, columns in general may have multiple rows with the same value. To address this, we realized that we could nest a key-only B-Tree or hash set within each value entry. For a disk based solution, the root node of an embedded B-Tree could, for instance, actually be embedded directly in the data portion of the value B-Tree. But what are the implications for the by-row structure. It turns out that the by-row structure can point to the leaves of an embedded B-Tree, so long as the leaves of that B-Tree have a direct or indirect reference to the root of that tree in order to find the value leaf node. Alternatively, for an embedded hash set in an in-memory, the value leaf node can be pointed to and efficiently scanned for a hash set that contains the given row ID.

## Remaining Issues

After prototyping this design, we found that we could get excellent performance both for read operations and for inserts. Inserts were actually twice as fast in our prototypes due to the row reconstruction costs. Under C#, with an in-memory solution we were able to achieve roughly 30K row reads/sec and 60K writes/sec for a modest sized table. In porting a subset of the prototype to C++ we found we could get several times the performance with the core structures.

All of these promising results, however, were shadowed by the fact that they were exclusively in-memory. It was clear to us that all columnar storage systems suffer from the tuple data locality problem, only to be overcome by undesirable redundancy. In other words, if our structures were stored on a hard disk, they would perform somewhat more poorly than a standard tuple store for heavy row reconstruction tasks due to the random read nature. We did realize, however, that the approach does have the advantage of being able to schedule significant IO in advance because we know rowIDs. For instance, if ranging over a Last Name column, we can Range Seek on that value B-Tree and obtain all matching row IDs. We can then, in parallel, request those row IDs from all other columns. With proper IO scheduling, that could significantly reduce the randomness of the loads, but nonetheless intrinsically involves more individual IOs. We address this next.

# Memory vs. Disk

There are basically two reasons to rely on hard disks drives (HDDs) rather than memory: persistence and capacity. Until computers have such architecture as to provide persistent main memory, most database systems will have the requirement that they persist data to “permanent” storage. The requirements around the other disk usage scenario, capacity, have changed significantly in recent years, however. Not only has 64bit architecture become the norm, allowing much better addressability of large amounts of memory, the price of RAM has shrunk to the point that it is economical to load even a low-end server with dozens of GB or RAM. At this point for instance, if capacity were considered alone, it wouldn’t make sense to use disk-based storage until the data size surpasses 96GB.

## Reconsidering Storage Architecture

Given modern persistence versus capacity requirements, coupled with the compressed characteristics of ordered columnar storage, it seemed a good opportunity to reconsider standard DBMS architecture. We wondered if we could even get something like the performance we obtained from our in-memory prototype in an actual system. The barrier to that seemed to clearly be the disk-oriented in-memory structures used by most systems and the many expensive locking schemes used to maximize concurrency.

## Memory / Disk Hybrid

The approach we arrived at was to keep lightweight column buffers in memory and to push changes to disk mostly in the background. DBMS scenarios that don’t involve synchronizing knowledge do not actually have the requirement that data be persisted before the client can move to the next operation. This fact is heavily utilized by the current crop of “NoSQL” systems which provide “fire and forget” update statements for those scenarios where durability is not required in order to continue[[1]](#footnote-1). We see this mode as appropriate for the majority of typical application uses and given the significant boost this can be to overall throughput and how well it complements the buffer/disk architecture this mode is the default.

So what is to be done in cases where, even with compression, the memory footprint of the buffers is too great for main memory? There are two options: distribute the data or rely on operating system paging. More will be said about distribution later. Operating system paging is not ideal, but adequate and may even provide fairly good performance for many cases. Normally, even systems such as Mongo which put heavy emphasis on in-memory usage, are required to arrange their data structures in a disk oriented manner in order to ensure page alignment and thus provide recovery and good IO performance. This was not a consideration for Fastore because the buffer representation is already redundant with the persistence system. The result is that the Fastore structures can be optimized for memory and in many cases are more compact.

Those points aside, Fastore did need data files and disk-based logs so that the same durability guarantees as are provided for major DBMS could be provided. For instance, if a consumer requests that a transaction be durable, the log will be flushed before that operation will complete as expected.

Due to the separation of buffer from disk orientation, some interesting new capabilities became possible. One such capability is that during re-load, the system may select whatever memory representation best matches the current data. For instance, if a particular column exhibits highly similar values the system may choose to use a buffer which uses compression.

# Concurrency

As mentioned, we wished to find ways to keep performance in league with the type of raw speed we observed in our prototypes, but this seems especially challenging in the presence of concurrency. Early tests which involved reconstructing columns in parallel, demonstrated to us how severe the cost of thread coordination primitives can be. A traditional concurrent B-Tree implementation is littered with complex latching mechanisms involving just such primitives and it became clear to us that if we wanted performance comparable to our aim, we’d have to rethink the concurrency strategy. In other words, it may seem advantageous to allow multiple threads to simultaneously operate within a given B-Tree, but the performance cost of accommodating for rare possibilities is so great that in many cases the entire operation could have be completed serially before the synchronization overhead is overcome.

## Column Buffer

It is this concept, around which we designed the structural concurrency aspect of Fastore. Rather than myriad fine-grain latches scattered throughout the B-Tree and other surrounding structures, we place the entire in-memory *Column Stash*, which is the primary B-Tree or other such structure as well as the by-rowID secondary structure, all behind a single, low overhead, spin wait queue[[2]](#footnote-2). The concept is that by bounding the workload of all operations which cross this boundary we are assured that no single thread will be able to hold up access to the column buffer for more than a small fraction of time. The Column Stash is wrapped by the *Column Buffer* which is responsible for providing this queued access. The job of turning larger operations into more fine grain ones can be reasonably pushed to a higher level of logic.

## Transaction Management

The Column Buffer thus provides for atomic operations within the internal structures of a single column, but clearly this leaves the problem of providing full transaction support across all the columns of the database. One approach would be to introduce a typical pessimistic locking scheme, whereby all operations flow through a lock manager. Though this has the advantage of utilizing common, well-known techniques, in some ways it negates the potential benefits of columnar storage in terms of distribution and sub-table orientation. Furthermore, pessimistic concurrency in general provides inferior performance to optimistic transactions for all but highly contentious write transaction loads. We wished to capitalize on the benefits of optimistic concurrency, while still providing pessimistic mechanisms for those certain situations.

## Column Manager

To provide a transactional conduit into the Column Buffer, we introduced the *Column Manager*. All requests to the Column Manager may include a revision*[[3]](#footnote-3)*; if one isn’t provided, the latest revision is assumed. Revisions are merely sequential numbers, associated with each modification transaction. The Column Manager ensures that read operations of a given revision always produce the same results. Write operations are more involved.

## Transaction Manager

To facilitate coordination of reading and writing across the entire database, we introduced the *Transaction Manager*. Note that transaction processing in a system where columns are potentially distributed implies that transactions are also distributed. When designing the transaction management strategy every effort was made to minimize centralized contention points and thus maximize the ability to distribute the work-load. The Transaction Manager represents an unavoidable contention point, so we strove to design it so that it was minimal.

### Transaction IDs

As a minimalistic distributed transaction coordinator, the primary role of the Transaction Manager is simply to generate and manage *Transaction ID*s (TIDs). A transaction ID includes the “origin” revision number, which was the active revision number when the transaction started. This transaction ID then becomes the mechanism for specifying the revision when performing read and write operations against the column managers. As mentioned previously, Column Managers will ensure that read operations from a given revision (encoded within the TID) always produce the same result.

### Other approaches

It should be noted that instead of revisions, we might have instead have chosen change-sets or old-new value verification. Using change-sets, we’d detect conflicts by comparing sets of changes, but this of course entails the cost of maintaining and comparing these. The old-new value scheme checks for conflicts by comparing old values against current ones before performing updates, but this requires the cost of tracking values, which are essentially variable in size, and doesn’t match up as well with the change primitive we use (essentially delete/insert). Furthermore, revisions provide the option of snapshot isolation.

## Transaction Manager API

|  |  |
| --- | --- |
| Start() : TID | Generate a new Transaction ID having an origin of the latest committed revision |
| Prepare(TID) : Revision | Begins the committal process – generates a revision number to provide to each column as the transaction commits. Calls to Prepare are blocked until (or error) until the transaction completes the commit. |
| Commit(TID) | Finalizes the transaction – all subsequent new transactions will take an incremented origin. Next transaction waiting to prepare is allowed to proceed. |
| Rollback(TID) | Indicates that there was a problem attempting the commit, cancel the pending status initiated by the Prepare. |

## Transaction Processing Strategy

Designing the transaction system, we realized there were essentially two main strategies we could take: immediate write or late write:

* **Immediate write** involves having transactions apply changes to the buffer as those changes occur, then back-off those changes for all readers from other transactions. Changes from all transactions must be kept and tracked centrally. Committing is low cost, merely a matter of removing the undo information from the centralized tracking agent. Synchronization between the buffer and other modules must occur at the change application level, more granular than the revision. Collisions would be detected at the point that changes are first applied, so long running transactions could have significant blocking effects on other transactions.
* **Late write** involves making no changes to the shared buffer(s) during change application; rather the distributed engine merely tracks the change. No transaction need be concerned with changes from other transactions until they are committed (atomically). Each transaction must redo changes made from that transaction against reads that occur from the buffer in order for the transaction’s own changes to be visible. Committing includes the process of actually applying the transactions changes within an atomic buffer operation. Collisions are not detected until commit time, eliminating possible blocking effects of long-running transactions.

In the end we opted for late write because transactions incur no centralized or ongoing tracking cost until commit time. Furthermore, the process of tracking and backing out appropriately all transaction changes centrally seemed inordinately complex.

### Buffer Manager Atomicity

An initially large concern we had with late writes concerns performing update processing within the commit critical section. Another issue was that changes need to be applied from the transaction to the buffer atomically, yet the buffer’s Column Stash enforces size limits in order to minimize stalling. We realized that we could solve both of these problems through a mechanism in the Column Buffer, which provides the atomicity. During the time that the changes are being applied to the stash (in sized blocks), requests from other transactions are provided transparency through auto-undo applied at the change level. In order to reduce the commit time, these changes can be pre-applied before entering the critical section (but not committed), so that during the commit, the buffer merely has to release the pending delta transactions. This is essentially the same process that would be necessary under the immediate write strategy, but limited to commit time.

### Pre-testing and pre-commit

Another process that must be performed during the first stage of commit time in either immediate or late write strategies is collision detection. Before changes are applied, it is necessary to ensure that all changes do not conflict with revision changes through the most recent. This detection cost could be significant and thus important to minimize while in the critical section. Fortunately, we realized that we could minimize this by pre-testing through the most recent revision captured before entering the critical section. Only new revisions committed since the beginning of this pre-test (if any) need be tested within the critical section.

### Transaction process

Simplified, the transaction processing thus works as follows:

1. Transaction Started – TransactionManager.Start() -> TID
2. Reads and writes performed
   1. Reads[[4]](#footnote-4):
      1. Get data – BufferManager.Get…(TID) -> data, rev
      2. Handle revision differences:
         1. Serializable: Detect read collissions – ColumnRevisor.Detect(TID, data, rev)
         2. Non: Rollback results to origin revision – ColumnRevisor.Revise(TID, data, rev)
      3. Re-apply transaction’s changes – Transactor.Reapply(data) -> data
      4. Track read cells if serializable – Transactor.AddRead(ColID, RowID)
   2. Writes:
      1. Track writes – Transactor.AddChanges(batch)
3. Pre-detect conflicts – ColumnRevisor.Detect(TID, Transactor.Changes) -> detectRev
4. Pre-apply changes – ColumnBuffer.PreApply(TID, Transactor.Changes)
5. Pre-store changes – ColumnStore.PreStore(TID, Transactor.Changes)
6. Prepare transaction – TransactionManager.Prepare(TID) -> commitRev
7. Detect conflicts – ColumnRevisor.Detect(TID, detectRev, Transactor.Changes)
8. Start storage sync – ColumnStore.BeginStore(TID, Transactor.Changes, otherColumnIDs, commitRev)
9. Apply changes – ColumnBuffer.Apply(TID, Transactor.Changes)
10. Commit transaction – TransactionManager.Commit(TID)
11. Complete storage sync – ColumnStore.Join(TID)

## Transactor

The *Transactor* is a non-centralized component which manages each transaction.[[5]](#footnote-5) The Transactor begins a transaction as a request to the centralized Transaction Manager, and commits transactions through comparable requests, but otherwise operates in isolation, coordinating activity for a given transaction consumer. Because changes are not applied centrally until transactions are committed, the majority of cost for long-running and large transactions is born by this decentralized component.

To provide for fully serializable transactions, a transactor must keeps track of the following:

* Transaction ID – This provides a unique identifier for the transaction as well as identifying the origin revision for the transaction.
* Changes – The set of changes made by this transaction
* Read Cells – The set of column/rows read by this transaction

The reason for tracking read cells is to detect conflicts between reads and data changes since the origin. This is necessary for full serializability in order to ensure that all nothing that was read has changed, but can be relaxed for lower levels of transaction isolation.

## Coordinated Persistence

Persistence of data to disk is certainly another area which has traditionally presented a concurrency challenge. We realized, however, that durability is an aspect that can be separated entirely from the buffer layer so long as there is some mechanism for describing the synchronization between them. Not only does such separation allow read-operations to execute entirely independently of persistence, but persistent write operations can also execute independently of the buffer access. Depending on the transactional requirements of the operation, the persistence may even be considered a background process. Storage is discussed in detail further on, but suffice it to say for now that revisions form the basis for coordination of storage and other modules.

# Distribution

As much as possible, we wanted to minimize bottlenecks in the architecture. Transaction management has traditionally been one such bottleneck, through which all operations passed. Subsystems such as lock management and even the DBMS service itself have represented

Distribution of columns, possibility of distributing within a column (sharding), distribution of engine, TP, Lock Manager, even persistence can be separated from buffering

Caching of low cardinality columns in the client engine. Extreme being all buffers in client, server(s) doing just storage

# Storage

# Architecture

Engine…

1. Unfortunately, these systems seem to go too far in this regard and also lose the ability to properly execute complex transactions, but that is discussed in more detail later. [↑](#footnote-ref-1)
2. It might be interesting to see if there is a reader-writer queue with sufficiently low overhead which could be substituted. [↑](#footnote-ref-2)
3. The concept is similar to what some DBMSs refer to as versioning, but we avoided the term because to most in CS the term version connotes similarity to a real number, where we wished to connote a sequential integer. [↑](#footnote-ref-3)
4. Note, due to the potentially high cost of detecting changes as a transaction grows increasingly stale, it may be advantageous to cache certain get operations. If advantageous, the entire buffer could even be copied as of a given revision. [↑](#footnote-ref-4)
5. As mentioned previously, if we had opted for immediate writes, the Transactor would be a centralized component. [↑](#footnote-ref-5)