Distributing the Mumps Database and Eliminating User-Level Locks

Nick Daly[*]

<2020-05-01 Fri>

Abstract

In Mumps, the lock operation (`L+`) is server-local which encourages centralizing processing on a single production server within a data center. This leads to significant capital investment costs to purchase servers large enough to handle the massive processing requirement of both entering and analyzing medical data both on demand and proactively. By storing data in a non-shared data structure, it is possible to reduce infrastructure costs by allowing multiple inexpensive servers to perform the work of one costly server. It is also be possible to improve user workflow throughput by preventing users from locking one another out of records.

1 Introduction

Mumps is both a classic NoSQL database and a programming language. Data are stored in first-class hierarchical, sparse, trees ("globals") that allow for high write throughput, while programs are written in an assembly-language similar syntax that supports allows for run-time evaluation. For all its simplicity and power, however, Mumps was originally designed in the 1960s for a much more centralized and server-focused world than the cluster-focused approach commonly used today. As such, all locks in the system are server-local, precluding many simple distributed designs. This paper takes the stance that, assuming the volume of data to process and analyze is too large to be cost-effectively handled on a single production server, it should be possible to circumvent the standard Mumps lock model to reduce centralization and hardware costs.

This research has three goals. First, it aims to reduce lock contention between servers by removing the concept of local locking, thus eliminating the concept of server-specific record ownership. Once record locks are no longer owned by any specific server, the database may be distributed across multiple servers without conflict, effectively making it a cluster-distributed-database. Another side-effect of removing local locks is that multiple users may document on a single record at the same time, using a type of last-edit-wins conflict resolution. This will enable future work to replace locks with a transactional model.

[*]ndaly@wisc.edu

2 Workflow

In order to provide an overview of the disparate parts of the system and how they work together, a workflow with example end-users is provided. Links in this section point to the project's annotated source code.

> *Mr. Steele has called Dr. Granite, asking for a refill on his allergy medications. On reviewing the patient's chart, she has decided to place a new medication prescription, because Mr. Steele's last prescription was over a year ago and he no longer has any refills remaining.*

When Dr. Granite goes to place the prescription, the system will then create a new medication prescription record in the `^MEDRX` global (database table) and associate any changes made in the doctor's session with a specific edit identifier. Since Mumps is a hierarchical key-value store, each change is recorded as a sub-node of the edit identifier. In this instance, the system has selected record ID 3150 and edit ID 29007.

> *Dr. Granite records a new prescription for 30 once-daily 10 milligram Claritin tablets, with 11 refills to last Mr. Steele a full year. She then asks a nurse to call the order in to Mr. Steele's pharmacy.*

The system will record these details in particular record fields, like the dispense quantity (field 6), or the number of refills remaining (field 8). To make sure historical data about a record don't get lost, each value is also timestamped by an edit instant, the current Unix Epoch in microseconds. This is detailed in the Data Global Structure section.

```
; (record, edit, field, instant)=value
^MEDRX(315,2907,6,1588363791787797)=30
```

When the new prescription is recorded in the prescription record, it's also separately recorded in the `^PATIENTLINK` global, which is used to associate other records with the patient's original record. This little bit of indirection allows many users to quickly link new records to the patient's record without ever locking the patient's record. Field 2 in the patient link record stores each of the prescriptions written for the patient. Since the patient link record allows multiple records of each type to be linked back to the patient, it includes another level in the data hierarchy, the entry, which stores one line for each concurrently-linked record.

```
; (record, edit, field, instant, entry)=value
^PATIENTLINK(219,1820,2,..7797,1)=315
```

At the same time the edits are recorded in the data global, they're also recorded in the `^AUDIT` global, a chronological ledger or journal of every change made in the system, for later reporting and synchronization purposes. In order to enforce chronological order, the first two subnodes of `^AUDIT` are timestamps. The first sub node is the local instant that the server first learned about the edit while the second subnode is the edit instant as recorded on the originating server. In the case of local edits, these values will be the same. This is detailed in the Audit section.

```
; (i1, i2, global, record, edit, field)=val
^AUDIT(..7797,..7797,"MEDRX",315,2907,6)=30
```

Once the data are saved to the `^AUDIT` global, the changes are submitted to a background process that pushes them on to the distributed database, an Apache ZooKeeper cluster. This is detailed in the Cluster section.

```
/audit = 9
/audit/9="^AUDIT(..."MEDRX",315,2907,..."
```

The local server periodically reads changes in from the cluster, in the form of `^AUDIT` nodes. When the nodes arrive, the first and second subnodes are equal (because the edit was synced from the server it was created on). The server then replaces the first subnode (the local instant) with its own local time before loading the edit into its own `^AUDIT` and data globals. This allows the data globals to appear chronologically according to when edits were originally made while the `^AUDIT` global is ordered according to when the server was first notified about the changes. This is detailed in the Audit section.

> *Dr. Granite moves to finalize her documentation and close the patient's chart. Before the chart saves and closes, however, she must review any warnings accompanying her new documentation as well as any other changes that were made to the Mr. Steele's chart since she opened it. It looks like her nurse, David, updated Mr. Steele's weight while on the phone. It looks like he's helpfully lost a few pounds, but is nothing that would affect his allergy medication. Dr. Granite makes a note to congratulate Mr. Steele on his exercise regimen when he comes in for his annual physical next month.*

In the future, this project will apply the user's changes to the record as a transaction and stop to display any other changes recorded in the `^AUDIT` global since the user opened the chart. The user will have the opportunity to review the changes, the originating user, and that user's contact information (in case clarification is needed), before accepting their own changes.

3 Design

3.1 Data Structure

The system is built from two primary pieces: the local database and the remote cluster.

3.1.1 Data Global Structure

In this system, data are stored in globals (equivalent to SQL tables) that contain five identifiers: record ID, edit ID, field ID, save instant, and value. IDs are not divided by record type, but a single list of available IDs is shared between each type of record. Edits follow the same shared ID paradigm. This was done to reduce the amount of state in the database and the amount of data that would need to be synced with the cluster. The meaning of an edit is use-case specific. A patient edit may be a specific appointment, while a medication edit might be an administrator loading a monthly medication pricing database update.

3.1.2 Cluster Structure

In this system, the usual distributed database configuration is assumed: that a single ZooKeeper cluster,

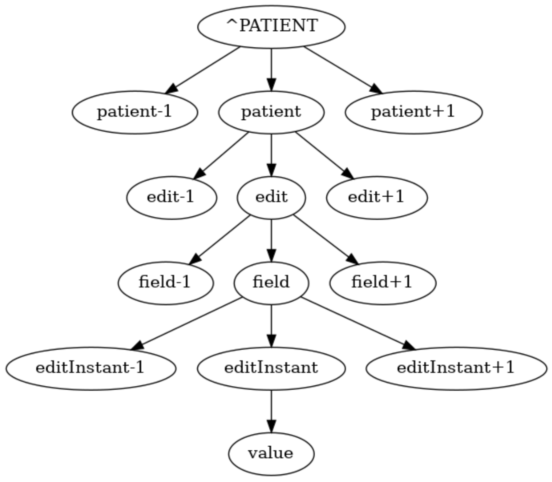


Figure 1: Patient key-value datastore, where every value has its own edit instant.

made up of several database nodes, serves requests for multiple application servers which each serve multiple users on any number of connected devices. Any node in the cluster may respond to read requests but, in order to preserve a single total order for events in the cluster, only the leader node may write to the cluster's state. A server trying requesting a write may contact any server in the cluster and will be redirected to the cluster's current leader. The changes submitted are the newly-created `^AUDIT` node entries which are pushed into the next free entry under the cluster's own "/audit" node.

3.2 Database Contents

Two main types of data are stored in the database:

1. Data Globals: Each data global stores one type of record data.

2. Audit Global: A running transaction log or journal of all the changes made to the database, used for syncing to and from the cluster.

3.2.1 Global List

For this experiment, a custom data model was devised to hold test patient data. Seven data types were created to hold six discrete types of data, along with a linkage record (1-to-many relation table). Each of those data types is stored in a separate global with different fields.

Patient Stored in the `^PATIENT` global, patients each contain basic information about the patient, like the patient's ID, name, sex, and birth date. Appointment-specific information is also recorded, like the patient's vitals, including blood pressure, temperature, and respiration rate.

Medication Stored in the `^MED` global, medications each contain an identifying name. More complex and complete information like generic form, RxNorm drug identifier [20], form-specific drug concentrations, or pharmacy availability, was not necessary to create a usefully complex data model.

Prescription Stored in `^MEDRX`, prescriptions each contain a link back to the patient's ID, as well as the medication ID that they represent. They also contain non-record-relation information, like prescription duration, and frequency and number of refills remaining.

Lab, Procedure Stored in `^PROC`, laboratory orders and procedures are, like medications, identified by name. These include procedures like a forearm x-ray or diagnostic laboratory analyses like a complete blood count ("CBC").

Notes Stored in `^NOTES`, medical notes are text-based documents users can write about patients during a visit. They are linked to the patient and may be hidden or made visible to the patient.

Ordered Procedures Stored in `^PROCRX`, ordered procedures are used to link a specific patient and procedure. They also contain information like the date the procedure is scheduled for and any comments the physician might want to note when requesting the procedure.

Patient Link Stored in `^PATIENTLINK`, this record links to lists of notes, prescriptions, and lab or procedure orders for a single patient. This allows, for example, multiple users to edit different prescriptions while another user was edits the patient's visit. In this case, users would only compete for a lock when adding or removing a prescription from the link record itself.

3.2.2 Audit

Core to this implementation is the `^AUDIT` node that contains the local times an edit was received from the cluster and when the edit was originally made on the local server. This allows us to preserve the total order of both local and remotely created edits The first two nodes of the `^AUDIT` global are timestamps: the local timestamp, when the record or edit

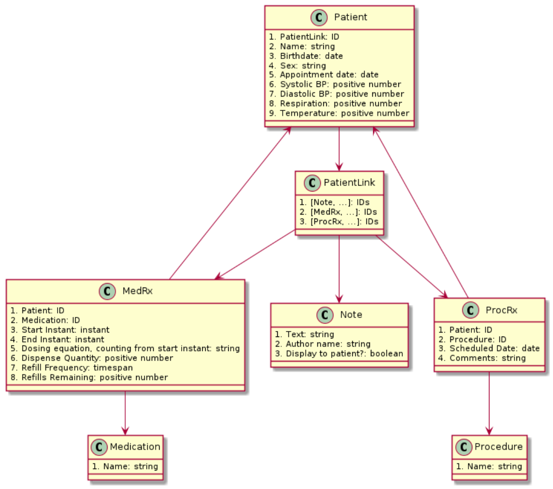


Figure 2: The experiment's data model and data linkages.

("change") was created or received, and the cluster-wide timestamp, when the edit was first created on its originating server before being uploaded to the cluster.

In the case of local edits, the first two nodes are always equal. However, when edits are loaded from the cluster, the server will replace the first timestamp with its own current timestamp. This keeps the server-local chronology of the audit trail self-consistent: entries are ordered according to when the server first learned about them, not when they were originally applied to a record on a remote server. Enough contextual information is provided that complex determinations about event ordering can be made, like "this morning, we learned that last month's appointment was updated yesterday." This is useful in the case of an end-user reviewing the patient's chart: it allows the system to highlight new data (edits to old appointments) that arrived since the user began the chart review while displaying it in the historical context of the appointment it was recorded in. It also makes it possible to play back what information users on the system had at any point when making decisions.

`^AUDIT` has a slightly different format for local edits than it does for remote edits. Since record and edit creation times are non-actionable metadata, they aren't mirrored to the cluster to reduce the total cluster load. Thus, all the record or edit creation times stored in `^AUDIT` are for locally created records or edits.

3.3 Eliminating Conflicting Edits

Methods used to prevent local and remote users from overwriting existing data in the database and thus eliminate the need for record-wide server locks, include:

1. Making a high-resolution, server-unique, edit instant part of each value's address.

2. Making a non-overlapping, server-unique, record and edit identifier part of each value's address.

3. Keeping a permanent historical record of the local and remote edit times for each change.

With these three methods together, it is nearly impossible for either local or remote users to accidentally overwrite existing edits. The single outstanding case is addressed in the Future Work's Data Structure Improvements section.

3.3.1 Global Structure: Removing Local Locks

Locks are applied to records to prevent users from making conflicting edits to a record. Normally, when a user opens a record for editing, the system first tries to acquire an exclusive write-lock on that record. If a record contains a broad array of data so that there are many workflows and use cases where a user might want to lock a record, or if a record is in high demand, any lock may delay a large number of changes.

Since locks are used to reduce the chance of conflicting edits, the obvious solution is to reduce the size of the window in which conflicts could occur. Thus, every value is addressed by an edit instant, and the database has no concept of ahistorical (single-entry, most-recent-value-only) data. This allows us to effectively treat the database itself as an append-only transaction log. Since edit instants have a microsecond resolution, are server-local, and are tied to a specific field on a record, rare conflicts may be resolved by incrementing the edit instant until an unused microsecond is found.

3.3.2 Cluster-Synchronized Edit Identifiers: Removing Remote Locks

Several other steps can be taken to eliminate the need for locks, even when the database is distributed across a cluster. The simplest approach is to give each server its own set of unique identifiers for both new records and edits within those records. This way, any change created on a server must have an ID within that server's range and won't accidentally be merged with any other server's records when it's pushed to the cluster. When a server has consumed 95% of its available change ID range, the server will query the cluster until the change range shortage is resolved. The server will also continue to allocate IDs until the entire range is consumed. If the entire range is consumed without a valid cluster reply, the server will hang until it receives a reply, as it can no longer safely allocate change IDs to processes.

With these reserved ID ranges, no server needs to worry about accidentally conflicting with another server's IDs. Since these ID ranges are monotonically increasing values, new servers can be added to the cluster at any time. New servers will notice that they have no ID range reserved and query the cluster before creating any new changes.

Taking guidance from Chubby [3], the cluster is able to modulate the rate at which servers request new change ID ranges. This is done by introducing a change scalar which determines the size of each server's change ID range. Currently, new ID range requests increment the cluster's change identifier by one, and the server then reads the cluster's scalar value to determine the new ID range: [ID * scalar, (ID+1) * scalar). The cluster may increase the scalar at any time to slow down the rate at which servers request new IDs.

3.4 Patient Safety Implications of Transactional and Lock-Free Design

Since this data sharing model could be used in safety-critical situations [6, 14, 18], it should be reviewed relative to what sort of harms could befall a patient in which situations.

In outpatient (family practice) situations, the data update analysis is straightforward. Assuming most documentation is completed while the patient is present or within 24 hours, it's unlikely that the patient would have interacted with a different server that would have made meaningful changes to the record while the local user is documenting. In that case, the most recent previous data is likely days or hours old and would have been loaded from the cluster minutes after it was initially documented. The patient's providers would have had all the relevant and updated information before they even started documenting on the patient.

The problem becomes harder for admitted inpatients where multiple providers on a care team can document on the patient at the same time. However, this issue is mitigated in two ways:

1. Server-locality. Since the patient's care team of providers are often server-local, limited to the team working within a single hospital, no cross-server talk is generally required. In the case of consulting providers (at other locations, on other servers) those providers are generally limited to a consulting role, and consult with a local provider who makes final decisions and orders the actual treatment, again keeping the edits local. This server-locality allows the server to act transactionally and prevent the user from saving edits if other, unreviewed, data has been created for the patient since the user started.

2. On-action warnings. Modern Electronic Medical Records (EMRs) contain a vast array of warnings that occur when users take actions. Some of the most urgent warnings are medication-administration warnings. When a nurse is preparing to administer a medication, that medication is validated in several ways, including against other medications it may interact with, and against medically recommended doses. If a medication is found to interact badly with another medication, one of the patients conditions, or is simply an unusually high or low dose (because the ordering user mistyped), the administering user will be warned before administration. Users would generally seek guidance for unanticipated or severe warnings, preventing them from unintentionally completing a potentially dangerous administration.

   A patient transitioning between servers is also unlikely to result in medication administrations being lost. Since outgoing syncing is immediate, the administration would be pulled down to the local server no more than two minutes after it was documented. It is unlikely that a patient would be discharged from one emergency department, arrive at another, be triaged, admitted, and prepped for the administration of medications in under 2 minutes. The more problematic workflow would be a patient who left against medical advice immediately after the administration, possibly distracting the administering user and preventing them from marking the medication administration as complete. However, the data would again sync inside of two minutes and be available for any future emergency department. A patient who arrived at an emergency department immediately after leaving another one would probably result in calls between the departments and certainly would be examined closely. In that situation, the EMR would have synced data quickly enough to flag end users that something fishy was going on.

Nonetheless, there two are cases where data may not be communicated. First, if the receiving site can't receive data from the EMR either because they don't have a compatible (or any) system, and second when one of the servers or the cluster is down. In that case, the information simply is not there to be made available to local users, though users expect it to be. Given this possibility, any robust analysis of the patient safety implications of a particular EMR on the local area should include basic availability metrics like system compatibility checks and uptime percentages.

3.5 Software Libraries

The following libraries were produced as part of this experiment and are available to users:

SetData Provides functions to save data to local records, in response to user edits.

Cluster Provides a convenient front-end for bidirectional cluster data movement.

3.5.1 SetData

The functions available in the `SetData` library are concerned with saving data to local records in response to user edits.

NewRecord Reserves a new record ID from the server's shared list of record IDs. If the number of IDs remaining in the ID list is below the remaining ID threshold, the server will query the cluster through `nextId^Cluster` to get a new list of available IDs. When the server receives the new list of IDs, it will immediately replace the existing ID list with the new one, regardless of how many IDs remain. Thus, setting the low-water mark is a balancing act between responsiveness and wasted ID consumption. The new ID requests are made on a separate thread to avoid unresponsiveness, though the server currently waits until enough IDs are available before continuing.

NewEdit Reserves a new edit ID from the server's shared list of edit IDs. Also queries the cluster for new edit IDs when necessary.

SetRecord After a record and edit ID are reserved, the user may then save data to that record on that edit. This has the side effect of also sending the newly set data out to the cluster on a separate thread. For ease of use, `SetRecord` assumes that each field contains a single value.

AppendRecord To handle `^PATIENTLINK` records, which can hold multiple values per field, `AppendRecord` adds an additional level to the data hierarchy that stores the value's current location in the field's list. Since Mumps databases are sparse this can lead to surprising data storage. It would be possible, for example, to have a list with only a third entry.

GetRecordList Return a whole list at once, for a particular field's edit instant.

SetRecordList Set an entire list at once, for a particular field's edit instant.

3.5.2 Cluster

Several functions for interacting with the ZooKeeper cluster are available.

Query Perform a query against a cluster. Currently, ZooKeeper clusters are the only supported cluster type. If getting data from the cluster, return the node's value and version. If setting data in the cluster, the user must include the most recent version ID for that node before the cluster will perform the set. The system will query the cluster no more often than the query delay, which is one of the settings available in the `^CLUSTER` global.

Increment Increment an already-existing node. The system queries the cluster at least twice: first to get the node's current value and version, and then again to increase the value by one. If no other sets have been performed on the node, then the version is still current and the set succeeds. However, if the node has been changed between the two requests, the local version is outdated and the process gets the node again. If the process can't increment the node in 10 tries, it fails. It is then up to the caller to retry the increment or throw an error.

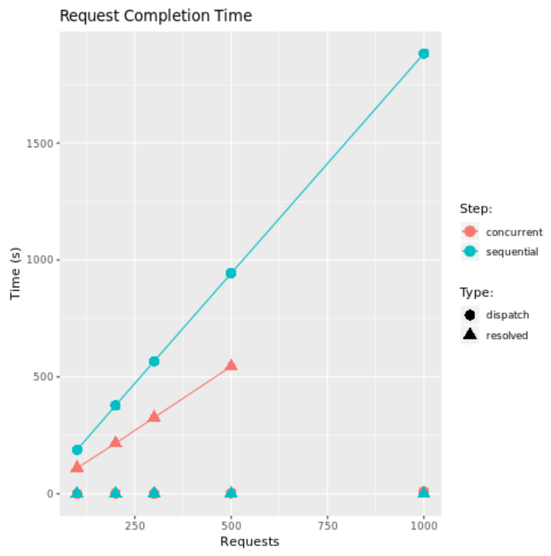
Start Starts a background job that reads in database changes from the cluster every minute. Since ZooKeeper supports 10,000 - 20,000 operations per second, this should be a very manageable load [8]. Every database change pushed to the cluster is given a monotonically increasing ID. The background job uses that ID to keep track of the last dataabase change it loaded and then, once every minute, loads in all the new changes from the cluster. This does result in the server reloading the changes it previously sent via `SetRecord`.

Stop Terminates the background job reading from the cluster. This function does not prevent the server from sending outstanding updates back to the cluster, in order to avoid stranding changes on the local server.

4 Performance

Performance measurements were taken over a period of two days on the CloudLab Wisconsin cluster, using a c220g1 system, as detailed in Table 2. The throughput of two different cluster communication dispatch methods were tested, in which the server queued up 100, 200, 300, 500, and 1000 requests and dispatched them either concurrently, as fast as the server could edit records in the Mumps database, or sequentially, allowing one record to complete before moving on to the next. The dispatch method was controlled by either forking off the database update process or allowing it to run in the main thread.

These measurements showed that dispatching cluster update events asynchronously, on a new thread, is the best way to keep the cluster updated. Overall, the concurrent dispatch process takes 0.57 seconds per request on average, while the sequential process takes 0.99 seconds per request, or 73% longer. This suggests that a significant amount of the delay is caused by synchronous network communication instead of disk time or CPU processing. These results were highly unexpected because ZooKeeper is advertised as handling thousands of requests per second. Nonetheless, the concurrent dispatch method was able to saturate and disconnect the remote ZooKeeper test server simply by overwhelming it with unresolved open connections. In those cases, the server could only be recovered by rebooting. Further investigation showed that the JRE was unable to create new threads to handle the incoming connections because no RAM was available. It is not clear whether the errors were due to the server being overburdened or were caused by the steps taken to recover the overburdened server.



5 Analysis

The data were analyzed with respect to both raw performance and hardware cost.

5.1 Change Performance

Further investigation into the performance properties of the system is warranted since, although the ZooKeeper cluster consumed over 35 GB of RAM during the run, one request per second is far below the expected rate of over 10,000 requests per second. These results were consistent with other testing, though, as even making 100 sequential requests to the ZooKeeper cluster directly from the command-line took 65 seconds. Given that every set request takes two queries (an extra request is required to load the current node version before setting it), then the system can only sequentially complete 50 Mumps data sets in 65 seconds. A different or custom cluster query system that replaces the ZooKeeper command line client might be required, as the cluster remained largely idle during the 30 minute 1000-sequential-connections test. Improving the throughput may also increase the number of concurrent requests that can be handled at once: the system emitted 100 - 300 Mumps changes per second, or an average of 400 cluster queries per second. Based on the 10,000 cluster queries per second target, ZooKeeper should be able to handle at least 5,000 Mumps changes per second.

While it is possible two different sets of cluster throttling settings could have come into play, this is unlikely for two reasons:

Table 1: Request completion time, in seconds, per mode.

Requests	Concurrent Dispatch (s)	Concurrent Complete (s)	Sequential Dispatch (s)	Sequential Complete (s)
100	0	110	188	0
200	1	216	378	0
300	1	326	567	0
500	3	545	943	0
1000	9	<error>	1883	0

Table 2: CloudLab University of Wisconsin c220g1 Hardware Configuration [4]

CPU	Two Intel E5-2630 v3 8-core CPUs at 2.40 GHz (Haswell w/ EM64T)
RAM	128GB ECC Memory (8x 16 GB DDR4 1866 MHz dual rank RDIMMs)
Disks	Two 1.2 TB 10K RPM 6G SAS SFF HDDs; One Intel DC S3500 480 GB 6G SATA SSDs
NICs	Dual-port Intel X520-DA2 10Gb NIC (PCIe v3.0, 8 lanes); Onboard Intel i350 1Gb

1. ZooKeeper doesn't being throttling the incoming connections by default until at least 1000 connections are open. Most of the tests never opened that many connections.

2. The `^CLUSTER` global's own throttling settings were set to prevent communication with the cluster more than every 0.1 seconds. Individual requests took longer than that to resolve in both dispatch methods, and so were not rate-limited by the internal throttling settings.

Regardless, the performance results are clear, the concurrent dispatch model, with a maximum-number of concurrent connections, is `1.73` times faster than the sequential model.

5.2 Per User Cost

The cost of recreating a c220g1, that supports at least 500 concurrent changes, is around $1,500, based on today's non-sale prices. If aggressive writes are assumed and that each user creates one record every few seconds while interacting with patients, then it can be inferred that, since creating prescription records require 9 changes, a small clinic with 50 concurrent users can be supported at a cost of $30/user. If the throughput can be improved to the expected values, 5,000 concurrent changes, then 500 users may be supported for $3 each. With write batching, as noted in Cluster Throughput Improvements, the cost per user may be able to continue to drop, precipitously.

The current limitation is the speed with which changes are committed to the database. Right now, one prescription record that contains 9 database changes would take nearly 10 seconds to commit, while it should take only a millisecond. If the performance issue can be resolved, this may be a viable option for organizations looking to save money while decentralizing their hardware.

6 Future Work

During the course of the project, several possible enhancements and optimizations were identified, but were unfortunately too late or large to implement in the available time.

6.1 Parallel Documentation

Next steps for this work include adding a UI for entering data and displaying potentially conflicting data to the user. In particular, the user must be shown data that was entered or received since the beginning of the user's session before they're allowed to save new data.

Performance measurements are expected to improve in office visit workflows where both nurses and physicians interact with the patient. The physician's section of the workflow (therapy planning and ordering, including writing prescriptions and lab requisitions) may now be started concurrently with the nurse's section (rooming, including documenting vitals, medical history, and therapy compliance). Since reviewing data takes less time than entering it, performing the workflows in parallel may result in a faster workflow, even when including additional time required for the physician to asynchronously receive the patient and review the nurse's documentation. Additional time would be required if reviewing the nurse's documentation invalidated the orders the physician was about to place.

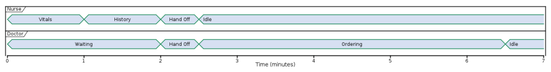


Figure 3: Sequential workflow, where both users document on the same physical device.

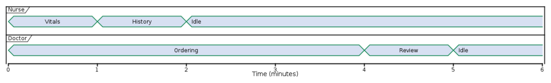


Figure 4: Parallel workflow, where physicians start charting before the nurse finishes with the patient.

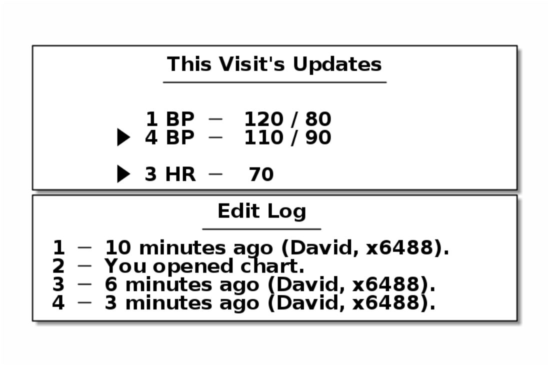
6.2 Record Entry and Data Review UI

The trickiest part of this system will be making the UI fast and intuitive. The system must record the instant a user opened a patient's chart and push any updates for that patient it receives from the cluster (or other users) to that user before any decisions are finalized. This involves an 8 step process:

1. User opens patient chart to begin documenting.
2. System records that time as user's documentation start time.
3. Additional information is recorded by other users on the patient's record.
4. Additional information is received by the cluster on the patient's record.
5. User attempts to finalize their documentation.
6. System pushes the changes that occurred between now and the user's documentation start time to the user's session, indicating which data elements changed and the date they changed on.
7. User reviews changes and either commits the changes or updates their own documentation (returning to step 2).
8. User closes patient's chart.

Step 6 is the most complex piece and requires the most care. When the user has indicated that they want to finalize their documentation, an efficient UI may switch from a single-pane data-entry view to a two column data-entry and data-review display. New data entered in the visit would need to be timestamped and highlighted in the review column. This effectively creates a change log against which the user can compare their changes.

The system could also be more restrictive by momentarily locking or turning the user's edits into a transaction for steps 5 and 6, and rejecting the edits if any new ones had been applied during the process. This would prevent any other users from finalizing documentation at the same time and preventing near-concurrent edits. Alternatively, the system could make a local copy of the patient's record and then commit the users changes on top of that copy before asking them to merge their copy back into the data globals.


This Visit's Updates

1 BP – 120 / 80
▶ 4 BP – 110 / 90

▶ 3 HR – 70

Edit Log

1 – 10 minutes ago (David, x6488).
2 – You opened chart.
3 – 6 minutes ago (David, x6488).
4 – 3 minutes ago (David, x6488).

Figure 5: Proposed data review UI elements.

6.3 Cluster Throughput Improvements

In order to minimize the number of writes made to the cluster, the outgoing cluster synchronization model should be changed from one that immediately writes every changed field to the cluster, to a batch model, similar to how cluster changes are ingested. The system should keep a timestamp of the last change synced to the cluster and periodically sync all the changes between then and the current instant to the cluster in a single batch of one or more messages. This change is no more inefficient than the current approach, because the system is already reading and writing its own edits from the cluster.

6.4 Data Structure Improvements

It may be wise to demote the edit identifier node in the non-audit globals to a lower position. Since edit identifiers are sequential only within servers and also work as server IDs, edit 59003 (at the beginning of the 59000-60000 range) may have been made months before edit 49871 (at the end of the (49000-50000 range). This is a highly unexpected outcome in a hierarchical and otherwise chronological data model.

The simplest solution is to demote the edit ID beneath the edit instant so that edits are correctly ordered. However, that mirrors the structure of the AUDIT global and hides the historical chronology of the data. A better solution may be to replace the edit ID with the edit's original date, and move the edit ID beneath that date. That would allow appointments to appear on the patient's record chronologically, while also keeping edits that affect the same date close together, regardless of their source server. Importantly, the standard Mumps date identifier (`+$H`) would need to be specially configured to return a UTC date [5].

Finally, there is one possible case that could result in overwriting existing data in a cluster, and the data model should be adjusted to handle that case. If two remote users on different remote servers make changes to a preexisting edit at the exact same microsecond, then the system will be unable to differentiate between those edits when they're synced back from the cluster. One would have a later local edit instant, but because the record ID, edit ID, and remote edit time were all the same, the two entries would overwrite one another. The obvious solution is to allow the remote edit loading process to reject already existing entries but the system relies on overwriting existing entries to advance the state loaded from the cluster without keeping a list of which entries are local. The simplest solution is to require each record edit to occur on a new, server-local, edit ID. This approach becomes much more reasonable if the edit ID is demoted beneath the edit's date, as mentioned above.

6.5 Cluster Audit Errors

If the above changes were made to the data hierarchy and storing rules, it would be entirely reasonable for the local server to log an error when ingesting cluster data that changes the value of already recorded data. While conflicting edits are incredibly unlikely, it is unwise to be unprepared for data corruption somewhere in pipeline.

6.6 Call Out to ZooKeeper through JNI

As noted above, switching query systems may help increase throughput. Implementing a new query system would require using the Mumps external call system [5] to call into a C-library that uses the Java Native Interface (JNI) to call into ZooKeeper and return replies. This would also help simplify the `query` and `parseOutput` calls since much of the large amount of spurious output could be avoided.

7 Related Work

While work is regularly done to improve hardware utilization and throughput such as when Intel and Intersystems worked together in 2015 to improve database throughput by 60% in 2015 [9], organizations involved with commercially available electronic medical record systems do not publicly publish studies about their private data models. Even the open source medical records, Gnu Health [11], OpenEMR [13], OpenMRS [21], and VistA [1] do not appear to study how the format of the data model affects data sharing and user workflows. Much more public work has recently been done on the data models used to exchange data between different healthcare organizations [10], including FHIR ("fire") [2, 7], the standard message structure through which organizations exchange data and USCDI [16, 15], the specific medical data elements organizations exchange. While those are important and necessary avenues of study for patient treatment, it seems likely that more research into how low-level data structures inform high-level user workflows is warranted.

8 Conclusion

Overall, eliminating Mumps database locks holds promise, but is not yet completely proven. It is possible to safely add a distributed ZooKeeper database to a Mumps database in a cooperative fashion to leverage the fast local access of the Mumps NoSQL database, while reducing hardware costs by distributing database writes across a cluster. However, before organizations could take advantage of the reduced hardware costs, further work would need to be done on improving the performance of the ZooKeeper database. Once that is resolved, the next step would be to implement a transactional user interface.

9 Reproducible Research

The sources for paper are embedded in this file and can also be downloaded or reproduced from the disk-image below by following the instructions in the README.

- `https://gitlab.com/nickdaly/cs790-p1/-/blob/master/bin/cs790-p1.img.xz`

New performance data may be captured by running `make src && make data` from within the cs790-p1 folder. Additionally, the full set of experimental notes may be reviewed.

10 Mumps Quick Reference

This is a very quick introduction to the Mumps language and database, though more comprehensive guides exist [17, 19, 12].

Mumps supports the normal flow control operations: `If`, `Else`, `For`, `$S` (select, case), `Quit` (return, break, continue), `Goto`, `Hang` (sleep), `Halt` (exit), `Job` (fork), and logical operators `&` (and) and `!` (or). Evaluation order, unless parenthesized, is strictly left-to-right, even for arithmetic. Nearly every control operator accepts a conditional expression (`:x`) that determines whether the statement is evaluated, as an `if` shorthand. Void functions are called with `Do`, which can also be used to create a new stack-level, like C's curly-braces. Functions that return values are prefixed with `$$` when called. Each operator may also be abbreviated with a single-character name for brevity, `i` for `if`, `f` for `for`, etc.

Mumps's does not enforce strict typing rules. Every value is implicitly a string, though any value may be coerced to a number by using it in arithmetic. Numbers are parsed from strings going left to right, stopping at the first non-numeric character (excluding "e", for scientific notation). Of special interest here is the special variable, horolog (`$H`), which is made up of the current number of days since December 31st, 1840, a comma, and then the number of seconds since midnight, today (`$H = 65499,4425`). The current date can thus be extracted from the horolog in a single statement, `s today=+$H`, or an empty ("null") or non-numeric variable can be trivially coerced to zero, as in `s zero=+""`.

Mumps uses stack-based dynamic scoping, with aliasing: every function exports its declared variable names to any called child functions. The `New` and `Kill` operators can be used to modify the current symbol table and redefine existing names. New creates a new symbol table at the current stack frame, if one does not already exist, and then creates an empty entry in that symbol table for the variable's name, replacing any existing entries. Kill removes the entry for the variable in the current stack frame. Future attempts to access that variable will raise an undefined variable error. Importantly, given the nature of dynamic scoping, new symbol tables are not created at each new stack-level, but only when explicitly created with New.

Mumps stores data in a hierarchical database where each root node is a hat-prefixed "named global," like "^Patients" or "^ORDERS". Subnodes are stored in a multi-dimensional comma-delimited array. Local variables may also have any number of subnodes but must be passed by reference, indicated with a preceding dot instead of the usual ampersand. For example, the "Address" global might have one entry for each home address, and be accessed through a comma-delimited notation.

<Address> =

```
set ^Addr("Street")=2
set ^Addr("Street",1)="2450 2nd Street"
set ^Addr("Street",2)="920 Ridge Street"
```

The data would be represented in the database like the following.

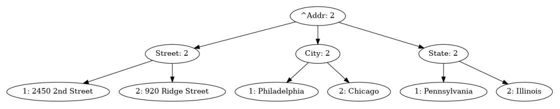


Figure 6: Database-level Address Global Layout.

Finally, Mumps supports indirect execution through the `eXecute` (eval) statement, which interprets arbitrary code. Mumps also allows indirect access to globals via @notation. In the above example, if `x = "^Addr"` then `@x` would evaluate to 2. If `y = "city"` then, `@x@(y,1)` evaluates to "Philadelphia".

A listing for 99-bottles-of-beer follows.

<99-bottles> =

```
bottles99()
  new btls
  set btls("max")=99
  set btls("min")=0
  for btls=btls("max"):-1:btls("min") do
  . write:btls>0 $$multipleBtls(btls),!
  . quit:btls>0
  . write !,"No beer!"
  quit

multipleBtls(bottleCount)
  new lyric
  set lyric=bottleCount_" "
  ; "1:" indicates the default case.
  set lyric=lyric_$S(bottleCount=1: \
    "bottle",1:"bottles")
  set lyric=lyric_" of beer on the wall."
  quit lyric
```

Many of the rules above can be bent or broken using more advanced features of Mumps, like `$QUIT`, which is true when the current code block was called as an extrinsic function (`set x=$$f()`), but false when called as an intrinsic procedure (`do f()`). This allows for magnificently inscrutable statements like `QUIT:$QUIT $QUIT QUIT`, which overloads the function's return value and always exits a function, but, returns true only when the caller expects a value.

References

[1] Advani, A., Tu, S., O'Connor, M., Coleman, R., Goldstein, M. K., and Musen, M. Integrating a modern knowledge-based system architecture with a legacy va database: the athena and eon projects at stanford. In *Proceedings of the AMIA Symposium* (1999), American Medical Informatics Association, p. 653.

[2] Bender, D., and Sartipi, K. HL7 FHIR: An Agile and RESTful approach to healthcare information exchange. In *Proceedings of the 26th IEEE international symposium on computer-based medical systems* (2013), IEEE, pp. 326–331.

[3] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 335–350.

[4] CloudLab. Hardware: CloudLab Wisconsin. https://docs.cloudlab.us/hardware.html#%28part._cloudlab-wisconsin%29, February 2020. [Online; accessed 29-April-2020].

[5] Fidelity National Information Services, I. GT.M Programmer's Guide. http://tinco.pair.com/bhaskar/gtm/doc/books/pg/UNIX_manual/pg_UNIX_screen.pdf, December 2019.

[6] Fry, E., and Schulte, F. Death by a Thousand Clicks: Where Electronic Health Records Went Wrong. https://fortune.com/longform/medical-records/, Mar 2020.

[7] HL7. FHIR v4.0.1. https://www.hl7.org/fhir/, 2019.

[8] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX annual technical conference* (2010), vol. 8.

[9] Intel. InterSystems and VMware Increase Database Scalability for Epic EMR Workload by 60 Percent with Intel Xeon Processor E7 v3 Family. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/epic-intersystems-vmware-paper.pdf, April 2015. "[Online; accessed 22-April-2020]".

[10] Kalra, Dipak. Electronic health record standards. *Yearbook of medical informatics 15*, 01 (2006), 136–144.

[11] Martín, L. F. Gnu health: A free/libre community-based health information system. In *Proceedings of the 12th International Symposium on Open Collaboration Companion* (2016), pp. 1–1.

[12] Newman, R. D. MUMPS Documentation. http://mumps.sourceforge.net/docs.html, 2003.

[13] Noll, J., Beecham, S., and Seichter, D. A qualitative study of open source software development: The open emr project. In *2011 International Symposium on Empirical Software Engineering and Measurement* (2011), IEEE, pp. 30–39.

[14] Parmar, A., Baum, S., DeArment, A., Dietsche, E., Truong, K., and News, K. H. Why electronic records didn't eliminate medical errors. https://medcitynews.com/2016/03/ehr-eliminate-medical-errors/, Mar 2016.

[15] Systems, E. Epic USCDI on FHIR. https://uscdi.epic.com/, 2020.

[16] The Office of the National Coordinator for Health Information Technology. U.S. Core Data for Interoperability - 2020 Version 1. https://www.healthit.gov/isa/united-states-core-data-interoperability-uscdi, February 2020.

[17] Walters, R. *M Programming: a Comprehensive Guide*. Digital Press, Boston, 1997.

[18] Weant, K. A., Bailey, A. M., and Baker, S. N. Strategies for reducing medication errors in the emergency department. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4753984/, Jul 2014.

[19] Wikipedia contributors. MUMPS — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=MUMPS&oldid=949599996, 2020. [Online; accessed 12-April-2020].

[20] Wikipedia contributors. RxNorm — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=RxNorm&oldid=941723876, 2020. [Online; accessed 26-April-2020].

[21] Wolfe, B. A., Mamlin, B. W., Biondich, P. G., Fraser, H. S., Jazayeri, D., Allen, C., Miranda, J., and Tierney, W. M.

The openmrs system: collaborating toward an open source emr for developing countries. In *AMIA annual symposium proceedings* (2006), vol. 2006, American Medical Informatics Association, p. 1146.