DISSERTATION

SCALABLE VISUAL ANALYTICS OVER VOLUMINOUS SPATIOTEMPORAL DATA

Submitted by

Ryan Stern

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2018

ABSTRACT

SCALABLE VISUAL ANALYTICS OVER VOLUMINOUS SPATIOTEMPORAL DATA

Visualization is a critical part of modern data analytics. This is especially true of interactive and exploratory visual analytics, which encourages speedy discovery of trends, patterns, and connections in data by allowing analysts to rapidly change what data is displayed and how it is displayed. Unfortunately, the explosion of data production in recent years has led to problems of scale as storage, processing, querying, and visualization have struggled to keep pace with data volumes. Visualization of spatiotemporal data pose unique challenges, thanks in part to high-dimensionality in the input feature space, interactions between features, and the production of voluminous, high-resolution outputs.

In this dissertation, we address challenges associated with supporting interactive, exploratory visualization of voluminous spatiotemporal datasets and underlying phenomena. This requires the visualization of millions of entities and changes to these entities as the spatiotemporal phenomena unfolds. The rendering and propagation of spatiotemporal phenomena must be both accurate and timely. Key contributions of this dissertation include: 1) the temporal and spatial coupling of spatially localized models to enable the visualization of phenomena at far greater geospatial scales; 2) the ability to directly compare and contrast diverging spatiotemporal outcomes that arise from multiple exploratory "what-if" queries; and 3) the computational framework required to support an interactive user experience in a heavily resource-constrained environment. We additionally provide support for collaborative and competitive exploration with multiple synchronized clients.

TABLE OF CONTENTS

ABSTRACT ii

LIST OF TABLES v

LIST OF FIGURES vi

1 Introduction 1
- 1.1 Research Challenges 1
- 1.2 Research Questions 2
- 1.3 Approach Summary 3
- 1.4 Contributions 6
- 1.5 Dissertation Organization 6

2 Related Work 8
- 2.1 Data Processing Frameworks 8
- 2.2 Aggregation with Data Cubes 10
- 2.3 Approximate Queries and Sampling 11
- 2.4 Spatiotemporal Data Management and Analytics 14
- 2.5 Visualization Software 15
- 2.6 Visualization of Large Graphs 19
- 2.7 Improving Visualizations 20
- 2.8 Systems Dealing with Disease and Outbreaks 21
- 2.9 Predecessors of this Work 23
- 2.10 Gap Analysis 24

3 Visualization of Disease Outbreaks 25
- 3.1 Sonata Interface Overview 25
- 3.2 Presenting Predictions: Outbreak Map 27
- 3.3 Presenting Predictions: Plots and Graphs 35
- 3.4 View Configuration and Management 39

4 Timeline and Parameters 44
- 4.1 Timeline and Branching 44
- 4.2 Scenario Parameters 49
- 4.3 Parameters Menu 55
- 4.4 Timeline Parameter Summary 61
- 4.5 Cadence Parameters Study 63

5 National-Scale Model 72
- 5.1 Single National Model 72
- 5.2 Cooperative Regional Models 73
- 5.3 State Loading and Selection 84
- 5.4 Tiling and Clustering Herds 87

6 Computation and Model Evaluation 94
6.1 Data Computations 94
6.2 Computation Evaluation 97
6.3 Outbreak Prediction Models 106
6.4 Paarlberg Economic Model 117
7 Interaction Graph 119
7.1 Graph Structure 119
7.2 Graph Production 123
7.3 Graph Queries 127
8 Collaboration in Sonata 129
8.1 Synchronizing Multiple Clients 129
8.2 Future Work 134
9 Conclusions 136
References 139

LIST OF TABLES

2.1 An analysis of gaps in existing solutions compared to this dissertation. 24

4.1 A summary of the seven tasks given to participants. 65
4.2 Shows the average researcher-assigned success scores for participants 67
4.3 Shows the 50th and 99th percentile of completion times 68
4.4 Shows the average survey scores provided by participants for how successful 69
4.5 Shows the average survey scores provided by participants for how quick 69
4.6 Shows average researcher-assigned performance scores 70

5.1 Depicts latent event count partitioning 82

6.1 Loading times in two browsers 115
6.2 Model evaluation times in Chrome using various computation configurations 116
6.3 The frequency and duration of various events 117

7.1 Shows the growth in file size for Interaction Graphs 122

LIST OF FIGURES

2.1 Cadence's user interface. 24

3.1 Sonata's user interface. 26
3.2 An example map view. 28
3.3 An early comparison between a simulation replay 31
3.4 An accuracy comparison comparing a scenario replay 32
3.5 A heat map plot produced by ADSM using data from 500 simulation runs 35
3.6 An example line chart view. 36
3.7 An example stacked bar chart view 37
3.8 An example table view. 38
3.9 The view configuration menu. 40
3.10 The view management system allows views to be placed as the user desires. 42
3.11 A user drags the Oklahoma line chart to place it on the left side of the map. 43

4.1 The timeline interface. 46
4.2 The timeline interface with three branches. 47
4.3 Examples of the interface for three input types. 52
4.4 The parameters ribbon menu 56
4.5 An early version of the parameters menu 58
4.6 Many missing parameters in the testing category 60
4.7 The parameter values summary 61

5.1 Depicts sector boundaries for five states 75
5.2 Sonata depicts the source and destination of infection events 78
5.3 A map view with tile boundaries enabled 89
5.4 Mean Shift clustering in action 91
5.5 The id hierarchy for an entity selected from Colorado, zoom level 7 to 10 92

6.1 Evaluation activity using our first computation scheduling algorithm 98
6.2 Evaluation activity using our current computation scheduling algorithm 102
6.3 The scheduling lifecycle of a computation 103
6.4 Timeline branches (above) versus computational branches (below) 105
6.5 The inner workings of a gradient boosting model 109
6.6 A plot showing diverging intermediate predictions 112
6.7 The branch setup for a large parameter change 116
6.8 Predictions from the economic model based on the losses suffered during an outbreak 118

7.1 Interaction Graphs depicting infection events from one source herd 121

1 Introduction

Visual analytics is a critical tool for understanding the patterns in large and complex datasets. In spite of this, notable challenges are posed by the need to visualize ever increasing data volumes, thanks in large part to enhancements in computing and storage capabilities. These challenges are even more apparent when interactive and exploratory analysis is desirable, as is often the case when working with spatiotemporal data.

Encouraging analysts to immediately and continuously experiment with the data through interaction and exploration keeps them engaged and fosters beneficial behavior that would not be present in query-now, review-later environments. Importantly, by providing different ways of looking at the data and encouraging real-time interaction through techniques including brushing and linking, panning and zooming, and rapid query refinement, we can help foster experimentation leading to insights and new discoveries.

The goal of this dissertation is to develop a framework capable of interactive, exploratory, and scalable visual analytics over voluminous spatiotemporal data. In addition, we aim for the framework to be suitable for a wide range of spatiotemporal phenomena. Accomplishing this involves a number of challenges and tradeoffs that will be discussed throughout the remainder of this dissertation. Our methodology addresses several of these issues with the objective of preserving timeliness and fidelity in resource constrained environments, all in support of rendering complex spatiotemporal phenomena. In the following sections, we outline the challenges and research questions that shaped this research effort, followed by a summary of our approach and contributions.

1.1 Research Challenges

Our goal is to support the interactive and exploratory visual analytics of voluminous spatiotemporal data. It is therefore essential for our solution to provide timeliness and fidelity of visual results, and we face a number of challenges in doing so. The root of several challenges is the data scale. Simply put, the datasets involved are voluminous and constantly growing, so the effort needed to extract phenomena increases accordingly.

As the datasets are also spatiotemporal, the solution needs to visualize both the geospatial and temporal dimensions of the data. In such a setting, the geospatial scope is variable and can span county, national, or even continental scales. There is also temporal variation, notably in the availability of temporal data points across different geographic locations. Alongside this, our solution needs to be able to effectively capture and reproduce the actual spatiotemporal phenomena that the dataset is recording. We need to be able to do this at a variety of scopes, as there is a great deal of variability in the spatial and temporal scopes for the phenomena that we consider. We must be able to patch together spatial and temporal data from multiple sources to visualize larger scopes.

Associated with these spatiotemporal phenomena are entities that influence and are influenced by unfolding events. Not only is the number of entities extremely large, but the number of states they can be in is as well. We must be able to faithfully reproduce these states, as well as be able to directly compare and contrast differences in these states in response to various visual queries. Comparisons may take place over differing temporal or spatial scopes.

Finally, our solution needs to be deployable on a wide range of devices, meaning that we must mediate all of the challenges above within an environment with constrained resources. It is critical to note that this work will not only require quantitative improvements, but also qualitative: our tool must be usable both in terms of promptness and intuitiveness.

1.2 Research Questions

The requirements of this system and the challenges that arise lead us to formulate the following research questions:

1. How can we support visual analytics of spatiotemporal phenomena where the number of entities being rendered is extremely large? *Metrics used to assess the suitability of the solution to this include: memory consumption, rendering time, rendering latency, bytes downloaded, download time.*
2. How can we visualize evolution of spatiotemporal phenomena at different temporal scopes? *Metrics used to assess the suitability of the solution to this include: memory consumption, rendering time, rendering latency, number of models evaluated.*
3. How can we effectively support visual analytics that support contrasts and comparisons of spatiotemporal outcomes? *Metrics used to assess the suitability of the solution to this include: memory consumption, processing time, number of models evaluated, rendering time, rendering latency, update latency, interaction rate, interaction duration.*

1.3 Approach Summary

We explore the aforementioned research questions and the suitability of our methodology in the context of Sonata, a new what-if style planning tool we have developed. Sonata enables interactive spatiotemporal visualization and exploration of simulated disease outbreaks (the phenomena) in livestock populations (the entities). The research is funded through a grant from the Department of Homeland Security, and will be used by various state and federal agencies, such as the USDA.

Although many visualization environments are experiencing the strain from increasing data volumes, an area in particular need of improvement is the interactive and exploratory visualization of spatiotemporal data produced through simulation. Like most data sources, modern simulation software is trending towards the production of higher quality, higher resolution outputs. Ultimately, this means that simulations consume more resources to run, take longer to complete, and produce significantly more output data. Due to the high dimensionality of the input parameter space combined with the stochastic nature of most simulations, visualizing such complex datasets provides an excellent proving ground for the methodologies developed as part of this dissertation. There are several systems in existence (a sampling included in Sections 2.5 and 2.8) which allow users to configure a simulation with a certain set of input parameters, run it, and then visualize the results when all is finished. Obtaining results in this manner takes time and renders interactivity and exploration ineffective, losing the desirable benefits. Our methodology, however, looks at datasets with a large input space as a prediction problem, allowing us to preserve the timeliness of results.

Sonata relies on data from existing outbreak simulators to predict the spread and control of diseases in livestock populations. The North American Animal Disease Spread Model (NAADSM)[1] and its successor ADSM are tools used for this purpose. The tools rely on configuration provided through scenario files, which expose a wide variety of outbreak parameters. These parameters control aspects of the simulation ranging from disease behavior, control zones, and farm/herd types. When run, the tools stochastically simulate the outbreak and produce outputs reflecting the state of the herds on each simulated day. To preclude the need for on-demand simulation runs at query-time, we pre-execute millions of simulation runs such that the input space is sufficiently covered and then train machine learning models to predict the produced outputs. The simulation data that our tool works with complements this research well: the dataset is large in scale (several million files consuming hundreds of terabytes), it is sensitive to spatiotemporal events and interaction, and it deals with hundreds of thousands of entities. Furthermore, the dataset poses interesting challenges thanks to its stochastic nature and high dimensionality that make the proposed solutions more applicable to a wide range of general problems.

Sonata is a visual analytics tool, so it would of course be incomplete without the ability to visualize predicted outcomes. Sonata constructs visualizations using predictions from machine learning models as well as data derived from those predictions. Sonata tells a compelling story by composing multiple views of the data, with each view depicting events in a distinct way. As such, Sonata includes maps, line charts, bar charts, and tables. Composing visualizations from multiple parts enables a complete view of the spatiotemporal phenomena, even though individual views may only focus on certain events or depict only a subset of dimensions.

Sonata makes predictions for every day during an outbreak. To do so, it relies on producing intermediate predictions that feed back into the models on the following days. To organize these daily predictions, Sonata features a timeline interface. Using this interface, users are able to interactively advance and reverse the flow of time, automatically updating views to match. To influence the predictions themselves, users set input parameters. The values of these parameters are able to be adjusted from day to day, selecting days to change from the timeline. Values set on a given day are inherited by future days until the value is reassigned by the user. This introduces functionality not available in the original simulation, which only uses one value for each parameter. To enhance the exploratory power of setting parameters, users are able to split the timeline, in turn creating timeline branches. By configuring parameters differently on separate branches, users are able to directly compare and contrast their influence.

Sonata is capable of predicting national-scale phenomena, but it does not do so by running a single set of models trained on national data. Since behaviors can vary greatly by location, we instead partition the data geospatially into sub-state chunks that we call sectors. A set of models is trained specifically for each sector, allowing us to present aggregated predictions at various geospatial scopes: sector, state, regional, or national. The sector approach has the advantage of being able to predict local outbreak activity. From these local predictions, we reconstruct entity-level activity for the purpose of visualization. Interactions between entities are predicted using probabilities captured from simulation runs by our Interaction Graph. Since the entity population is collectively on the order of millions, we tile and cluster population and interaction data, enabling Sonata to load a subset of data relevant to the current visualization.

We have opted to build Sonata using web technologies. While this has the benefit of allowing the tool to run within a web browser already preinstalled on a majority of modern devices, it also imposes additional processing, memory, and data limitations. To cope with this environment, our approach makes use of intelligent model evaluation with output dependency-tracking and caching. Views subscribe to computations, which in turn subscribe to other data computations building a complex web of dependencies. These dependencies ensure that we run computations in a safe order and then only when needed by one or more views. We make use of WebWorker threads to evaluate models in parallel.

Different people have their own way of exploring problems and each bring different expertise to the table. This is especially the case when people from different departments are working together, as will be the case when Sonata is used for planning sessions. For this reason, we have incorporated collaboration functionality into Sonata, making it a multi-user experience. Users are able to join collaborative sessions where they are able to share timeline branches and parameter sets with their peers. When in a session, visuals are synchronized between all participants allowing everyone to contribute to the exercise at hand.

1.4 Contributions

This research enables interactive and exploratory visual analytics of voluminous spatiotemporal datasets. In line with this, our work enables systems to preserve timeliness and fidelity in resource constrained environments, a must for exploratory visual analytics. To support these requirements, our methodology provides the mechanisms needed to enhance predictions originating from machine learning models and other sources in order to create compelling visuals that accurately reproduce recorded spatiotemporal phenomena. Additionally, this work enables the rapid and direct comparison of multiple spatiotemporal outcomes at a variety of spatiotemporal scopes. This research is applicable in the visualization of a wide variety of natural and man-made spatiotemporal phenomena. This can include phenomena such as planning city traffic, predicting climate phenomena, exploring supply-chain bottlenecks, and in our case limiting disease outbreaks.

1.5 Dissertation Organization

This dissertation is organized as follows: In Chapter 2, we summarize works related to and pre-dating our research effort. We also explore the gaps in these works relevant to our requirements. In Chapter 3, we provide a top-level overview of Sonata's interface and Sonata's visualization capabilities as experienced by an end user. We have organized this dissertation so that readers have an opportunity to become acquainted with the end goals and requirements of Sonata before we reveal the technical framework that makes the tool possible. In Chapter 4, we explore Sonata's method of exposing temporal controls, the timeline and parameter management. Following this in Chapter 5, we explore the techniques used to support geospatial phenomena at the national scale. In Chapter 6, we take a look at Sonata’s computation and prediction framework, and explore how we achieve interactive speeds while balancing limited browser resources. Afterward, in Chapter 7, we explore the methods used to facilitate accurate reproduction of the various interactions occurring between millions of entities. In Chapter 8, we explore how we incorporated collaborative features into Sonata, allowing multiple users to cooperate (or compete) toward the successful completion a common goal. Finally, we conclude this dissertation in Chapter 9.

2 Related Work

There are a variety of approaches that support the querying and visual components of large-scale spatiotemporal visual analytics. In this section, we will look at some of the processing approaches currently available for use by querying and visual analytics systems as well as some of the work done to improve the quality and understandability of visualized results. In addition, we include a sampling of visual analytics systems and explore gaps in their functionality compared to our proposed approach.

2.1 Data Processing Frameworks

Data processing systems are the backbone of any visualization system, providing the data to be rendered in response to an analyst's query. Big-data scale data-processing comes in three fundamental types: batch, stream, and micro-batch. Alongside these types, systems make additional use of a variety of methods to short-circuit and optimize the data-access process.

Apache Hadoop [2], an open source MapReduce implementation, is an example of a batch processing system. In the MapReduce paradigm, input records are fed into a map function as (key, value) tuples resulting in new (key, value) tuples. These outputs are then partitioned by key and fed in as a collection to an associated reduce function that once again produces zero or more (key, value) tuples. Since the output of one MapReduce job can easily be fed into a subsequent job, a wide variety of tasks ranging from computing simple aggregates to updating Google's web search index can be accomplished using this approach. Despite this, batch processing is not well suited for backing live exploratory visualizations. Historically, batch jobs were intended to be non-interactive and long running, processing a large number of inputs with a single run of a program that was designed to make use of computing resources even during nighttime hours. Furthermore, ad-hoc exploratory queries require the development a new MapReduce program for each new kind of query. Systems like Pig [3] alleviate this by converting queries written in languages such as *Pig Latin* into MapReduce jobs, simplifying the development workload and reducing mistakes.

Stream processing systems tend to be more dynamic than batch systems. MillWheel [4], an example of such a system, allows a processing task to be organized as a directed acyclic graph of stream computations that operate on (key, value, timestamp) tuples. The user can specify a key extraction function for each source to specify how incoming events will be keyed, giving access to key-specific persistent storage that allows for stateful processing needed for operations such as aggregation. Changes to the compute graph can be made at run-time, allowing the creation, reordering, replication, and removal of computations. This is useful for exploratory querying, as analysts can rapidly adapt and reissue queries without needing to restart the stream system. Additionally, multiple queries can be issued simultaneously; obtaining records from the same stream where appropriate. The benefits of streaming systems are not without cost, as they experience higher processing and network overhead than batch systems as events are processed and forwarded one at a time. Other issues arise when performing aggregations, as applying some aggregations one record at a time becomes a complicated process making less efficient use of key-specific storage a requirement.

Micro-batch systems are the middle ground between batch and stream processing systems, collecting events into small groups and scheduling them together. The small size of the batches allows the contents to be processed extremely quickly, on the order of milliseconds, and passed to downstream stages without the delay of batch systems. Determining how data is to be grouped into batches is a primary difference among the micro-batch systems. Google's Dataflow Model [5] generalizes the concept of windowing, the very process of grouping data. While prior frameworks separated aligned, unaligned, and session based windowing, the Dataflow Model treats each as a special case of unaligned windowing, allowing all three to be easily handled with the same system logic. This logic is backed by two simple operations: events can be assigned to windows and windows can be merged. The Dataflow Model also supports early, tentative processing. Triggers waiting on arbitrary conditions such as number of records received, wall time, or last event arrival allow computations to be run without being delayed by late events. Additionally, refinement modes including discarding, accumulating, and accumulating with retraction are available for controlling the behavior when updating tentative results.

Generalization is also a theme in Resilient Distributed Datasets (RDDs) [6], another micro-batch framework. At a high level, RDDs are a read-only collection of records with associated lineage that can only be created though deterministic operations on other RDDs or data in stable storage. A lineage graph specifies the steps taken to arrive at any given data partition, enabling the selective generation (and in case of failure, regeneration) of partitions only when required by a downstream child partition. Individual RDD partitions can remain in memory, be shared across computations, and optionally be reliably persisted to stable storage. In combination with this, the locality of partitions is tracked by the system, allowing the computation pipeline to be atomically and intelligently distributed across the cluster. Due to the generalizations provided by RDDs, a wide variety of other data processing systems along with their desirable optimizations and fault tolerance schemes can be implemented using only operations on RDDs. These systems include graph processing system Pregel [7], MapReduce Online [8], computation graph based Dryad [9], iterative systems like HaLoop [10], and non-transactional SQL used within data warehouses.

2.2 Aggregation with Data Cubes

When attempting to visualize data, especially in Big Data environments with billions or more records, it is often the case that there is too much data to visualize on a single screen. Analysts want to view the data through filters and aggregations [11], however, in order to gain a sufficiently accurate understanding of the data, enough of the data must be visible to discern trends and other patterns.

Data cubes are an n-dimensional data structure, capable of storing aggregations grouped across a variety of dimensions [12]. Each dimension, or face, corresponds to a column that could be used in a GROUPBY statement in an SQL query had the data been organized in a relational table. Data cubes are most effective when they pre-aggregate values across a wide variety of dimensions, as this drastically cuts down on future query time in exchange for a one time expensive batch pre-aggregation phase. Data cubes offer a number of core operations like slice, dice, rotate, roll-up, and drill-down that allow analysts to summarize data in different ways. Slice and dice extract a subset of the data cube, roll-up summarizes along a single dimension, and drill-down exposes more detail by traversing a dimension hierarchy. Large datasets cannot be fully captured as fine-grained high-dimensional data cubes, primarily because data cubes grow exponentially with each additional face, ultimately consuming vast amounts of memory.

Unlike traditional data cubes, where data is stored densely, NanoCubes [13] utilizes a sparse data structure that is capable of data sharing. This reduces the amount of data that needs to be stored, at the cost of overhead from pointers. As most records share the same per-dimension keys, the data structure will usually grow at slower rates as more data is seen. NanoCubes uses different pointer schemes for the spatial, categorical, and time dimensions, making it a powerful solution for aggregating spatiotemporal data. Spatial dimensions are organized as a quad-tree, categorical dimensions are implemented as flat trees, and time dimensions are implemented with sparse time-area tables.

Whether using traditional data cubes or NanoCubes, the analyst is trading memory utilization for speed and accuracy. There comes a point, however, when the memory and pre-processing cost of less-popular dimensions is overwhelming. To provide access to all dimensions, it is possible to partition and tile the data cube, storing it on disk at the cost of speed similar to imMens [14]. Alternatively, one could compute accurate aggregates on the fly (an expensive process), or settle for less accurate approximate queries.

2.3 Approximate Queries and Sampling

Storage systems are required to process ever increasing amounts of data, leading to slower queries despite the continual advancement of hardware [15]. Systems can improve the speed of querying and visualizing large-scale datasets by returning incomplete, approximate results. Fast approximate results go hand in hand with interactive visualization, as analysts often need just enough information to refine their query [15, 16]. The faster the results, the better, as results received within 100ms are perceived as instantaneous and results received within 10 seconds are perceived as belonging to the same action as the query itself. Approximate querying also allows more flexible queries. Unlike pre-computed data structures such as data cubes that must limit the number of interactions maintained due to memory and computation costs, approximate queries work primarily on the raw dataset. The queries are therefore capable of more complex operations over a wider set of dimensions, some of which would not be suitable for a precomputed data structure due to rarity of access or inability to be described as an aggregate.

Data sampling is critical for making sound statistical judgments about how an incremental approximate query is progressing. Generally, the most efficient sampling solution is to store records on disk in a random order [15]. This allows samples to be read in contiguously with the same efficiency as any other read operation using the given system. Additionally, no matter the number of records read in, the set of records processed so far will always be a true random sample of the dataset [17]. Randomly ordering the data does have drawbacks, however. For one, to get the data ordered that way in the first place requires an expensive batch shuffle operation, although this is a one-off requirement for a static dataset. In addition, random ordering tends to not work well with indexing structures, potentially limiting the optimizations that can be applied to very selective queries. This is a notable problem, considering that the more selective the query, the slower it will converge.

In the case of stream processing systems, random access to records is not necessarily available. One solution to this is to use reservoir sampling to probabilistically replace values in the sample with new ones over time [18]. Such a sample would start out very poor leading to highly inaccurate results compared to a true random sample. As records in the sample become more random over time through replacement, confidence about the current query result becomes more accurate.

For queries that are very selective, it is hard to obtain matching records using random samples. A typical solution would be to use an index, but index structures tend to require sorted data. The ACE tree [17] attempts to remedy this by allowing samples of a range to be taken using the index itself. It accomplishes this by assigning random pages and page sections to each record to randomize the order. The assigned sections are restricted so that only values falling within a section's specified range are permitted, with range sizes growing exponentially with the section index. During query evaluation, the structure of the index tree allows relevant samples to be produced rapidly during the first few seconds, but this rate drops with time. Sections with the same index from separate pages can be appended and filtered to arrive at another valid random sample as individual sections are exhausted. Unfortunately, this data structure alone is not suitable for distributed workloads, although a separate tree could be built at each node. In order to implement sampling without replacement, the index flips flags in the tree structure, meaning that only one query can be evaluated at a time, a major drawback for today's parallel systems.

Another solution to the selectivity problem is the use of stratified sampling, as is done in BlinkDB [19]. A stratified sample is a pre-generated sample of the dataset that includes a higher proportion of rare records than a normal sample would, allowing for faster evaluation of filter and group-by queries that select those rare records while not penalizing other queries. This introduces some bias into the sample that must be accounted for during query evaluation if values are mixed, however. The bias can be calculated by tracking and storing the true occurrence rate versus the sample occurrence rate when building the stratified sample. BlinkBD attempts to balance efficiency with generality by producing multiple stratified samples of the dataset that attempt to cover the most common column sets that occur together in queries submitted to the system. The solution does not limit exploratory querying, as the system can fall back on reading samples from the full dataset for a minor performance hit.

Random sampling is not suitable for all queries. For one, random sampling cannot find outliers [16], and although satisfied sampling can make their detection more likely, it is not guaranteed. Another issue that arises is the preservation of patterns. For graph and network based datasets, the authors of [20] provide a solution by combining the act of sampling with subsequent iterative growth. This technique involves randomly sampling nodes and links from the network to act as seeds, and then growing the sample network by probabilistically incorporating nodes connected to the sample constituents. Multiple growth phases occur until a network sample appropriate for visualization is obtained. The act of incorporating neighbors allows the structure of the network to be preserved and for patterns to emerge. It should be noted that sampling in this manner does not accurately preserve many of the mathematical properties of the network, although it can still be suitable for visual analysis.

Continuing to run a query over successively larger portions of the dataset will eventually produce more accurate results as the approximate value converges on the true value. Confidence is a metric that can be reported alongside approximate results that decreases with more variance in record values and improves proportional to the square root of the number of records seen. The metric describes the degree to which the value is likely to change with further records, putting the analyst in a much better position to determine whether the cost of waiting for better results is worth the degree of improvement.

2.4 Spatiotemporal Data Management and Analytics

Spatiotemporal data management underpins effective access to data. Several systems rely on the geohash algorithm [21] to identify proximate data. The Galileo system relies on geohashes and is designed as distributed hashtable [22]. Galileo leverages geohash based partitioning to support fast ad hoc queries [23], analytic queries [24], proximity-constrained queries [25], continuous queries [26], and anomaly detection[27]. Effectively leveraging metadata in support of powerful query evaluations over scientific data collections have been explored in [28-31].

Systems such as Synopsis [32] take a different approach to spatiotemporal data management. Rather than store the data on disk, Synopsis relies on compacting data (~1000-fold) via data sketching, and pinning these sketches to memory. The system supports a diversity of queries that are evaluated over in-memory data.

An aspect related to storage of data streams is dynamic and real-time processing of streams. Buddhika et al. provide a framework for online scheduling of data streams in continuous sensing environments [33, 34]. The Swarm system orchestrates these as tasks of loosely-coupled HPC clusters with the objective of timeliness and throughput preservation [35].

Budgaga et al. describe a framework for model construction using statistical, ensemble, and machine learning frameworks [36]. By effectively performing dimensionality reduction, feature ranking, and bias-variance decomposition of model errors the framework results in specialized models that are accurate for particular portions of the feature space.

The Trident system supports a query-driven approach to model construction, and forecast generation of voluminous spatiotemporal datasets [37]. The system focuses on storage-level innovations to improve the responsiveness of analytics pipelines, analyzing multidimensional data points as they are stored to produce in-memory data structures that can be queried in real-time.

2.5 Visualization Software

Bender et al. [38] present a framework to simplify the process of implementing web-based visualizations. As an early work on web-based visualization from 2000, the framework generalized the visualization process through the use of generic modules that handle geometry storage and rendering. The authors attempted to overcome browser incompatibilities and inconsistencies (a notable issue at the time) through the use of browser plugins including Java and VRML. The framework provided access to primitive shapes, like lines, triangles, prisms, cones, and spheres so that developers could create a variety of unique, custom visualizations. The framework also utilized a custom geometry description format that could easily be adapted for use with pluggable rendering components without risking the format being obsoleted by a future software upgrade.

In the years following [38], most visualization frameworks became accustomed to providing a limited set of preprogrammed visualizations such as the histogram and pie chart that were difficult to adapt and extend. Protovis [39] attempts to enable more generic visualizations on the web by once again focusing on the production of complex visualizations through the composition of a small set of low-level shapes, in this case called marks. Protovis provided a custom JavaScript API for implementing visualizations, allowing the framework code to work around browser inconsistencies. Should the provided marks not be sufficient, Protovis takes advantage of JavaScript prototyping, allowing new marks to be developed by the user.

Data Driven Documents [40], or D3, is the successor of Protovis and the current state of the art. D3's contribution was changing the focus from drawing marks tuned by data to using data to drive arbitrary changes in the document object model (DOM). Given a set of data, D3 allows modification of DOM elements corresponding to the data entries in bulk, modifying all relevant elements with each method call. The JavaScript API is generalized and vastly simplified, small enough to be learned in minutes. The core API provides shorthand operations that apply generally to all HTML and SVG elements, and future improvements to the HTML and SVG standards will be usable immediately without needing to update D3. Being a web standard, the attributes and behavior of SVG elements are very well documented, making them better candidates for building abstract visualizations than higher-level marks. Critically, D3 allows visualizations to become more interactive by managing browser event handling, transitions, and animations with a similarly lightweight API. D3 is not an out of the box visualization framework, and requires development time to build visualizations. Other developers can, of course, provide out of the box plotting solutions utilizing D3.

Map Cube [41] is a visualization system created to help display the aggregates commonly stored as data cubes. In Map Cube, visualizations are pre-rendered, a process that takes place as the data cube is being computed. This process results in an album of map graphics with regions colored based on the aggregate values across various dimensions. The number of maps produced is large, growing exponentially in number with more dimensions, just as data cubes do in size. These maps are then displayed to analysts either as a table or cube. The analysts can specify the dimensions to view as well as slice and dice the visualization to cut down on extraneous information. A benefit of the system is that the maps built from complete data are able to be displayed immediately, supporting interactive exploration that analysts can immediately trust. It needs to be noted, however, that computing the images is an expensive, offline process. Should an analyst want to examine a dimension not included in the album or should the data be updated, then they will potentially have to wait hours for new images to be generated.

The visualization of cubes becomes challenging as the number of dimensions increases and the ability to display all dimensions on a 2-d screen becomes less straightforward. Like Map Cube, Polaris [42] combats this problem by organizing multiple graphics into a two-dimensional table. While Map Cube displays static graphics in a table, however, Polaris computes the visualizations at runtime from multiple database queries. The system makes use of the PivotTable interface to logically rotate data and encode different dimensions across rows and columns of the table. One of the goals of Polaris is to facilitate exploratory analysis by allowing analysts to create multiple views of the data simultaneously. Data shared between these views can be highlighted using brushing and linking, a technique where data selected on one view is selected in another view. In addition, analysts are permitted to rapidly drill up/down, change dimensions, and alter plot types for each view through the user interface. This is not without limitation: while users are permitted to customize graphics by customizing mappings for shapes, colors, sizes, and orientations per dimension, analysts are unable to add completely new chart types. This limits the power of Polaris, as new chart types can expose relationships that are not apparent in other graphics. Polaris does not make use of precomputed aggregates stored within a data cube, as individual cells in the display have the potential to have overlapping relations. This means that separate queries need to be issued for each cell in the display, a critical inefficiency. Additionally, Polaris attempts to draw one mark for every available data record. Large amounts of data both clutter the display and slow response times, meaning that Polaris must work on a subset of the data.

ImMens [14] is another system that operates on aggregates stored in data cubes. The system tiles the data cubes using a tiling scheme similar in concept to the map tiles downloaded from interactive map services, except that the data to produce dynamic visualizations is stored instead of pre-rendering an image. To produce the tiles, the full data cubes are decomposed into subcubes of four dimensions. The choice of four dimensions is not a requirement, but it permits the tiles to remain small enough to be loaded and processed on a GPU. Four-dimensional subcubes are also suitable for supporting brushing and linking between two two-dimensional plots. There are limitations, however, many similar to those of Map Cube. Since the data tiles are pre-generated, an early choice of which dimensions, resolutions, and aggregations to include can severely limit which relationships can be explored in the future. Other issues that can arise include the need to regenerate some tiles when new data is added to the dataset and challenges selecting appropriate bin sizes when aggregating.

The POI Pulse [43] system visualizes spatiotemporal activity over the course of a week, highlighting points of interest within a city. POI Pulse makes use of data tiles to produce dynamic spatiotemporal visuals.

The tiles are pre-generated using time-series data collected from social media. POI Pulse then uses D3 alongside map framework Leaflet [44] to load the tiles and create vector graphics based on them, a fast process that only accesses relevant data. The visualization enables dynamic spatial and temporal panning and zooming, encouraging exploration of the data.

EDEN [45] is an interactive visual analytics framework for climate data. It is based on parallel coordinates [46], a 2-d visualization technique for multivariate datasets. It should be noted that parallel coordinates easily suffers from clutter, but the tool can arrange axes by correlation to help alleviate this. EDEN features panels for temporal range selection, geographic region selection, and variable selection. EDEN incorporates additional information into the parallel coordinates visual, including correlation indicators, variability, typical value, interquartile range whiskers, and histograms. The tool additionally provides correlation matrices and scatterplots that interactively update based on range selections in the main view. The tool provides results quickly for smaller-scale datasets, but requires the use of supercomputers for large scale datasets. EDEN performs all processing within the client, as opposed to issuing queries to a remote service, opting to use multithreading and parCAT [47] for parallelization. At the time, the tool did not support direct comparisons between queries, but the authors note this as an area of future work.

CMWeb [48] is an interactive web application for visualizing amino acid residue-residue contacts. The tool allows users to visualize sequences in the Protein Data Bank (an external project), specifying a desired contact prediction method. Requests made in the client are submitted and evaluated on a remote server, which produces results that can then be visualized by the client. Users may need to wait a few minutes as results are processed. Results are presented as a contact map, structure viewer, multiple sequence alignment viewer, and several plots with scores and distributions. CMWeb does not allow users to submit custom sequences, although the list of available entries is updated automatically as new sequences are submitted to the data bank.

TimeFork [49] is a what-if style tool for interactive prediction of time-series data. The tool is a mixed-initiative [50] system, combining contributions from the user and predictions from machine learning models to form more accurate final predictions. In TimeFork, the system presents an initial prediction and presents the user the option to manually correct the prediction, select a correction from a set of options, or accept the prediction. Any changes made act as a what-if query, resulting in interactive predictions for the provided time-series. StockFork, a stock exchange themed implementation of TimeFork, presents the time-series data as line charts with Bollinger bands for stock prices, color coded bar charts for trade volume, and a node-link diagram for correlations between stocks. Analysts update trends by clicking on the line charts. A self-organizing map is then used to provide quick conditional predictions informed by these trends.

2.6 Visualization of Large Graphs

GreenMax [51] aims to provide real-time interactive visualization of large small-world graphs. Due to the data scale, visualization algorithms are selective about which graph elements are displayed and as a side effect tend to destroy or distort the graph's structure. The Walshaw layout algorithm [52] reduces this effect by constructing a multilevel layout. To preserve the structure, the algorithm merges vertices and removes extra edges, coarsening the graph at each zoom level. Once a threshold is reached, the remaining vertices are then positioned and graph detail is restored level by level in reverse. GreenMax adapts the Walshaw algorithm to support graphs with highly connected vertices, like those seen in small-word graphs.

GraphMap [53] is a graph visualization tool that takes inspiration from online tile-based street-mapping services to reduce clutter and disorientation. To do so, the system creates a mesh surrounding the vertices and routes the original edges via the shortest path through this mesh. Using data and raster quad-tiles, GraphMap renders vertices prioritized by an importance metric and the mesh segments that connect them, revealing more at each additional zoom level. The use of a mesh prevents the shape of the paths from being distorted across levels while still being able to gradually reveal finer details.

Dadi et al. [54] describe memory management techniques when dealing with Keyhole Markup Language (KML), an XML based standard for geospatial data. To reduce memory consumption, large KML files are petitioned by region and level of detail. They then make use of existing KML tags to implement quad-tree functionality, loading a hierarchy of linked KML files relevant to the current visualization. Furthermore, they describe mixing vector data with pre-rendered images, favoring images when there are a large number of features that are too small to interact with at a given level of detail.

2.7 Improving Visualizations

When working with visualization systems, it is not enough to only focus on algorithmic improvements: we must also be mindful of how users experience the results produced by the system. Work on ThemeRiver [55] exposes how making the most of how viewers perceive objects on the screen can greatly improve a user's understanding of presented data. Humans organize objects into groups automatically, being influenced by similarity, continuity, symmetry, proximity, and closure. When the visualization brings elements together in a way counter to these ideals, the region stands out. Whether or not this is a good thing depends on the goals of the visualization designer; to break similarity by using a differently colored highlight, for example, may be advantageous. They further discuss the concept of a visual metaphor, a visual what relates to concepts that the viewer is likely to already be familiar with. Using visual metaphors help reduce the effort needed to grasp concepts relayed by the visualization and therefore help improve the exploratory experience by relaying complex relationships more effectively. In the case of ThemeRiver, the visual river appears thicker with a higher concentration of themes, analogous to more water flowing in a natural river.

A visualization has the biggest impact when it is able to reveal the structure in the data, allowing analysts to identify trends, patterns, and outliers. Clutter obscures the data's structure, introducing crowding or disorder within visuals. Discussed in [56], there are a number of approaches to tackle this problem, including dimensionality reduction (such as Principal Component Analysis [57] and Multidimensional Scaling [58]), distortion, and the use of multi-level resolution. For example, distortion methods put emphasis on certain areas of the data by providing more display space. No solution is one size fits all; the best approach depends on the type of visualization. One often overlooked method of clutter reduction is dimension reordering, which can reveal different relationships and patterns in the data simply though proximity. Determining the best ordering is an NP-complete problem, and the fastest solutions are suboptimal and greedy. A few techniques include reducing the number of outliers between dimensions, placing similar or correlated plots near each other, or sorting by dimension cardinality or plot size.

Successfully conveying the trustworthiness of data to the analyst without increasing clutter continues to be a challenge in the visual analytics world. [15] and [16] describe potential methods of relaying uncertainty, including using opacity, fuzzy boundaries, and wide intervals. For any method that modifies the visualization, the visual modifications should approach the original appearance as approximate values converge. The use of opacity or varied color runs the risk of being misinterpreted and discerning between small changes in opacity or color is difficult. The use of fuzzy boundaries and error bars is risky as it increases the amount of perceived clutter in the visual, impairing the ability to quickly draw conclusions. Widening the displayed area to represent the range of potential future values runs the risk of visually shrinking the majority of records received so far as if the data was actually low resolution. Considering the overwhelming number of downsides for incorporating uncertainty into the visuals, it seems best to keep the measure separate. One approach would be to supply two progress bars immediately under the visualization, one showing how close the approximate value is to the projected true value based on where it started, and one depicting the expected time remaining until the value converges.

2.8 Systems Dealing with Disease and Outbreaks

InfluSim [59] is a planning tool for influenza outbreaks from early 2007. The tool uses over 1000 differential equations to model outbreak behavior, providing results within a second to the planner. This approach is a middle ground between spreadsheet-based models and stochastic simulations, and has the side effect of not being well suited for small-scale outbreaks involving a small population. The system does not rely on a large data-set, and can be run successfully on a standalone desktop computer with no network access. A graphical user interface allows planners to modify equation variables with input boxes and sliders. Results are presented as tables and line charts.

EpiFast [60] is a stochastic simulation algorithm designed with the aim of studying how diseases spread through large populations of individuals as well as how policy decisions impact that spread. Within the model, time passes in discrete days. The model focuses on interactions between individuals, simulating disease dynamics over large, high-resolution, social contact networks. Policy interventions have the potential to dynamically change the contact network, activating via trigger conditions. While EpiFast does not rely on shared memory environments, the algorithm is parallelizable to take advantage of them. In the distributed setting, the system was able to finish a complete simulation covering 1.8 million individuals with 900 million contact edges within 1 to 3 hours, depending on the level of policy intervention; a single run takes 15 minutes. Some of the delay is due to the initialization phase, where the contact graph is read from disc and distributed to the processing nodes. The system makes use of in-house software called Didactic [61] to enable users to configure and submit the simulation via a web-based user interface and visualize the results as plots.

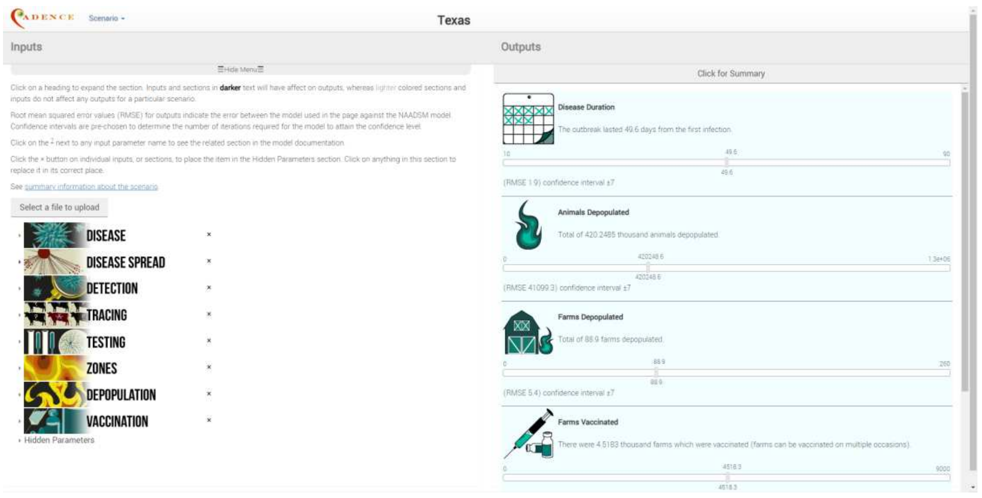
GLEaMviz [62] is an interactive spatiotemporal visualization and modeling tool designed for viewing the global spread of human transmissible diseases. The tool aims to balance complex data-drive epidemic modeling, accessible commutation speed, and flexibility in describing a scenario. They allow configuration of the simulation including compartmental models, mobility classes, transition rates, environmental effects, intervention measures, and outbreak conditions through a GUI. The tool makes use of multiple layers to incorporate a high-resolution population grid as well as short- and long-range mobility data. GLEaMviz is divided into three components: a client, proxy middleware, and simulation engine. The clients present the GUI and contact the proxy middleware to request that simulation be executed on the server. The number of stochastic runs and duration can be configured, with a single run taking a few minutes on a desktop computer. Results from single runs are provided back to the client incrementally, but users must wait for multi-run simulations to finish so that the results can undergo statistical analysis. Results are presented as geographic maps and a collection of graphs that evolve over time. Graphs are limited to presenting the number of cases over time, but the geographic region of the presented data can be specified. Infection paths are also depicted for single simulation runs.

Disease BioPortal [63] is a tool that assists animal health experts with disease surveillance. It is a web-based system that provides visualization and analysis of disease data at near real-time through information sharing, including support for phylogenetic analysis of genetic sequences related to an outbreak. Requests from the BioPortal client are submitted as a query to a remote database, which returns results to be visualized by the client. The tool supports multiple views of the data. These include a timeline, illustrations of spatiotemporal patterns, a phylogenetic tree, geographic location of cases, and plots depicting case trends. Due to the sensitive nature of the data, the tool utilizes access controls and other security techniques to maintain data confidentiality.

2.9 Predecessors of this Work

Sonata greatly improves upon the work done on Cadence [36, 64]. Cadence is a what-if style tool that enables interactive exploration of simulation parameters. The tool is presented as a webpage with sliders for scenario parameters on the left and expected final outputs on the right. As the parameter sliders are adjusted by users, the outputs on the right are instantly updated. This interactivity is made possible by the use of machine learning models encoded as JavaScript and embedded into the webpage. This is in contrast with obtaining predictions directly from the source simulation, which requires modifying the scenario file and executing the entire simulation multiple times for every small change (multiple runs are required for sufficient output coverage due to the stochastic nature of the simulation).

Cadence only makes four model evaluations per parameter change and does not have to contend with resource limitations. Furthermore, the tool is only capable of predicting the final outcome of an outbreak without any spatiotemporal context. Cadence's user interface is shown in Figure 2.1, and Table 2.1 notes core differences between the goals of Cadence and those of Sonata (as well as those of related systems).



Click on a heading to expand the section. Inputs and sections in **darker** text will have affect on outputs, whereas lighter colored sections and inputs do not affect any outputs for a particular scenario.

Root mean squared error values (RMSE) for outputs indicate the error between the model used in the page against the NAADSM model. Confidence intervals are pre-chosen to determine the number of iterations required for the model to attain the confidence level.

Click on the ? next to any input parameter name to see the related section in the model documentation.

Click the × button on individual inputs, or sections, to place the item in the Hidden Parameters section. Click on anything in this section to replace it in its correct place.

See summary information about the scenario

The outbreak lasted 49.6 days from the first infection.

10 49.6 90

(RMSE 1.9) confidence interval ±7

Total of 420.2485 thousand animals depopulated.

0 420248.6 1.3e+06

(RMSE 41099.3) confidence interval ±7

Total of 88.9 farms depopulated.

0 88.9 260

(RMSE 5.4) confidence interval ±7

There were 4.5183 thousand farms which were vaccinated (farms can be vaccinated on multiple occasions).

0 4518.3 9000

Figure 2.1: *Cadence's user interface. Four models predict the final outcome of an outbreak. Sonata greatly improves upon the capabilities of Cadence.*

2.10 Gap Analysis

Table 2.1: *An analysis of gaps in existing solutions compared to this dissertation.*

System	Platform	Data Source	Resource Consumption	Query Flexibility	Data Fidelity	Entity Rendering	Spatial Dimension	Temporal Dimension	Interaction	Exploration	Comparison
InfluSim	Exe	Model	Low	Good	Good	No	No	Yes	No	Yes	No
EpiFast	Exe	Disk/Sim	High	High	High	High	?	Yes	No	No	No
GLEaMviz	Web	Net/Sim	High	High	High	Cell	Yes	Yes	Yes	No	No
BioPortal	Web	Net/DB	Low	Low	High	Low	Yes	Yes	Low	Low	No
POI Pulse	Web	Net/Disk	Low	Low	High	Tiled	Yes	Yes	Yes	No	No
TimeFork	Web	Model	Low	Fair	Good	No	No	Yes	Yes	Yes	Yes
ADSM	Exe	Disk/Sim	High	High	High	High	Yes	Yes	No	No	No
Cadence	Web	Model	Low	Good	Good	No	No	No	Yes	Yes	No
Sonata	Web	Model	Fair	Good	Good	Tiled	Yes	Yes	Yes	Yes	Yes

3 Visualization of Disease Outbreaks

Sonata is a web-based planning tool that enables interactive visualization of disease outbreaks in livestock populations. The tool was created as a result of the research presented within this dissertation, and serves as validation of the ideas discussed within. In this chapter, we explore the evolution in techniques as we implement the most immediately visible aspects of the tool, the visualization of outbreak predictions within Sonata. In later chapters, we will explore the various internal aspects of the tool that make these visualizations possible.

3.1 Sonata Interface Overview

A lot goes into the visualization of outbreak predictions. In order to form an appreciation of the inner workings of Sonata, it is important to first have a firm grasp of Sonata's visible side: its user interface. The interface, shown in Figure 3.1, is divided into multiple sections, each with a distinct purpose. An overview of each of the sections is provided below, and more detail is provided later on in the chapters devoted to each given feature.

We start the interface tour across the top with the timeline, which provides the user with a variety of temporal information. The ticked diamonds on the timeline each represent a day. Users are able to select the diamonds, making the days they represent the target of various operations, namely setting input parameter values. The lines connecting the diamonds represent branches, which represent a connect sequence of events over time. Users can split a branch to compare the impact of different inputs, as outcomes on separate branches are allowed to diverge. Continuing on, the vertical marker represents the currently visualized day. Many of the data views either highlight the currently visualized day, or display results specific to that day. A timeline overview is provided immediately below the timeline. The overview not only shows the entire timeline, but also highlights the portion that is visible above it. Like the main timeline, selected and changed days are highlighted in color. For quick navigation, users may click a day on the overview to immediately set the currently visualized day. Additional details on the timeline are available in Section 4.1.

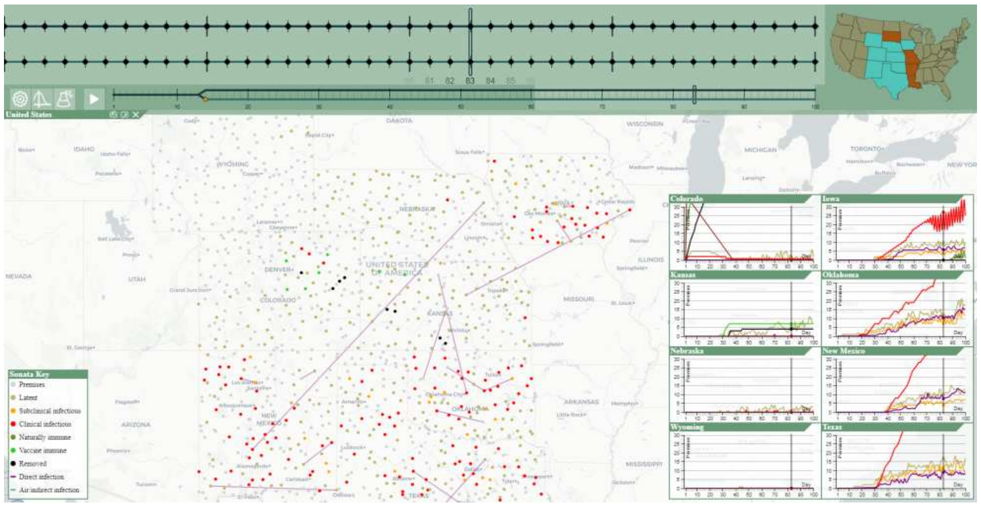


Figure 3.1: *Sonata's user interface. The timeline, state selection, and menus appear above the views. A map view showing the spatial dimension of the outbreak prediction is in the background. In front, floating line chart views for states of interest depict the temporal dimension.*

To the right of the timeline is a map depicting the United States. This map shows a variety of information including which states are available, which ones are currently loaded, and which ones are selected. The current state selection is taken into account when creating views or setting input parameters.

To the bottom left of the timeline are the control buttons. The control buttons are organized into clear groups, with one group to open menus and another group to control playback of the outbreak visualization. Starting from the left, the button with the cog icon opens the options menu. This menu allows users to load and unload state scenarios, join collaboration sessions, and configure various tool settings. Next, the button with the plot icon will open the view configuration menu. This menu gives users the ability to create and configure various views of the outbreak. Next, the button with the vial icon opens the parameter configuration menu. This menu allows users to adjust the parameters that influence the models used by Sonata to predict the events that occur during the outbreak. Changes take effect immediately. Finally, the play/pause button controls the flow of time. While playing, the currently visualized day advances at a constant rate.

Below the timeline and menus is the main view area. The views that appear in this area are configured by the user, but typically include a map and various accompanying charts. The views can be resized and rearranged, with their position governed by Sonata's view manager. These features will be described throughout the remainder of this chapter.

3.1.1 Libraries

We make use of three JavaScript libraries within Sonata. The one that we make heaviest use of is D3, standing for Data Driven Documents. D3 provides several convince methods for creating DOM elements in bulk, allowing the configuration of attributes, styles, animations and more in very few lines of code. D3 is most powerful when used to produce dynamic visualizations from an array of data, but it is not limited to this. As such, we make use of D3 not only with our maps, charts, and tables, but also for the lifecycle of nearly every HTML element that makes up the page.

We also use Leaflet [44], an open-source library for interactive maps. Leaflet provides the base functionality for our map visualizations, including converting latitudes and longitudes to pixel coordinates and managing which tiles are loaded and displayed in response to user actions. Leaflet is a performant library when using small to moderate datasets, but it seems to struggle with the scale of data that we present to it. In response, we have had to make numerous optimizations in the areas where Leaflet is used.

Our final library is RawInflate. This library is a JavaScript implementation of the zip inflate algorithm, which we use to unpack compressed models loaded from JSON files. Inflating the models is faster than downloading them uncompressed, potentially shaving off several seconds of loading time over slower connections.

3.2 Presenting Predictions: Outbreak Map

When planners first start Sonata, they are greeted by the visual cornerstone of our tool, the interactive outbreak map. This component, shown in Figure 3.2, displays the approximate geographic location of herds used within the original NAADSM and ADSM simulations over a familiar tile-based map allowing planners to grasp the geospatial scale of the planning exercise. The map is tied to the currently visualized day on the timeline, and automatically updates as this day moves forward or backward in time. The herds are color-coded according to their health state and actively change color as the outbreak unfolds, coinciding with the herds becoming ill, vaccinated, or culled. Whenever a new infection occurs, the event is highlighted on the map by simultaneously depicting the spread method, source herd, and destination herd. State changes and infection events are animated, and recent changes periodically pulse to grab the user's attention.

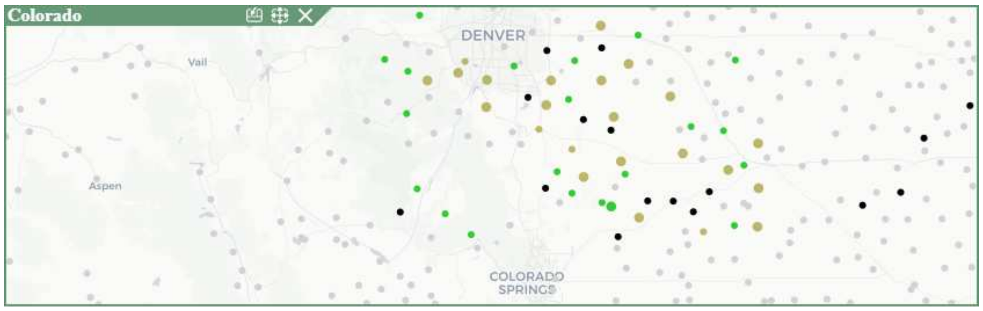


Figure 3.2: *An example map view. Backed by quad tree tiles for image and population data, users can pan and zoom any area of interest. Each dot is a herd, farm, or facility.*

We use a variety of colors to depict herd health states. We use gray for susceptible, tan for latent, orange for subclinical infectious, red for clinical infections, shades of green for immunity, and black for deceased. In addition, we use violet for direct infection events and blue for indirect and airborne events. To improve readability and understanding of the predicted results, the same colors are used across all views of the data. The tool is designed so that users can customize the colors used within the tool, allowing them to select ones that they find easiest to recognize. This functionally is especially important for colorblind individuals, as they may not be able to distinguish the colors in our default color palate. We originally used brown for susceptible herds, but user feedback indicated that it drowned out the smaller subset of herds actively involved in the outbreak.

The power of the map view is that it provides the context for the output values that are displayed in the other data views. Users are able to tell not only where the outbreak has spread, but also when it happened. The spatiotemporal nature of Sonata emerges more clearly from the map view than perhaps any of our other methods of presenting outputs. Despite this, all of the various view types work hand in hand to provide a more complete understanding of the predicted behavior of an outbreak. This is a good thing, as the outputs presented in the map view are derived from less specific predictions and are the most heavily processed.

Maintaining the fidelity of events displayed on the map has been a considerable challenge. In the upcoming sections, we explore the various approaches we have taken to reconstruct the behavior from the original simulation from simple numeric predictions.

3.2.1 Generating Map Data: First Method

Sonata must prepare information on herd states and infection events before it can display anything on the map. As it turns out, this process is not as straightforward as it may initially seem. Sonata makes use of gradient boosting [65, 66] models to predict the number of herds in each of the health states (see Section 6.3 for greater detail); we refer to these as the outputs. These models succeed at predicting the quantity of herds in each health state as well as when changes happen at the population level, but these models do not provide any information about which individual herds change state or when individual changes occur. We must construct this fine-grained information at runtime, ensuring that the result matches the behavior of the original simulation.

For the greatest flexibility and responsiveness to parameter changes, models that predict the where and when for individual herds is highly desirable. Unfortunately, the size of the herd population makes such models untenable. With over 500,000 herds in our current test set and more on the way, we would need orders of magnitude more training data points in order for the models to have a chance of acceptable accuracy; the dimensionality is simply too high. We needed some method of determining which herds should be shown as infected on the map, so in the absence of a model we introduced a custom data structure, our Interaction Graph. We dedicate an entire chapter to our Interaction Graph (see Chapter 7), but in summary the graph maintains a count of various herd events from the source simulation, including infections and state changes. The events are recorded for individual herds in the population, allowing us to resolve simple queries about the herds within Sonata. From these counts, we calculate a base probability of the disease spreading from a given herd to any of its neighbors. To keep file size low, the Interaction Graph is unable to respond to changes in input parameters. Despite this, the base probability is enough to make informed decisions about herd interactions that can be tuned with other metrics.

With this groundwork in place, we now have enough information to generate compelling map data for each day. The first step in this process is to predict the number of herds in each of the health states. We then update a copy of the previous day's map data to reflect the new day's predictions. We chose to do this stochastically, so that more of the possible outcomes have a chance to occur. We begin by iterating over the previous day's involved herds and add them to probability spinners based on which state changes are legal at the time, weighted by the number of days spent in the current state. The probability spinners consist of an array where each element (a herd) covers a range of numbers corresponding to its weight. An element's range starts where the previous element's ended. A random number from the complete range is selected; the element whose range the number falls in is returned (and optionally removed), simulating a physical spinner. For the states that are infections, we add their neighbors (via the Interaction Graph) to a spinner containing infection destinations.

For each of the output states where the predicted number is higher than the number on the previous day, we spin for herds that are allowed to change to the given state. If there are no herds available in the spinners, then we spin for new infection destinations. In the case that we are unable to spin for a new herd, a random herd is selected. Selecting an infection destination results in the outbreak visually spreading, and this case happens every time a latent herd is added. We also handle the case where the predicted value decreases from the previous day. In this situation, we spin for herds to remove from the given state, returning them to the susceptible state. Note that only involved herds are included within the map data; when data is missing for a herd, it is depicted in the initial susceptible state. Figure 3.3 depicts how out first approach looked at the time, comparing it to a replay of actual simulation output.

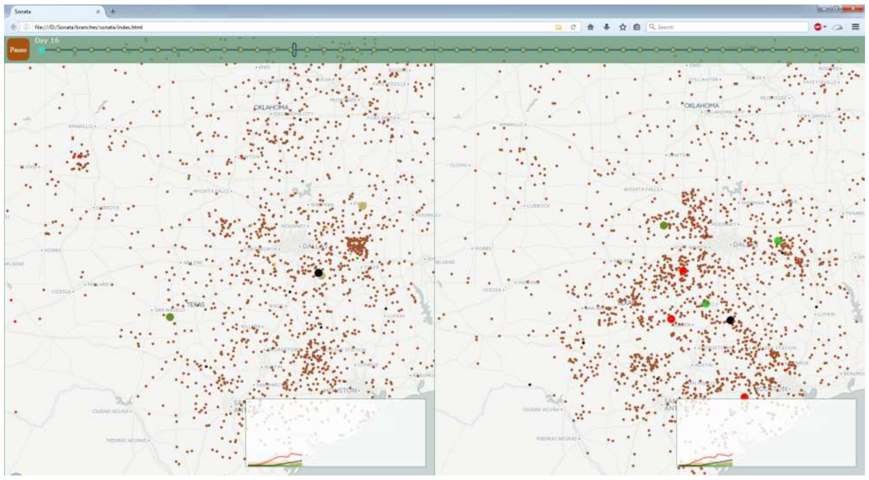


Figure 3.3: *An early comparison between a simulation replay (left) and our first approach (right). The number of herds in each state over time is similar between sides.*

There were a few issues with this design. To start, we moved herds from state to state in the same order that the simulation does. This meant that the predictions for later states had not yet pulled herds from the earlier states, so Sonata created too few herds in the earlier state and removed some that could have been advanced. Returning excess herds to the susceptible state introduced a noticeable flutter of involved herds over several days, as the model predictions would often oscillate. This flutter appears in the form of herds in more advanced states disappearing and then reappearing on a later day in new locations, breaking from expected state to state advancement. Finally, we only iterated over the herds in order to populate the spinners once. This meant that herds are never eligible for double state advancement in a single day, an event that happens on occasion in the original simulations.

3.2.2 Challenges with Depicting Vaccination

The geospatial visualization of vaccination usage has presented a number of problems, as it has behavior distinct from any other output. With our first approach, our models correctly predict the number of units that are vaccinated on each visualized day, but the locations where the vaccinations are applied had not been accurate. Specifically, the simulated results from ADSM depict the vaccination of units clustered around a previously infected unit, while our solution randomly distributes the vaccinations across the landscape. This was noticeably incorrect behavior, and fixing this visual anomaly was a priority.

This discrepancy is shown in Figure 3.4. The left half of the figure is a simulation replay, showing the expected behavior that we are trying to reproduce. The right half shows our first approach, distributing vaccinations randomly over the population. We needed to remedy this for our map results to be trustworthy, but doing so with the Interaction Graph is not necessarily the most straightforward solution, although managing to do so would make our solution more generally suited for a wider variety of problems. The issue with the Interaction Graph is that it does not respect user changes to input parameters, at least not without greatly increasing the graph file size. Since we have access to parameter values as configured by users within Sonata, the most straightforward way to implement vaccination is to use this information and randomly select herds within the specified radius directly within Sonata. The downside of this approach is, as mentioned, that adding specialized cases to our solution has the potential to reduce the overall generality of our system.

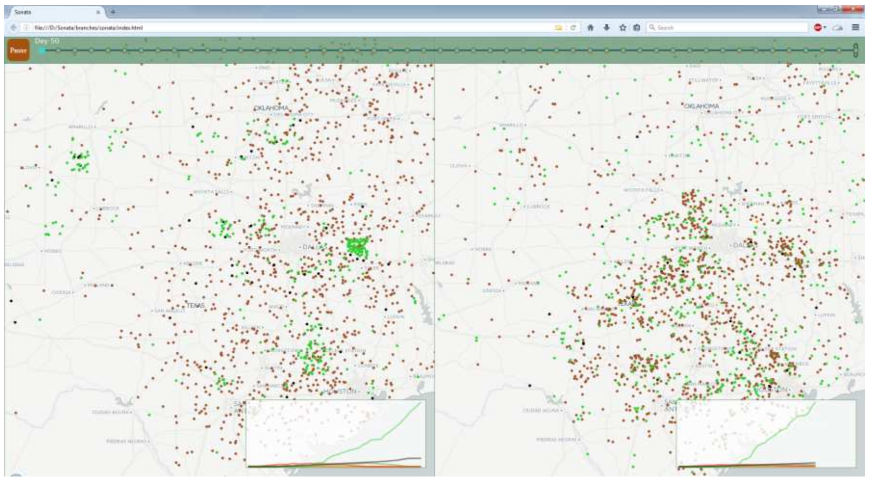


Figure 3.4: *An accuracy comparison comparing a scenario replay (left) to our approach (right). In the replay, vaccinated herds (green) appear in clusters around infection sites. In our approach, vaccinated herds are selected randomly, an issue we will resolve.*

Another issue arises when there are not enough visible units to satisfy model predictions. Due to browser limitations, we must sample the displayed herd population as browsers are unable to cope with all units at once. Anytime the model predictions exceed the number of visible units, most commonly seen with vaccination, all visualized units within the sector will be displayed as vaccinated. This leads to a significant departure from reality more serious than randomly placing a few vaccinations. We can remedy this by clustering units instead of sampling them, allowing one visualized unit to represent multiple nearby units from the full population.

3.2.3 Generating Map Data: Second Method

Our initial algorithm for generating map data did not reproduce the simulation behavior to a level that we found satisfactory. In response, we set out to adjust the algorithm in order to improve the accuracy of the map visualization. Most of the constraints from the first approach remained the same: Sonata still used machine learning models to predict how many units should be in each health state on any given day, and these models were not capable of accurately predicting which individual units should become infected or change state.

In our new approach we process the predicted counts in passes while using output-specific methods to select relevant units. We select and move units to new health states in a specific order: starting with deceased state, then vaccinated, then naturally immune, then clinically infectious, then subclinical, then latent and finally new exposures. Note that this is the reverse order of the previous approach, and also the reverse order of state changes in the source simulation. For each of the output states, we iterate over the involved herds and add them to a probability spinner if they can legally change to that state. We determine which states can legally change into others by looking up output to output pairs in a dictionary; the value in the dictionary is the base weight influencing the likelihood that the herd is selected from the spinner. The base weights are derived from the Interaction Graph; a weight of zero indicates that the change is not possible according to the simulation. This base weight is then adjusted per-herd depending on the behavior of the particular output, taking into consideration the time the herd has spent in the current state and its distance from other herds of interest.

Once the probability spinner is constructed for a given output, we use it to select state-change candidates to be advanced to that specific output. Like the previous approach, we spin for change candidates until the existing number of herds in the state matches the predicted amount. Since we are filling out the states in reverse order, we select herds in the earlier health stages to advance to the later stages before we fill the former to the predicted amounts. This produces a more natural flow from state to state within the map visualization. We also no longer reset herds to the susceptible state when the current amount exceeds predicted levels; instead we opt to let the herds be advanced to the later states naturally over time. This helps to smooth out inconsistencies between models while improving the realism of the visualization.

When assigning vaccinated states and creating exposure events, the candidate selected using the spinner is used as a source unit, with the destination determined by adding a random normally distributed offset to the source position and finding the closest unit to that point. We use K-D Trees [67] to find the closest unit to this point efficiently. Exposure events are depicted the day before units actually become latent, and the newly assigned latent, subclinical, and clinical units on each day are valid infection sources. Because of this fact, we build probability spinners for exposure sources after all other states have been assigned. We also need to visualize the predicted cross-state direct shipments, so the spinners are built including units from relevant external sectors. After computing exposure events, we apply the actual latent state using the previous day's exposure events.

Our second approach managed to fix many of the issues with our first approach, but still has some room for improvement. One issue is that the base weights used with the probability spinners do not reflect changes in input parameters. Because of this, the flexibility of the behavior depicted within the map visualization is limited. Furthermore, we have traded generality for accuracy in our implementation. The algorithm is tuned for reproducing the behavior of ADSM simulations, meaning that it cannot be directly applied to most other visualization problems. This may not be as big of an issue as we initially believed: it is likely that other subjects will share this restriction, demanding a customized main visual to most accurately reflect the theme and behavior of the predicted phenomena. This main visual can be swapped to reflect the problem area, while the underlying backbone of the tool remains fully generalizable.

3.3 Presenting Predictions: Plots and Graphs

We offer a variety of complimentary data views within Sonata. Each view type has its own strengths, and when examined together they provide planners with a more complete understanding of the scale and impact of the visualized outbreak. Currently, we offer line charts, stacked bar charts, and tables. Using our data management system alongside D3, Sonata can be extended with additional view types in the future. Each of the views can be positioned and configured to better meet the needs of the user base.

3.3.1 Line chart

Determining the type of charts to provide has led us to make tough decisions. Our user base is used to heat maps that are able to capture the likelihood of outcomes across multiple runs of a single stochastic ADSM scenario, an example of which is shown in Figure 3.5. Unfortunately, we cannot reproduce this likelihood information without increasing the number of models we load and run within Sonata. The models are the most expensive component of Sonata, both in terms of CPU and memory utilization, and adding more would be unwise considering our ambitions for other areas of the tool. Since the heat maps take the basic shape of bell curves and line charts, we decided to implement a line chart as our first accompanying view type.

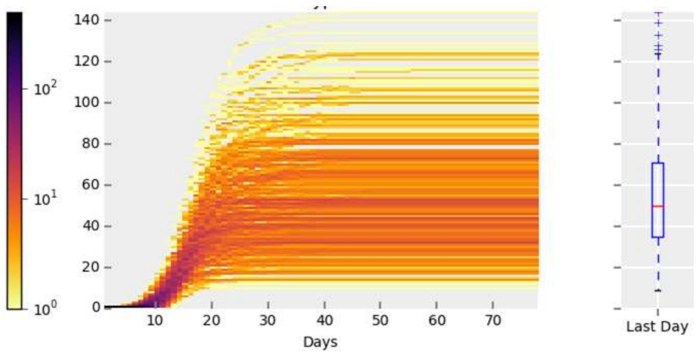


Figure 3.5: *A heat map plot produced by ADSM using data from 500 simulation runs.*

Our line chart view plots the daily changes in predicted outputs, with each output having its own line. The models predict discrete values, so we apply a curve to the line in order to prevent it from appearing jagged.

We were initially using a basis spline, but it interpolated the values in such a way that the resulting line often did not reach the predicted points. We later changed this to a cubic cardinal spline, which travels through the predicted points. The shaping of the line is handled for us by the D3 library. An example is shown in Figure 3.6.

The charts include a few features to help reinforce the notion of time as well as tie together changes that result from parameter changes. To start, the charts display a current visualized day marker and line-marker intersection points are highlighted. When time advances or rewinds, the bar and intersection points are animated to their new positions. The position of the lines is also animated, allowing the line to smoothly transition when input parameters change. This smooth transition is important, as it helps to reinforce the connection between old and new values, and therefore the impact of the latest parameter change.

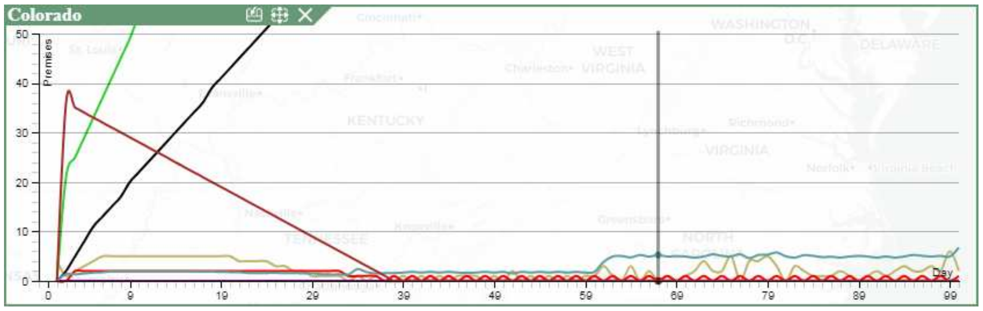


Figure 3.6: *An example line chart view. The current day is marked with a vertical rule.*

Updating the predicted values is reasonably fast, but not immediate, so we needed some way to indicate that some of the values are stale. Prematurely removing the previous values is counterproductive, as it produces a disconnect between the values in the user's mind. Instead, when a parameter changes and models are reevaluating, the stale chart values are displayed over a sliding gray background. This background will gradually return to white as new values are produced.

3.3.2 Stacked bar chart

The second supplemental view type that we introduced was stacked bar charts. The stacked bar charts allow planners to see the cumulative number of herds involved in an outbreak and how this sum changes over time. The stacked aspect of the chart allows planners to determine each individual output's (latent, clinical, etc.) relative contribution to the number of involved herds. This is in contrast to the line charts, which present the data in a way that makes it easy to measure each individual output and compare one output to another, but requires some work to determine the overall quantity of herds involved. Figure 3.7 shows outbreak predictions from Colorado using the stacked bar chart.

In the new chart, each bar is constructed from a series of distinctly colored rectangles, one color per output. To draw these rectangles, D3 provides a handy built-in function for stacking the data points. Unfortunately, we were unable to use it due to differences in the expected data arrangement; Sonata prepares data rowwise instead of columnwise. Seeing that we would have to transform the data anyway to be able to use the function, we implemented the functionality ourselves instead of processing the data twice. For each output we compute and emit a running sum, adding the current output to the sum of the outputs before it. This sum (and the screen position it corresponds to) is reset to zero after each day of data.

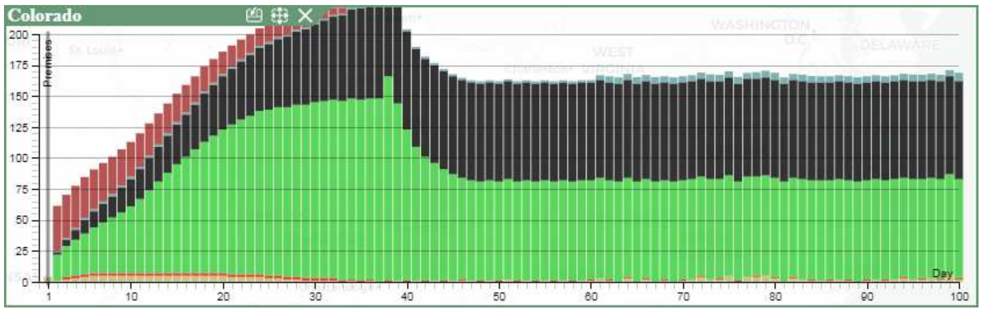
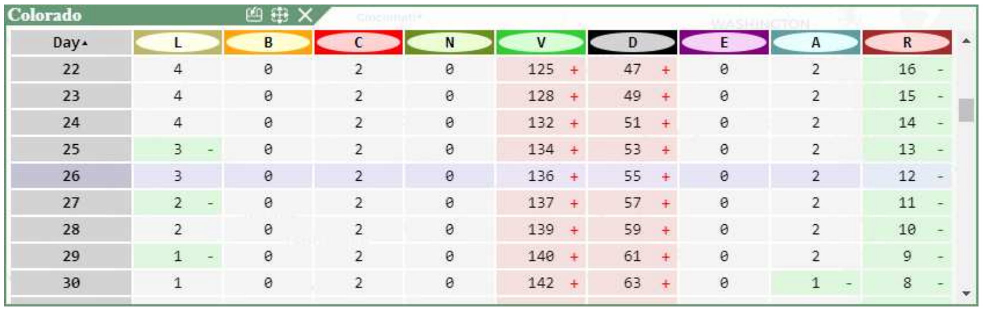


Figure 3.7: *An example stacked bar chart view, showing the proportion of herds in each health state.*

3.3.3 Tables

We introduced tables alongside the stacked bar charts. Reading and mentally processing data presented as text is less effective than more visual methods, but there are areas where tables shine. One of these areas is when planners want to know the exact quantity of herds for a given output. Due to chart scaling and reduction in the number of ticks in an effort to reduce clutter, extracting such information from one of the charts often requires estimation on the user's part. To obtain more exact values from a chart requires careful tracking of position along multiple axes or extra interaction (hovering over the chart in a specific location, for example). The table is also better at displaying data with several variables in a way that is at least readable, a common scenario that could arise simply by wanting to view a full range of outputs across a large number of locations (i.e. states). Trying to visualize the same amount of data using a more visual approach typically results in a complex and cluttered display, hindering comprehension.

As it is more difficult to grasp patterns in table-formatted data, we have added a few features to help planners get the most out of them. First, users are able to sort the data ascending or descending by column, useful for identifying areas of greater risk or that require more resources. Second, the table output is tied to the currently visualized day. This means that it will either highlight the row corresponding to the current day (auto-scrolling if necessary) or update the data displayed, depending on the data dimension configured on each axis. Finally, we have added highlighting for daily changes in individual cell values. Normally a cell's background appears light gray, but if the value increases from the previous day we add a hint of red and display a red plus sign. Correspondingly, if the value decreases then we add a hint of green to the background and display a green minus sign. Using red for increase and green for decrease is the opposite of what may be expected in common contexts (think financial ledgers), but in the context of an outbreak an increase in most outputs is an undesirable event. The exception to this is the immunity columns, including vaccination. We considered reversing the color scheme for these columns, but thought that remaining consistent across all outputs would improve overall readability. These features can be seen in Figure 3.8, below.



Colorado

Day▴	L	B	C	N	V	D	E	A	R
22	4	0	2	0	125 +	47 +	0	2	16 -
23	4	0	2	0	128 +	49 +	0	2	15 -
24	4	0	2	0	132 +	51 +	0	2	14 -
25	3 -	0	2	0	134 +	53 +	0	2	13 -
26	3	0	2	0	136 +	55 +	0	2	12 -
27	2 -	0	2	0	137 +	57 +	0	2	11 -
28	2	0	2	0	139 +	59 +	0	2	10 -
29	1 -	0	2	0	140 +	61 +	0	2	9 -
30	1	0	2	0	142 +	63 +	0	1 -	8 -

Figure 3.8: *An example table view. Daily increases are highlighted in red, while decreases are highlighted in green. The table automatically seeks as time advances.*

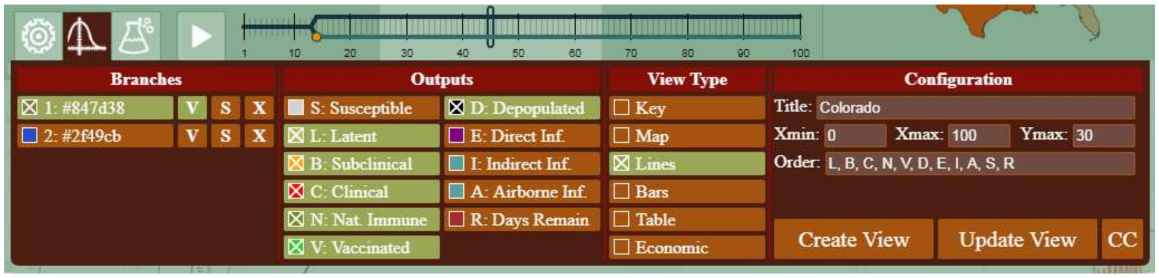
The tables currently offer two modes of operation: each row representing a state (data for the current day) or each row representing a day (data for a single state). Sonata automatically changes the mode depending on the number of states the table is configured to display. In both modes, the columns are the selected outputs. These modes provide the most practical information to planners without requiring much configuration, but limits the analytics options available to planners overall. Technically speaking, Sonata is able to display any two of the four data dimensions (branch, time, location, or output) along any axis, aggregating selected entries from the remaining dimensions. This ability applies to line, bar, and table views. Although a few of the possible combinations and resulting aggregations may be of use to a planner, the majority are likely not and may mislead planners if the views are not carefully configured. We will add the ability to configure axis dimension in the future, if this functionality is desired by the end users.

3.4 View Configuration and Management

In Sonata, users are given the freedom to select which views of the data they would like to see and where they would like the views to be positioned. Selecting how views display data is done through a configuration menu accessible from the Sonata interface header. The positioning of views is accomplished by dragging various interface elements, but drawing the changes is accomplished by Sonata's internal view management system. We explore both of these components throughout the following section.

3.4.1 View configuration menu

The view configuration menu provides a way for users to adjust the way content is displayed by Sonata. The menu exposes information about selected views, including the view type and any branches, outputs, and state scenarios that the view is subscribed to. In addition to this information, there is a menu section devoted to configuration specific to the selected view type that includes options such as the rate of ticks on an axis or whether a linear or logarithmic scale should be used. Using the menu, shown in Figure 3.9 below, planners are able to change the provided configuration and either save the changes to the selected view or open a new view. Users are able to change the view type (e.g. chart to table), and Sonata will attempt to preserve as many configuration options as possible. We have implemented a configuration copy mode where selecting a view will only adjust the selected states, allowing the changes to be saved to more than one view without needing to reselect all of the desired options multiple times. The process begins by opening the menu and selecting any view by clicking on it. As a convenient shortcut, double-clicking a view will open the menu and select the view all at once. Either method will show the selected view's configuration within the menu.



Branches				Outputs		View Type	Configuration
☒ 1: #847d38	V	S	X	☐ S: Susceptible	☒ D: Depopulated	☐ Key	Title: Colorado
☐ 2: #2f49cb	V	S	X	☒ L: Latent	☐ E: Direct Inf.	☐ Map	Xmin: 0 Xmax: 100 Ymax: 30
				☒ B: Subclinical	☐ I: Indirect Inf.	☒ Lines	Order: L, B, C, N, V, D, E, I, A, S, R
				☒ C: Clinical	☐ A: Airborne Inf.	☐ Bars	
				☒ N: Nat. Immune	☐ R: Days Remain	☐ Table	Create View / Update View / CC
				☒ V: Vaccinated		☐ Economic	

Figure 3.9: *The view configuration menu. Using the menu, users can configure the view type as well as which model outputs the view is subscribed to. Some options that adjust the layout of the data is also exposed.*

The menu communicates with views by passing a configuration object in the form of a key-value dictionary. We can request that the views populate the object with their current configuration, or use the contents of the object to set their configuration. After a view populates the object, the menu code will highlight the branches, outputs, and states that are reported as used by the views. In reverse, the views will subscribe to relevant data computations that provide the outputs desired by the user. In the case that a key is missing, the views will retain their current value. Keys may be missing when users create new views or change view types via the menu, but we also take advantage of this behavior when opening views programmatically, as we do when opening views during startup. This allows us to only list the options that we care about, reducing the need for future maintenance as new features are added.

3.4.2 View management

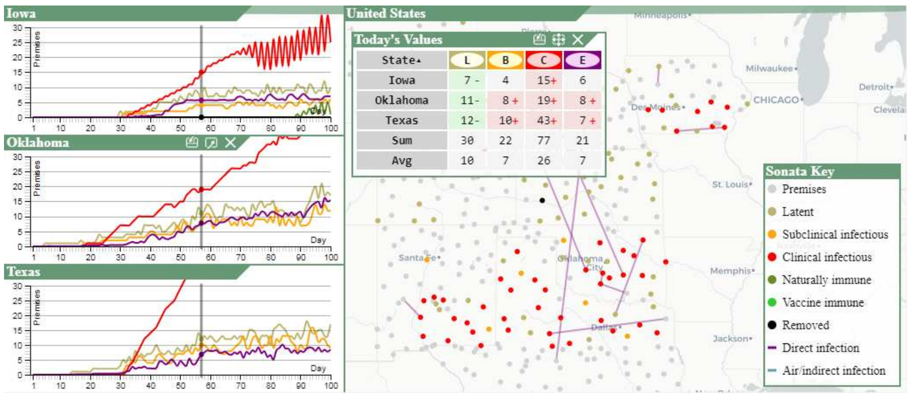
It is difficult to know at design time what kind of analysis a given user might be interested in performing. In this light, we wanted to give users the opportunity to freely position and rearrange the data views to fit their needs during any given planning exercise. As such, we needed a component for allocating screen space to views and managing their movement, Sonata's view management system.

For context, each view is assigned a single HTML element, and the location of this element determines where the view appears on the screen. To modify the displayed content, each view is allowed to freely change the child nodes of its assigned element. Adjusting the size and position of the element, however, is the responsibility of the view manager.

To share screen space, we initially divided the area into columns. The width of the columns was adjustable by the user, but the height extended from the top of the page to the bottom. With this design, each column had its own element and nested directly within were elements for the views. The views in a single column were stacked on top of each other, and each view extended the fill width of the column. Altogether, the views filled the full height of the column, but the proportion of the height that each view consumed could be adjusted by the user. Views could swap places by dragging one view into another view's location, and a new column could be created by dragging a view near the edge. At the time, new map views could be opened by dragging the timeline into the view area.

This approach did not fulfill our requirements and had a number of downsides. Of primary concern was that all views were locked in a grid; it was not possible to have charts overlaying the map, for example. Furthermore, the system only allowed stacking views one to one, so configurations that featured multiple charts over a single map could not be created. We attempted to remedy the latter by nesting additional columns within the area that would have been previously occupied by a single view. This nesting ended up complicating the solution, and made rebalancing screen space difficult.

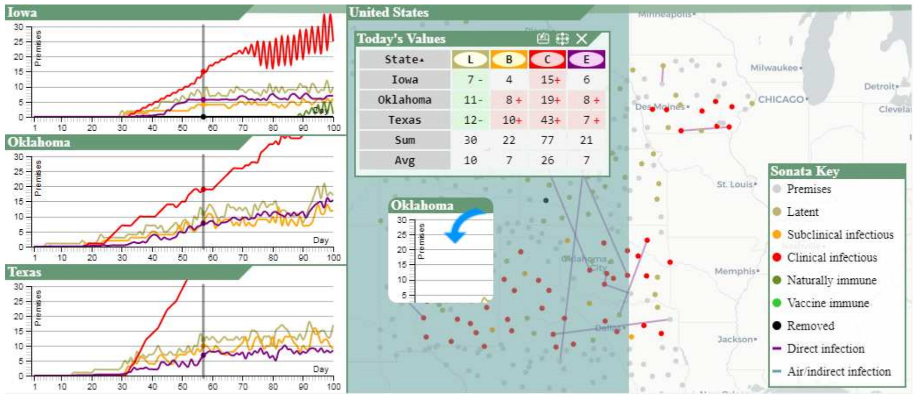
Considering the complications and lack of flexibility of our first approach, we decided to re-implement the view manager, shown in Figures 3.10 and 3.11. In order to more easily accomplish our design goals, we reworked the code from using nested column-based elements to using absolutely positioned elements, as required by floating views (i.e. views that appear to float above other views). Instead of using different approaches for grid and floating views, both rely on absolute positioning in which a top and left screen offset as well as a width and a height are defined. Implementing floating views this way is fairly straightforward, as moving and resizing the view simply adjusts these four values.



State▲	L	B	C	E
Iowa	7 -	4	15+	6
Oklahoma	11-	8 +	19+	8 +
Texas	12-	10+	43+	7 +
Sum	30	22	77	21
Avg	10	7	26	7

Figure 3.10: *The view management system allows views to be placed as the user desires.*

Floating views are repositioned by dragging the title bar and resized by dragging the edges. Grid views have the requirement that the entire space is occupied by views with no gaps, so operations on one view often require adjusting the positioning of neighboring views. Grid views are subject to five operations: swap, replace, insert, resize, and remove. Swapping and replacing maintains the size and position of the areas the views were previously assigned to, but changes the assigned views. Like with the column approach, this is accomplished by dragging one view into the center area of another view. The insert operation (shown in Figure 3.11 below) splits the area of one view in half and assigns the other half to a different view. Views can be inserted on the top, left, right, or bottom of another view; this is determined by dragging the view to be inserted over the specific side of the view to be split. Generated edges appear between views in the grid. Resizing a view is accomplished by dragging one of these edges. In fact, the user can select multiple edges in a single touch motion to resize additional nearby views in the same action. As the edge is dragged, views on one side will grow and views on the other side will shrink. When a view faces more than one other view across the same side, a single edge is shared between them all. Sharing the edges ensures that all touching views are resized properly and that no gaps appear. While resizing, edges will snap to various convenient locations. These include even fractions of the screen width and the location of other edges. The last operation, removing a view, results in the area occupied by the view being given to one of its neighbors.



State▲	L	B	C	E
Iowa	7 -	4	15+	6
Oklahoma	11-	8 +	19+	8 +
Texas	12-	10+	43+	7 +
Sum	30	22	77	21
Avg	10	7	26	7

Figure 3.11: *A user drags the Oklahoma line chart to place it on the left side of the map.*

Altogether, these changes greatly improved the flexibility users were afforded when laying out various outbreak results. Unfortunately, this approach opened the door to a fair number of unexpected edge-related edge cases. During testing, the grid views would regularly enter a broken state in the form of overlapping views or un-closable gaps, all from what seemed to us to be normal interaction. After investigating the cause, the fix for one of the cases required complex bounds checks covering multiple views and edges, ensuring that views in the particular configuration were unable to move relative to each other. Instead of working out solutions for every case, we inverted the logic in order to eliminate the edge cases altogether. In our current approach, views no longer specify the offsets, width, or height and instead reference the edge that they are touching on their top, left, right, and bottom side. The edges are defined as either vertical or horizontal and have a corresponding single position variable; the actual view size and position is computed from the position of its edges. All operations are now in terms of the edges: adding, moving, merging, splitting, and sliding them, with each view's edge references changed when appropriate. This simplified the solution in most cases and made it impossible for gaps to form between views.

The view manager has other responsibilities as well. The view manager tracks a view's data subscriptions, and ensures that resources are properly cleaned up when the view is closed. It additionally exposes callbacks that inform views of various Sonata events. These events include changes in the currently visualized day, updates to model data, changes to the view's width or height, and changes in configuration.

4 Timeline and Parameters

While Cadence was intended as a tool capable of predicting the final outcome of an outbreak, our aim for Sonata is to predict and visualize the course of an outbreak as time unfolds. This goal brings Sonata's timeline feature into focus, a component that allows planners to control how events unfold over time within the visualization. Many visual components change with the flow of time, including the outbreak map and various graphs and plots. As we consider the timeline to be a critical part of our tool, we have given it a prominent place in our interface. It is permanently located at the bottom of the header, available right above the outbreak map.

4.1 Timeline and Branching

Time is of great importance in Sonata, as the course of an outbreak can change at any given moment and understanding the reason for these changes is critical for developing a suitable counter strategy. Any tool that faithfully portrays spatiotemporal data needs a method of representing and controlling time. Within Sonata, the timeline component fulfills this crucial role. The timeline not only gives planners the ability to see when visualized events happen with respect to the duration of the outbreak, but also allows them to influence those events. In this section, we explore various aspects of the timeline, including the design of the interface, branching functionality for comparison, and tying the timeline in with the various views of the outbreak.

4.1.1 Timeline Interface

The timeline interface is composed of several parts. Of these, there is the actual line representing time, a current time marker, and controls for manipulating time. We believe that the timeline is crucial for understanding the context of predictions presented in Sonata's data views. For this reason, we have given it a prominent place in the overall interface, the top left of the screen, consuming a majority of the space allocated for the header. Furthermore, the timeline is always visible unlike the various configuration menus that collapse into the header.

As seen in Figure 4.1, the line itself is structured as a chain of diamond icons, each representing a day during an outbreak. Each day icon is connected to the previous and following day icon for continuity. Any day icon may be clicked to select it, and multiple day icons can be selected at once. When a day is selected, its icon is changed from a small diamond to a larger diamond with the given day number in its center. When users set parameters or configure data views in Sonata, the currently selected days are used as targets for the operation. Changes made to any selected day will take effect immediately, and the temporally linked outcomes on subsequent days will update as well. To indicate that a change will have a cascading effect, the color of all subsequent day icons from a selected day is changed from black to teal. Furthermore, adjusting any parameter on a selected day will result in the color being changed to orange with precedence over teal. This color change allows modified days to stand out during later interactions with the tool. Part of the challenge of designing the timeline was ensuring that it is recognizable as a timeline to new users. In an attempt to do so, we added stylistic tick marks to the top and bottom of the day icons. Furthermore, the tick marks are longer every tenth day which improves the process of locating a specific day on the timeline.

It is not practical for some of the data views to visualize all of time all at once. Imagine the confusion if events across all time were overlaid on a single map view, for example. For this reason, Sonata maintains a variable for the currently visualized day. A vertical slider bar sits across the timeline at that day’s position in order to inform and remind the user of the current day. The slider is adjustable and dragging it over the various day icons will update the currently visualized day accordingly and instantaneously. Sonata is also capable of advancing the currently visualized day automatically. This is enabled via a play/pause control button that is located near the bottom left of the timeline, just after the menu control buttons. When playback is active, the slider becomes animated and moves across the timeline at a constant rate. The rate of advancements defaults to one visualized day every two real seconds, but this can be adjusted in the options. Data views can subscribe to changes in the currently visualized day. Whenever the value changes, either manually or automatically, the timeline informs the visual components of the change in day so that they may make any required visual adjustments.

A large challenge that we faced was incorporating support for the visualization of longer outbreaks. Our original design involved drawing the complete timeline, always drawing all days at once. When doing this, adding more days to the timeline results in the days being drawn closer together so that they all fit on screen. If the days become too close, then users are not able to reliably interact with them for selection and identification purposes. This ultimately limited the visualization to 50 days, which is not long enough for some scenarios. To correct this, we have made it so that days on the timeline are a constant distance apart and only part of the timeline is visible at a given time, centered on the currently visualized day. We have added an overview section which shows where the visible main section is on the overall timeline, as well as highlighting days that are selected or modified as the main timeline does. The overview section features a current day slider and clicking any day on the overview will instantly set the currently visualized day, jumping to that day in the main section.

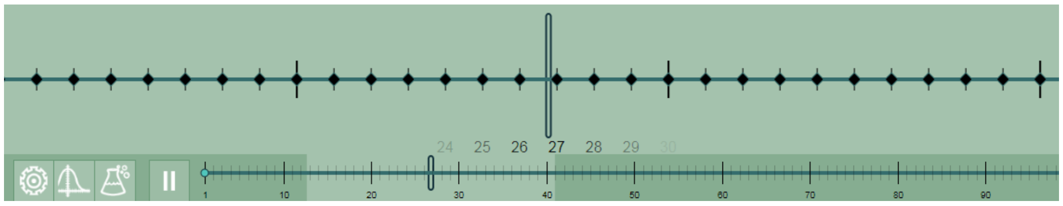


Figure 4.1: *The timeline interface. Time is automatically advancing via the play/pause button. As time advances, the timeline scrolls and views animate changes.*

4.1.2 Timeline Branching

In Sonata, a single path of day icons on the timeline represents a single sequence of temporally connected outcomes. While it is interesting to observe a predicted outbreak unfold, the amount of insight that can be gleaned from a single potential outcome is limited. To increase the impact of our tool, we have implemented timeline branching, seen in Figure 4.2.

Branching treats the timeline as a tree, allowing a single day to be logically followed by multiple days on different branches of the timeline. A branch point can be identified by multiple day icons connected and emerging from a single day in the direction of time. Using the interface, a branch point can be created by selecting any day and pressing the split button that appears directly underneath. When a branch is no longer needed, it can be removed by selecting the last day on that branch and pressing the remove button.

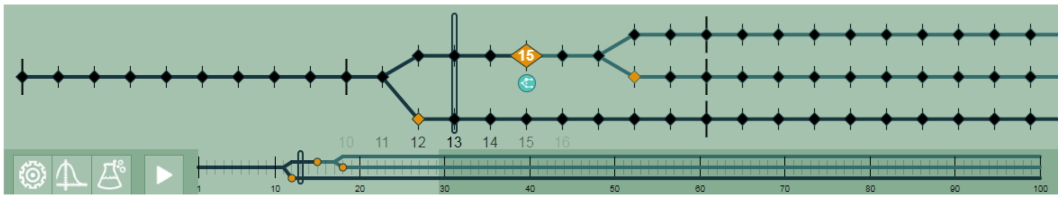


Figure 4.2: *The timeline interface with three branches. Days highlighted in orange have parameter changes. Larger icons indicate selected days, and days affected by a particular parameter change appear in cyan.*

Timeline branching creates a fork in the flow of events, where one or more decisions or events can happen differently and as a consequence will lead to distinct outcomes. Importantly, Sonata evaluates both outcomes simultaneously and presents both results. This functionality makes it easier for planners to answer "what-if" style questions relating to the impact one or more variables will have on future events, as they do not have to note the prior results and reset the tool to ask the next question.

Parameter inheritance over time plays a key role in the timeline's functionality. On the timeline, any parameters set on a day will be inherited by all subsequent days, continuing until a day where the value is reassigned. This is true after branch points as well, in which case all diverging branches inherit the value. Due to this inheritance scheme, a newly created branch will depict exactly the same outcome as its sibling, making it an excellent starting point for exploration. Changes made after the branch point will only influence the specific branch, while changes made before the branch point will influence both branches. Planners can use this behavior to their advantage, giving them the power to investigate the role and impact of individual parameters, as well as allowing them to investigate how groups of parameters influence each other over time. Furthermore, if planners are interested in optimizing predicted outcomes to meet a given goal, they can do so by keeping branches that depict the best results according to their criteria. Branches can be created that diverge from these best results, allowing them to be iteratively refined. The user can continue the exploration process with as many branches as they desire, primarily limited by their available screen space.

4.1.3 Managing Branching Visuals

Sonata automatically opens views when a new branch is created. When this occurs, screen space in the view area is divided evenly by the number of branches. In each branch's partition, a map view is opened in grid mode to act as the background. In addition to this, Sonata will copy the configuration, size, and relative position of the floating views associated with the topmost branch and then opens corresponding views for each other branch. This operation makes heavy use of the view configuration objects described in Section 3.4.1.

The large number of branches and views that can be open simultaneously poses interesting usability and performance challenges. One problem that we encountered was that there was no visual mechanism for associating views with a given branch. Without such a mechanism, it is rather easy to lose track of which branch's changes were being reflected, as views from all branches look otherwise identical. To complicate the issue, it was important that any changes we made to the views did not distract users from the visuals presented. As our solution, when users click or tap on a branch in the timeline, it will highlight all views that are associated with that branch. Each branch has its own generated color, and both the branch and the view borders will display this color when highlighting is active. We experimented with flashing effects, hovering, and transparency to determine an effective identification mechanism with the least distraction, but it turned out that a steady highlight using a bold border was sufficient for grabbing user attention.

We have also needed to make space tradeoffs when determining how much screen real-estate should be allocated to the timeline. On one hand, if we increase the screen space allocated to the timeline, then fewer map and chart views can be displayed; on the other hand, if we reduce the screen space, then we reduce the number of branches that will fit. These two sides are closely linked, as using more branches inherently requires more views. To have the best of both worlds, we tried collapsing the timeline into a single thin line as the visualization was playing. Unfortunately, this ended up hiding valuable information about where branch points were, leading to difficulties anticipating when the displayed visuals where going to diverge. We ultimately came to a balance when we incorporated state status and selection into Sonata. This component, which sits to the right of the timeline in the header, depicts status of states in the continental United States and must be at least tall enough for users to be able to interact with it. This set a minimum reasonable height for the timeline, which we decided to keep.

4.2 Scenario Parameters

As has been hinted at by the timeline, Sonata provides users with a way to influence the predicted outbreak. To accomplish this, we allow users to adjust various scenario parameters, parameters that ultimately feed into the models that back Sonata's visualizations. The parameters are defined by the source simulations, NAADSM and ADSM in our case. In this section, we provide some background on these parameters and our process for including them in Sonata.

4.2.1 Preparing Parameters

To ensure that Sonata remains as flexible as possible and can be applied to a wide range of problem sets, none of the parameters nor associated models are hardcoded. Instead, the entire collection of parameters and models are loaded at runtime. In order to make the information available to Sonata, we encode the parameters and models across multiple JSON files. These JSON files are generated by scripts written in the Python programming language. The scripts read scenario files produced by subject matter experts to determine the parameters used within those scenarios.

Before we can include any parameters in our JSON files, we must extract them from the original simulations. We extract parameter values from scenario files belonging to both NAADSM and its successor ADSM. The two simulations use different file formats, so we need two separate implementations to read them. The NAADSM scenario files are in XML, with parameter values nested within various model tags. We utilize XPath for querying the scenario file, as the values for a given parameter have a well-defined location. The ADSM scenarios, on the other hand, are stored in SQLite 3 database files. To extract values from these databases, we construct and issue SQL queries. The parameters are spread across several tables, and require converting production types into id numbers.

Multiple production types are defined within the scenario, with many of the parameters duplicated for each production type. In the scenarios, production types primarily include variations of cow calf farms, dairy farms, feedlots, swine, and small ruminates. This poses some complications, as the number of types, type names, and type ids are determined by the scenario designers. This ultimately leads to different scenarios having different types and names, as is the case with the Texas Region scenario, Iowa scenario, and Great Plains scenario. To cope with this, we map the production types to a standard set of types presented to users while using Sonata. We use a standard set of production types within Sonata to improve the interoperability between scenarios reflecting different regions, allowing users to configure parameters for multiple regions simultaneously without needing to worry about type differences.

When we extract parameters, we separate the information into multiple JSON files. The first of these files contains data to be used by the parameters menu within Sonata. This file specifies the name, location, description, and path to an icon image for every parameter available to users through the menu. Charts are displayed with some parameters, so this file includes data points for the chart as well as unit text for the x and y axes. This file is loaded once by Sonata when the tool first loads.

The other JSON files that are produced contain default values for each of the parameters. The default values are extracted from a base scenario, a scenario which is also used during model building as a starting point for the creation of scenario variants that include altered parameters. The base scenario is created by subject matter experts and should contain reasonable expected values that are suitable for use as defaults. Alongside the default values, we also include the minimum and maximum allowed value range in the JSON files. This happens to be the same range used when making scenario variants, ensuring that users can only enter values that satisfy the established bounds.

We produce a set of default values for each state in the scenario. For background, we divide the scenario into its component states so that each state can be loaded and unloaded independently from the others. In fact, states originating from multiple scenarios can be mixed and matched. Historically, we included the default parameters values in the menu JSON file. This had some undesirable consequences when using data from multiple scenarios as we had multiple sets of default values, and needed to keep track of which states could use each set of defaults. Additionally, this required us to wait for both the menu and state JSON files to load, as processing could not proceed without data from both files. Bundling the default values into the state JSON files simplified the process, as a separate state parameter mapping was no longer required and processing could begin as soon as the state's JSON file loaded. Since values are shared by all states originating from the same scenario, we do duplicate the values across the files, but they only contribute a couple tenths of a percent of the overall file size once compressed. The state JSON files are described in more detail alongside the other national-scale features in Section 5.3.2.

4.2.2 Parameter Types

There are three distinct types of parameters used by the source ADSM simulation: numeric values, probability density functions (PDF), and x-y relation charts. To best capture the flexibility afforded by the simulation, we support all three types within Sonata and each parameter type is presented to the user with unique slider controls. For simple numeric values, we present a slider and display the current value in larger text above. An example that uses a numeric parameter is the average daily rate of outgoing shipments from a farm. For PDFs, we offer multiple sliders styled as a box-plot with a beta distribution drawn overhead. An example of a PDF parameter would be the number of days a farm spends in the incubating disease stage. For x-y charts, we provide sliders along the sides of a chart visual that adjust the minimum and maximum of the x and y range. An example within Sonata would be the ramp-up of available vaccination resources over time. Examples of the slider interface for controlling the three types are shown in Figure 4.3. The positions of the sliders correspond to numeric values that are used by the various models. For more exact entry, we also offer a text entry mode for each parameter type.

It is important to draw a distinction between scenario parameters and model features. Each one scenario parameter can be composed of several model features, with each model feature being a single dimension within the input space. Numeric parameters are the simplest, and only contribute one dimension to the models. On the other hand, PDFs involve defining the min, max, mean, and other shaping information.

Although listed as single parameters within Sonata, they contribute five dimensions to the models. Similarly, x-y charts contribute 4 dimensions due to their min and max components. When we discuss parameters within this dissertation, we are referring to the scenario parameters unless otherwise stated.

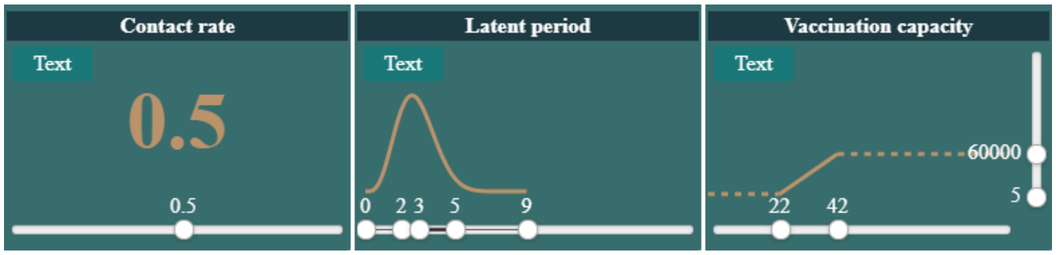


Figure 4.3: *Examples of the interface for three input types. On the left is a simple numeric value, in the center is a PDF, and on the right is an x-y relation chart.*

4.2.3 Parameter Limitations

The range of acceptable parameter values is limited compared to what is achievable in the original simulation. The reasoning for this is twofold: reducing model complexity and general ease of use. First off, if we were to give users the ability to fully configure the x-y relation charts and PDFs, then an unreasonably large number of model parameters would be required to capture them faithfully. This would lead to dimensionality problems during the training process and consequentially less accurate models. As a compromise, we only allow users to modify the x and y min and max for relation charts, resulting in a stretched version of the original chart produced by the subject experts that produced the base scenario. The PDF parameters face a similar restriction, as we limit users to adjusting the shape of beta distributions.

Expert users may have to adjust their routine due to the inability to customize the base relation, but the restriction is necessary due to our use of machine learning models combined with the number of scenario parameters involved. On the bright side, limiting parameter ranges in this way allows faster modification of the parameter values. As they are, the ranges are wide enough that they should be able to reflect most plausible outbreak scenarios from the mundane to the extreme. As the values are limited to plausible quantities, users can quickly adjust the parameter without needing to worry about potentially entering an unrealistic value.

4.2.4 Parameter value storage

Where we store parameter values has changed several times over the course of developing Sonata. Initially, the parameter values were stored within day objects on the timeline. Parameters can be set per day, so storing them as part of the timeline makes sense. When the tool initially loaded, the default parameter values for the first day were set using values prepared in a JSON file. Immediately following that, all subsequent days are updated so that they refer to this initial value, as parameter values are inherited by subsequent days until they are reassigned. Using references saves on memory costs as we do not need a duplicate value object for every inherited day. In addition to this, we only need to compute which value is referenced once when the parameter is reassigned, saving time when rapidly accessing values during model evaluation. If a day with a reassigned value is changed again (the most common case), then subsequent days automatically inherit the new value via the reference. We do not even need to recompute inheritance in this case, as inheritance only changes if a parameter value that is currently inherited on a given day is assigned its own value on that day.

We changed this approach when we introduced our Computation Dependency Graph (see Section 6.1.2). For context, the computations execute a function and cache the value. Computations register which input computations they depend on, and are only reevaluated when one of these inputs change. There was now a benefit to storing the parameter values within the computations: specific parameters could be declared as input dependencies for model computations, and thanks to the Dependency Graph only the subset of models that rely on a particular parameter will be reevaluated when the given parameter's value changes. Additionally, the day to day parameter inheritance scheme could be handled automatically using computation dependencies, provided that there were separate parameter computation instances for each day. The issue with this approach was that all parameters were being evaluated as computations. Individual parameter computations evaluate very quickly as they simply check the previous day's value for inheritance purposes, but due to the number of parameters over all timeline days, these small contributions added up. In fact, a majority of the computations were parameter computations, and these were all scheduled for evaluation upon the tool's initial start, contributing to loading delays.

In response to this, we developed a middle of the road solution that took aspects from both prior approaches. As part of this, we have returned to storing the parameter values within timeline days, instead of exclusively as computations. The values stored within timeline days and computations now serve distinct purposes, and as a result are often set at different times. The known values for all parameters, whether configured manually using the menu or automatically by other aspects of the tool, are maintained in the timeline. Instead of using references in the timeline as our first approach did, days hold a value for a parameter only if the parameter value was changed on that day. We attempted to save the current state of each parameter on every day of the timeline, but it turned out doing so resulted in an overwhelming number of parameter entries. With just one state and 100 days on the timeline, we would need 295,200 parameter entries. This number increases with each additional state and branch. In the end, this slowed Sonata's loading time and branch creation noticeably, so we changed our approach to only store value changes within the timeline days. A missing value implies that the value should be inherited recursively from the previous day, and the first day of each branch contains initial values for all parameters. Resolving inheritance and model dependencies is handled by the Computation Dependency Graph. Parameter computations are now only created when requested by another computation (such as a model), slightly speeding up result times. Each parameter computation registers the associated parameter computation from the previous day as a dependency, so should a value change, the computations on subsequent days will reevaluate in order. When one of the parameter computations is evaluated, it checks the corresponding timeline day to see if a value is set. If so, the computation result is a reference to the value stored in the timeline; if not, the computation result becomes the value reference held by its dependency, inheriting the value. To ensure that computations update when the master value stored in the timeline changes, the code for setting parameter values checks for the existence of a matching parameter computation, and if one exists its value is changed as well. This value change will cause any dependent computations to update, like before.

4.3 Parameters Menu

Allowing users to change parameters on demand is essential to Sonata's design. We provided this level of access to scenario parameters via a ribbon menu, available as part of Sonata's main control interface. Importantly, this menu can be opened or closed at any time and any changes made via the menu take effect immediately, allowing users to actively see the impact of their decisions.

4.3.1 Menu Layout

There are several hundred adjustable scenario parameters exposed by Sonata. This introduces a bit of a challenge, as we have to provide access to all of these parameters within a limited space, all while not overwhelming the user with information. While we could open a full-screen popup window and list all of the available parameters, doing so would obscure the data views and require users to close the popup after every change before they could examine the impact. In such a scenario we would also need to delay any view update animations until the parameters popup had closed, otherwise the animations would be not be visible to the user and their usefulness lost. Our desire for users to see the impact changes right as they made them required a new solution.

Since screen space is limited, we instead structure the parameters as a ribbon menu, shown in Figure 4.4. The ribbon menu stretches horizontally across the top of the screen just below the timeline. The menu can be opened from a bubbling vial icon (representing experimentation) near the bottom left of the timeline. The parameters menu is thin, consuming the same vertical height as the view configuration menu and permits the majority views on the screen to be seen while it is open. The menu is broken up into three sections. In order from left to right, these are the parameter list section, the production types sections, and the value section.

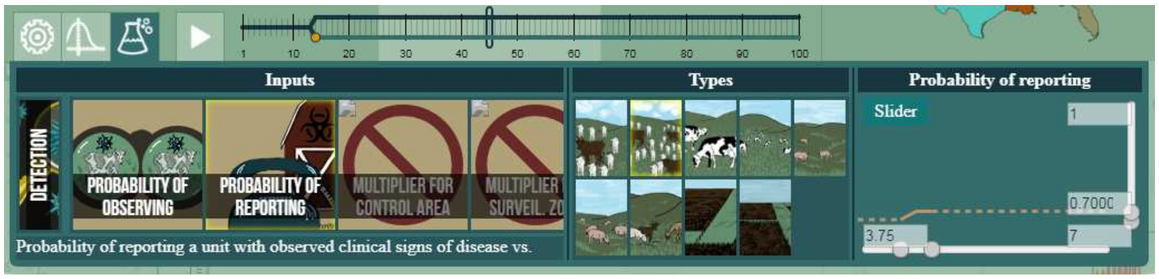


Figure 4.4: *The parameters ribbon menu, located at the top of the control interface. The icons, when clicked, list parameters relevant to the selected production type.*

4.3.2 Parameter List Section

The purpose of the parameters section is to allow users to locate the parameters that they are interested in quickly. In an attempt to be touch-screen friendly, the parameters are presented as a scrollable list of large square icons. The background of each square features an image depicting the parameter, and across the lower center of the square is the name of the parameter distinguished by a partially-transparent black background. As there are hundreds of parameters, it would be counterproductive to list them all one after the other; doing so would require users to constantly search through the list. Instead, the parameters are structured within nested sub-menus for ease of access, with only the immediate contents of one sub-menu being displayed across the screen at one time. The parameters are categorized based on what behaviors they target within the NAADSM and ADSM simulations, with some top level categories including "Testing", "Vaccination", and "Detection", among others. Ensuring that the categorization is intuitive is essential for allowing users to navigate the menus effectively.

To enable quick navigation of parameters, we designed the list section with the goal of allowing parameters to be accessed with only a few clicks. Using our nested menus and categorization, most parameters can be selected with only two or three clicks. Clicks occur in three ways while navigating the parameter list. First, clicking a category icon will replace the listed icons with that sub-menu's nested children, moving into a deeper level of the menu hierarchy. Next, clicking a parameter icon will display that parameter's value within the value section while leaving the listed parameters unchanged. The clicked parameter is given a yellow border highlight to emphasize that its value is the one currently selected for display. In addition to showing the value, a text description of the parameter is displayed below the parameter list. Since the listed parameters are unchanged when adjusting values, related parameters can be selected immediately without needing to re-navigate the menu. The last way clicks are required is when returning to higher levels within the menu hierarchy. Return navigation is possible thanks to current location icons that appear to the left of the parameter and category icons, one per level in the hierarchy. Clicking a current location icon will undo the action that put it there, placing the user in the parent menu. As an example, clicking the "Detection" category icon will place a "Detection" icon on the left side of the list as an indication that users are now in the "Detection" sub-menu. Clicking the "Detection" current location icon will place the user back in the menu that contained the "Detection" category icon and remove the location icon.

We added additional features in an attempt to reduce the number of clicks while navigating the menu. One of these was bookmarking, a feature that pinned a tab link to any parameter or category icon on the top of the parameter list, allowing users to quickly revisit parameters of interest. A bookmark, depicted at the top of Figure 4.5, is created by dragging either a menu or parameter icon upwards and releasing it. Existing bookmark tabs can be reordered by dragging them sideways; when doing so, other tabs shift to leave a gap where the dragged bookmark will end up when released. Unfortunately, this particular feature ended up not being used, as users constantly forgot to add the bookmarks before navigating away. One reason for this might be that users only realize that they want to revisit certain parameters after further interactions. As a result of the feature's poor performance, we removed it in favor of other solutions. One particularly promising alternative is the incorporation of navigation shortcuts into parameter change summaries. Change summaries list every parameter that was altered on a particular day, and can be accessed by double clicking the orange-highlighted days on the timeline. Not only can users see exactly how the values changed, but they can also adjust them further without needing to re-navigate the menu. Every listed parameter will take the user directly to the correct location within the parameters menu.

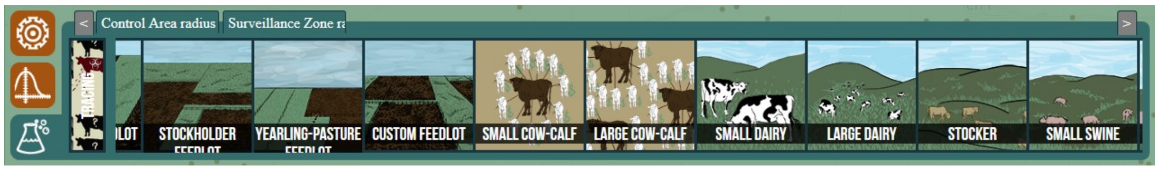


Figure 4.5: *An early version of the parameters menu. The production types are located in the navigation hierarchy, and bookmark tabs can be seen on the top.*

4.3.3 Production Types Section

We introduced the production types section in our effort to reduce the number of clicks needed to find and adjust parameters. This section of the menu displays all of the production types known to Sonata. This is useful as many parameters exist per production type, with unique values for each. Users are able to select any number of production types from this section and parameter value changes will apply to all the selected types simultaneously. Furthermore, users are able to hover over a single type to preview its value.

Prior to the introduction of the production types section, selection of the production type was done through sub-menu hierarchy navigation. Within each of the top level categories was another category icon for each of the production types. Clicking one of these icons would present the user with the list of parameters tied to the production type. The parameters that were not associated with a production type were listed alongside the production type categories, one level higher.

This approach had additional downsides besides the additional clicks. Due to the need to navigate one level deeper to view type-specific parameters, users cannot quickly check the existence of certain parameters within each top-level category. As such, it takes more time to find parameters and is considerably easier to get lost in the sub-menu tree while doing so. In addition to this issue, users must set the value for each individual production type separately, re-navigating the menu to reach the same parameter for a different type. There are several parameters where it makes sense to change the parameter value for multiple production types at the same time, such as for response times or test success rates. Having the production types as a section separate for the list menu remedies these issues.

4.3.4 Parameter Value Section

The third section of the parameters menu is the parameter value section. Located on the right side of the menu, its purpose is to allow users to view and adjust a selected parameter's value. How the value is displayed depends on the parameter type. We support simple numbers, probability density functions, and relational charts. For each of these types, we make use of sliders to visually adjust and shape the parameter value as described in Section 4.3.4. In addition to adjustment via sliders, we also offer a text entry mode for precise value entry. The axis units (e.g. days, kilometers, frequency) for sliders and text entry are labeled so that users know exactly what they are changing.

When viewing a parameter, the value shown depends on the selected parameter, selected production types, selected timeline days, and selected states. Changing any of these selections will immediately cause the displayed value to update accordingly. Multiple production types, days, and states can be selected at once, but each combination can have its own distinct value. In the case that multiple parameter instances are matched and their values differ, we show the value from the last selected combination (as opposed to averaging all matches, which can change the value in undesirable ways). We detect when differences exist and display a warning icon to users, advising them to verify their selection. Changes made to a parameter via the value section are applied to each combination of type, day, and state. Making a change when differences exist will overwrite every value, making all parameter instances the same.

As with most aspects of the current iteration of the parameters menu, the value section arose out of an attempt to reduce the number of clicks needed to navigate the menu. Before its introduction, selecting a parameter from the menu would open a popup window for slider adjustment. Planners interested in viewing or changing a value needed to click the top level menu icon, select the production type, select the parameter name (opening a value popup window), set the value, and then close the popup window. By removing the popup window, we eliminate at least one click and have the potential to remove another. Users still need to click on the parameter icon and select it in order to change the value, but if the planner only wishes to preview an existing value, then desktop users have the option to hover over the input name to show the current value.

This allows them to quickly move over the list of parameters and review the value for each without clicking. Production type, day, and state selections remain the same when moving between parameters.

4.3.5 Missing and Inactive Parameters

Not all parameters and productions types are defined in every scenario, and even where they are defined, some do not significantly impact model outcomes and are excluded. If a parameter is not used by any models, allowing its value to be adjusted by users not only wastes the user's time, but also can introduce concerns that the tool is not working properly, discouraging further exploration. Removing unused parameters from the list is not a suitable option either, as some users might be explicitly looking for one and expecting it to be in a specific location. If they do not find the parameter in the expected location, they are likely to waste their time searching the other category sub-menus for it, even rechecking some sub-menus several times to make sure they did not accidentally skip over it. To combat these behaviors, we ensure that unused parameters are properly marked.

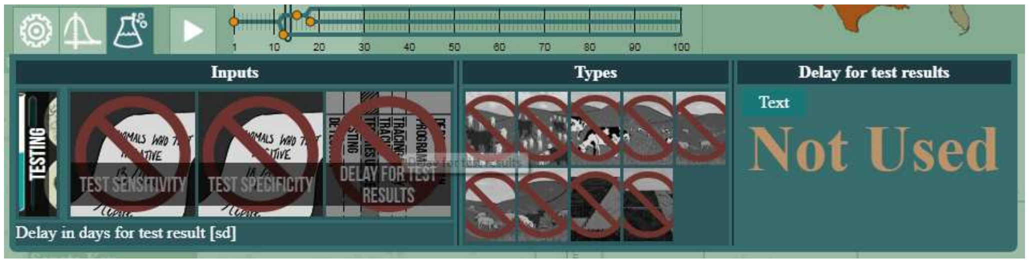
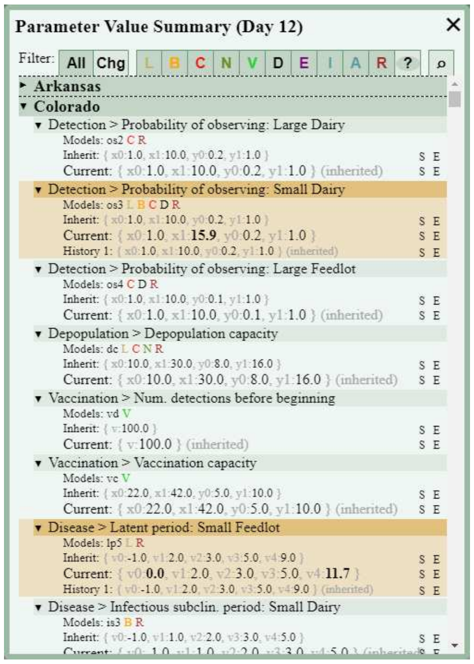


Figure 4.6: *Many missing parameters in the testing category. Values for these parameters are not defined in the Great Plains simulation scenario.*

There are several visual differences within the menu for missing and unused parameters. To start, the parameter icon is grayed-out and the name of the parameter that appears over the icon is also darkened. For users that experience difficulty distinguishing grays from the other light colors that appear in the icons, a dark red "no sign" is also displayed over the icon, layered under the parameter name. When a user hovers over or selects one of the missing parameters, the phrase "Not Used" will appear in the value section as seen in Figure 4.6. Parameter values are often not defined for specific production types. In such a case, the production type icons corresponding to the missing value are changed in the same way as parameters. A parameter's icon is grayed out only if its value is undefined or unused for all production types.

4.4 Timeline Parameter Summary

Sonata is able to display a summary of all parameter values set on a given day. Shown in Figure 4.7, the summary is accessible from the timeline by double clicking (or double tapping) any day on the timeline. When this is done, the day is exclusively selected (automatically unselecting any other days) and a summary overlay is opened that lists parameter information for that day.



```
Parameter Value Summary (Day 12)                                                    ✕
Filter: All Chg  L B C N V D E I A R ?
▸ Arkansas
▾ Colorado
  ▾ Detection > Probability of observing: Large Dairy
      Models: os2 C R
      Inherit: { x0:1.0, x1:10.0, y0:0.2, y1:1.0 }                          S E
      Current: { x0:1.0, x1:10.0, y0:0.2, y1:1.0 } (inherited)              S E
  ▾ Detection > Probability of observing: Small Dairy
      Models: os3 L B C D R
      Inherit: { x0:1.0, x1:10.0, y0:0.2, y1:1.0 }                          S E
      Current: { x0:1.0, x1:15.9, y0:0.2, y1:1.0 }                          S E
      History 1: { x0:1.0, x1:10.0, y0:0.2, y1:1.0 } (inherited)            S E
  ▾ Detection > Probability of observing: Large Feedlot
      Models: os4 C D R
      Inherit: { x0:1.0, x1:10.0, y0:0.1, y1:1.0 }                          S E
      Current: { x0:1.0, x1:10.0, y0:0.1, y1:1.0 } (inherited)              S E
  ▾ Depopulation > Depopulation capacity
      Models: dc L C N R
      Inherit: { x0:10.0, x1:30.0, y0:8.0, y1:16.0 }                        S E
      Current: { x0:10.0, x1:30.0, y0:8.0, y1:16.0 } (inherited)            S E
  ▾ Vaccination > Num. detections before beginning
      Models: vd V
      Inherit: { v:100.0 }                                                  S E
      Current: { v:100.0 } (inherited)                                      S E
  ▾ Vaccination > Vaccination capacity
      Models: vc V
      Inherit: { x0:22.0, x1:42.0, y0:5.0, y1:10.0 }                        S E
      Current: { x0:22.0, x1:42.0, y0:5.0, y1:10.0 } (inherited)            S E
  ▾ Disease > Latent period: Small Feedlot
      Models: lp5 L R
      Inherit: { v0:-1.0, v1:2.0, v2:3.0, v3:5.0, v4:9.0 }                  S E
      Current: { v0:0.0, v1:2.0, v2:3.0, v3:5.0, v4:11.7 }                  S E
      History 1: { v0:-1.0, v1:2.0, v2:3.0, v3:5.0, v4:9.0 } (inherited)    S E
  ▾ Disease > Infectious subclin. period: Small Dairy
      Models: is3 B R
      Inherit: { v0:-1.0, v1:1.0, v2:2.0, v3:3.0, v4:5.0 }                  S E
      Current: { v0:-1.0, v1:1.0, v2:2.0, v3:3.0, v4:5.0 } (inherited       S E
```

Figure 4.7: *The parameter values summary. Users can review prior values, undo changes, and jump directly to each parameter in the menu. Changed entries are highlighted in orange and changed values are bold.*

Each unique parameter has its own entry on the overlay. This entry shows the value that would be inherited from the previous day for comparison, the parameter's value on the selected day, and the last five values set on the selected day as a change history. When reviewing the first day of the timeline, the potentially inherited value is the default value obtained from the base ADSM scenario. A primary use of the summary is to review value changes, so where the previous day's value does not match the selected day's value, the parameter will be highlighted in orange as changed, just as changed days on the timeline are. Next to the various values are “set” and “edit” buttons. These adjust the value of the parameter of the selected day to match the given value. Setting the value to that of the previous day’s will cause the value to be inherited. Setting the value to one in the history effectively reverts changes made to the parameter. Clicking one of the edit buttons will perform the corresponding set action and then open the parameter within the parameters menu for editing, directly navigating to it. In such a case, the correct state and production type will automatically be selected as well. Changes made using the menu apply immediately to the selected day, as they do normally.

The parameter entries also show which models make use of each parameter. This feature of the summary overlay allows users to identify parameters that are used by outputs of interest. As an example, a planner interested in improving predictions for the number of latent units could quickly identify the parameters that have a direct impact on the predictions of the latent model. Currently, the single letter code for a model is listed under the parameter’s name if the parameter is used by that model. The letter codes use the same color scheme as the map and charts, allowing for easy identification.

The organization of the summary overlay differs from the parameters menu. Instead of nesting parameters within category menus, all parameters are listed on a single level. Parameters are prefixed with the name of the category they appear in within the parameters menu. Each production type variant of a value also has its own entry in the summary list. Since each state can have its own value for each of the parameters, an entry is listed for each of the states. For ease of finding the parameters, they are first sorted by state alphabetically, and then by the order in which they appear in the parameters menu. Each state can be individually collapsed to hide all of the listed parameters from that state, allowing users to review the values for just the states of interest. Additionally, each parameter entry can be individually collapsed to show or hide information about that parameter. Which entries are collapsed is remembered when closing the overlay and opening it on another day. In addition to this, the contents of the summary overlay can be filtered. First, users can choose whether to show only changed parameters, or all parameters. Next, users can filter the parameters for those used by specific models. Users are able to simultaneously choose any of the available models for this purpose, and can even filter for parameters that are not used by any models. Finally, users have the option to only include parameters with summary names that match a search string. This can be useful for filtering by category, filtering by production type, of if a user is having trouble finding a specific parameter in the menu.

4.5 Cadence Parameters Study

All tools come with a learning curve, but we aim to reduce ours where possible. One of the best ways to reduce the learning curve is to develop and incorporate an intuitive user interface. This may sound straightforward, but in reality it involves tackling design challenges not present when solving logical and algorithmic problems.

A major issue occurs when trying to determine whether the interface is actually intuitive. While advancing methods to test intuitiveness is a subject all its own, one approach commonly taken is to analyze a test group's interactions with the software in question. Finding the right participants for such a task is important. Since different users will have different knowledge and expectations coming in, what one user considers to be intuitive may not be intuitive for another user. Furthermore, a user's ability to use an interface evolves over time with training and use, so we are unable to use the same participants in a later study unless we are aiming to measure aspects like retention over multiple sessions. For the time being, the participants are pulled from the general student population and are not expected to come in with any knowledge on how to use the tool, what the inputs mean, or how to improve the outbreak situation presented to them. When Sonata matures into a final polished product, more professional end users will be included in the studies. Until then, we do not want to spoil their perception of Sonata by having them work with an unfinished development version of the tool.

The following section details the design and results of our user study on Cadence, Sonata's predecessor. It is followed by an analysis of how we believe we can improve the study when it comes time to test the intuitiveness of Sonata. As part of this study, we were looking to see if providing a small amount of information about the inner workings of Cadence to first time users would in any way impact their productivity with the tool. As the participants are students who have never worked with Cadence before, we get a window into the attitude and approach of new users. All participants perform a set of tasks designed to reveal the thought process involved when using the tool, especially for the first time. The tasks will resemble the tasks that a planner might take when using the tool, and will have multiple challenge levels useful for judging how fast a user is able to discover aspects of the interface and what steps they took while exploring it. It also exposes the challenges faced by planners that have not yet been trained to use Cadence by their supervisor.

4.5.1 Design of User Study

For our study, participants were asked to complete seven tasks, answering three survey questions on a 4-point Likert scale [68] for each. The questions asked the participants whether they understood the task, whether they thought they completed the tasks successfully, and whether they thought they completed the task quickly. We monitored the participants' actions and noted when they got lost in the interface or stalled while working toward the goal. Furthermore, we consistently graded the participants' success and speed on a similar 4-point scale for each task based on what we would expect of someone using the tool for the first time; we describe our grading criteria later on.

We divided the participants into two equally-sized groups. Participants in Group A received information about how Cadence works under the hood. Specifically, they were told that changes they made to the input parameters are passed to machine learning models trained using data from many runs of a disease outbreak simulation. Participants in Group B proceeded directly into the tasks without receiving this information. We were careful not to provide any training on how to use the tool so that no group was provided an undue advantage. Notably, neither group was taught how to navigate the menus or where to find specific inputs or what those inputs actually controlled in the simulation.

Of the seven tasks completed by participants, four involved searching for specific inputs available within Cadence and the remaining three involved attempting to meet specific output goals. The rationale for the first four tasks was threefold: 1) It allowed the participants a chance to familiarize themselves with the layout and types of inputs before attempting to satisfy more complicated goals, 2) The wording of the tasks is similar to the directions a lead planner may give to participants during an actual planning exercise, allowing us to see how users react in that setting, 3) By observing the path participants took through the input tree, we are able to judge how intuitive our input layout is, and help guide any future rearrangement. The remaining three tasks were meant to give us a better understanding of how individuals attempt to manipulate the Cadence inputs to meet each goal. We hoped to determine whether new participants would approach the goal randomly, or use their understanding of the tool to make more reasoned input decisions. Furthermore, we wanted this category of tasks to reflect scenarios presented during actual training sessions. A summary of the seven tasks given to participants is available in Table 4.1.

Table 4.1: *A summary of the seven tasks given to participants.*

Task	Description
T1	Find the input parameter controlling the number of detections before vaccination begins
T2	Find the input parameter controlling the probability of successfully tracing a direct contact for cow calf
T3	Find the input parameter controlling the delay for test results for cow calf
T4	Find the input parameter controlling the depopulation capacity
T5	Reduce the outbreak duration by adjusting any parameters
T6	Reduce the number of farms and animals depopulated
T7	Reduce the disease duration by at least 4 days while vaccinating at least 1000 fewer farms

4.5.2 Participants

We report on 30 participants that took part in our study. All participants were college-aged, with a mean age of 21. Of these participants, 9 were female and 21 were male. The participants were asked if they had any experience in areas related to Cadence: nine had some experience with machine learning, five had experience with simulations, and only one had experience with epidemiology. Most of the participants had significant experience using a computer, ranging from 2 to 20 years with an average of 10 years.

The general lack of in-depth knowledge of machine learning matches our target planner audience well, allowing us to accurately gauge the impact of describing the tool's components to planners. We generally expect actual planners to perform slightly better than this group of participants due to their background in epidemiology.

4.5.3 Common Behavior Trends

While conducting the study, we observed a number of behavior patterns shared by many participants. One such behavior was the tendency to get stuck within a category while searching for an input. They would continue revisiting explored inputs, seemingly get frustrated, and then pick the wrong input without checking the other categories (which are all visible when they start). Another common behavior was that participants checked the influence of the input parameters in a seemingly random order with no clear strategy. As they jumped from category to category, they unknowingly left drastically altered inputs in their path that had the potential to negatively impact future predictions they receive from the tool. Our study was not designed to capture individual strategies, something we hope to accomplish as future work.

Prior to reviewing the data, participants seemed to take similar actions and make similar mistakes regardless of whether they received additional information or not. This was unexpected, as we hypothesized the participants in group A to have a better understanding of how the tool operated and use that understanding to make more intelligent decisions.

4.5.4 Qualitative Results

During analysis, we assigned values to the responses provided by participants: 2 for strongly agree, 1 for agree, -1 for disagree, and -2 for strongly disagree. Similarly, we consistently gave scores to the participants based on how close they were to the goal. For the first four tasks, we gave a score of 2 if the participant found the correct input, -1 if they were in the correct category and chose the wrong input, and -2 if they were in the wrong category. For the remaining three tasks, a score of 2 was given if the participant reached or exceeded the stated goal, 1 was given if they were approaching the goal, but did not reach it, -1 if they did not make significant progress, and -2 if they moved in the opposite direction. Additionally, we considered their level of success toward reaching the stated goal and the amount of time taken to complete the task as separate. The recorded level of success toward reaching a given goal is not influenced by the time taken, and successfully reaching the goal does not necessarily end a participant's involvement with the task.

Table 4.2: *Shows the average researcher-assigned success scores for participants given (A) versus not given (B) information about Cadence, ranging from 2 to -2. The last row shows the delta between the two groups.*

Group	T1	T2	T3	T4	T5	T6	T7
A	0.7	1.7	2	1.7	1.3	0.3	0.8
B	-0.3	0.6	1.7	1.2	0.6	0.3	0.9
Delta	1	1.1	0.3	0.5	0.7	<0.1	0.1

When inspecting the data, a pattern emerges. For the first two tasks, participants in group A achieved the goals more successfully on average than those in group B, as can be seen in the first two rows of Table 4.2. T1 was the most failed task, with many participants thinking they had found the correct input, when in reality they were searching the wrong category. Most of the failures were simply due to unfamiliarity with Cadence, which is reasonably expected for first time users. The vast majority of participants grasped the mistakes they had made on this first task while working through T2, and by T6 both groups had nearly the same rate of success toward reaching the stated goal of each task.

Despite the increased similarity in goal success with each additional task, the time taken to successfully complete the tasks remained different. Table 4.3 contains the 50% and 99% percentile of time taken to complete tasks in seconds. As shown in the table, the participants in group B took longer to complete the tasks, upward of 45% varying with task complexity. Given the information we provided participants, it seems highly unlikely that one group was better trained in terms of navigating the interface or finding optimal solutions. It seems plausible that group A was primed to identify the logic behind the categorization, while those in group B did not put as much focus on it. Perhaps this explains the behavior where some group B participants would get stuck in the wrong category menu and continue navigating deeper, while group A participants would quickly identify that they were likely in the wrong place and try a different category.

Table 4.3: *Shows the 50th and 99th percentile of completion times (in seconds) for participants given versus not given information. The percent increase in time taken by participants not given information is also shown.*

Group	T1	T2	T3	T4	T5	T6	T7
A-50	35	15	15	10	85	129	90
B-50	60	39	18	18	115	165	165
Diff	41.7%	61.5%	14.3%	42.9%	26.1%	21.8%	45.5%
A-99	135	102	58	20	178	325	288
B-99	278	145	48	39	289	286	375
Diff	51.4%	29.8%	-21.3%	48.6%	38.3%	-13.6%	23.1%

Consider now T5 through T7, the tasks with an output goal. Here too, group A completed the tasks more quickly. Unlike the first four tasks which had a unique solution, there are multiple ways to reach the goal for the last three tasks, some more effective than others. Those with information may have made the determination that any further time spent attempting solutions was unlikely to result in a significant stride toward the goal, while those in group B remained hopeful of improved results. Although group A participants save time, they have a higher chance of skipping inputs that happen to be more effective. The majority of participants in both groups tend to stick with the first ``good enough" solution.

The results for how successfully participants believed they were at reaching the goal is depicted in Table 4.4. Reviewing the data, there does not appear to be a strong relationship between a participant receiving information about the tool's internal workings and a change in reported confidence in their results. What can be said is that participants in group A seem to be slightly more confident in their initial performance, as can be seen in the T1 results.

Similar to the results seen with actual success rates versus actual completion times, the participants' beliefs on how quickly they completed tasks diverged between the two groups. Participants in group A believed that they completed tasks quickly more often than participants in group B, as can be seen in Table 4.5.

Table 4.4: *Shows the average survey scores provided by participants for how successful they thought they were at completing each task, ranging from 2 to -2. The bottom two rows show the average difference from the researcher-assigned success scores.*

Group	T1	T2	T3	T4	T5	T6	T7
A	0.8	1.3	1.6	1.7	1.1	0.3	0.6
B	0.2	1.6	1.8	1.7	0.8	0.6	0.7
A	0.1	-0.4	-0.4	0.0	-0.2	0.1	-0.2
B	0.5	1.0	0.1	0.5	0.2	0.3	-0.3

Table 4.5: *Shows the average survey scores provided by participants for how quick they thought they were at completing each task, ranging from 2 to -2. The bottom two rows show the average difference from the researcher-assigned success scores.*

Group	T1	T2	T3	T4	T5	T6	T7
A	0.5	1.2	1.5	1.7	-0.4	-0.8	0.4
B	-0.5	0.8	1.1	1.3	-1.0	-1.1	-0.5
A	0.7	0.2	-0.2	0.1	-2.3	-1.0	-0.3
B	0.6	1.3	-0.3	0.3	-1.7	-1.2	-1.0

4.5.5 Emergence of Gamer Behavior

Although Cadence can be treated as a game, we did not present the tool as such to the participants. Despite this, 10 of the participants started playing with the tool at some point during the study. We labeled participants as gamers based on changes in their attitude toward the tool: generally being suddenly more excited by the changes in output each input invoked and being interested in locating inputs with greater impact. This mode of interaction is commonly referred to as *playfulness* or *gamefulness*, depending on the structure and rules of the activity [69]. The gamers were more motivated to continue trying even when encountering a run of insignificant inputs. One participant that started displaying this mentality during T7 decided to completely reset all of their input changes for the task and then proceeded to surpass the goal in under a minute. Some gamers even asked for the URL to Cadence so that they could continue playing with the tool on their own time. This is in contrast to most of the participants, which largely stuck with the first good combination of inputs they found, even if the goal had not been fully reached.

We were not expecting the natural emergence of gaming behavior so soon during our study. The reasons for this quick emergence must be multiple. It may be that the challenge of the task naturally presented itself as a game to some people, and some of the influence for gaming behavior could fall to the individual personality of the participants. Considering that one third of participants stated displaying gamer behavior within ten minutes of using Cadence for the first time, it seems reasonable that tool design played some part. For the majority of these participants, their change in demeanor occurred during T6. Seven participants from group A were identified as gamers, while three emerged in group B. It is possible that those with information were more confident and willing to take the risks inherent with gaming, but the small numbers make it difficult to draw any firm conclusions.

Looking at the task success rates of gamers versus non-gamers from Table 4.6, participants with a gamer mentality were clearly more successful in achieving the goal of the task. When considering that the difference in success rates between T6 and T7 barely changed between the groups that received information and those that did not, seeing such a drastic change on these two tasks is noteworthy. The time gamers took to complete these two tasks is slightly longer than the group that received information, as seen previously. This is due to the tendency of gamers to spend more time experimenting with inputs in an attempt to better their score.

Table 4.6: *Shows average researcher-assigned performance scores for participants identified versus not identified as gamers, ranging from 2 to -2. The last row shows the delta between the two groups.*

Group	T1	T2	T3	T4	T5	T6	T7
YES	0.0	1.6	2.0	1.7	1.1	1.1	1.6
NO	0.5	1.1	1.8	1.4	1.0	-0.1	0.5
Delta	0.5	0.5	0.2	0.3	0.1	1.2	1.1

4.5.6 Lessons Learned

Although the study was a decent attempt at measuring the intuitiveness of Cadence, there is much room to improve our methodology. One of the biggest shortcomings of our analysis was our relatively small sample size given the two groups and questionnaire style. It turns out that to have the statistical power to be sure that the results are significant, we would need at least three times the number of participants. Given that HCI is not the focus of our research group, finding and interviewing dozens of participants has an extremely high time cost, time that could be better spent on improving the tool itself. As the capabilities of Sonata are significantly more complex than Cadence, we will need to better structure the interviews so that we can cover more aspects of the tool in less time.

We also need to improve the tasks themselves. For instance, we should try to provide more focused goals to make the tasks competitive. We noticed some unexpected emergent "gamer behavior" in our first study that reviewers found interesting, we may be able to capitalize on this to produce some interesting results. Furthermore, we may need to provide the tasks to participants in a random order to help reduce the influence that learning has on the results of tasks completed later in the interview.

Our scoring system combined with our choice of a 4-point scale may have hurt the usefulness of our results. When collecting the data, we should look into using standardized questionnaires commonly used in literature, such as NASA-TLX [70]. This will help evaluate the significance of the results, as well as compare the results to work in other areas. Furthermore, we need to ensure that more emphasis is put on task completion times and error rates, as opposed to subjective results that are difficult to compare.

5 National-Scale Model

Constructing a national-scale visualization of outbreak behavior is an important design goal. In spite of this, implementing such a model is one of the most challenging tasks as we do not have a single encompassing dataset or simulation. In this chapter we explore how we overcame this challenge, among others, to enable the creation of a framework capable of predicting outbreak behavior at multiple geospatial scales.

5.1 Single National Model

There are different approaches to predicting the spread of events at the national scale, each with their own benefits and shortcomings. In this section we look at a naïve approach to the problem, simple in its design but riddled with shortcomings that make it untenable for the chosen solution. Exploring these shortcomings allows us to construct a solution that can overcome the most troublesome roadblocks in order to meet our requirements.

Using a single model for predictions at the national scale at first seems appealing. Loading and evaluating one model is likely to have a lower resource footprint in terms of both memory and CPU utilization. A single national model only needs to be run once for each needed output, but this leaves open the question of how many outputs are needed to accurately reflect the outbreak. Sonata relies on knowing the number of involved entities in each of the health states, so models need to be run at least as many times as there are outputs. In order to be able to show where the outbreak was taking place with any reasonable degree of accuracy, we would need to run the model additional times to obtain what is predicted to occur within individual political states or groups of states. Using a single model at the national scale, this can be accomplished by training the model with data categorized with different state and region codes, and providing that code as an input during model evaluation.

When designing the solution, it is important to keep in mind that not all regions of the United States have the same outbreak risk or behavior at any given time. The model can largely account for this diversity at training time, provided the number of variables remains manageable. In conflict with this is our desire for users to be able to configure policy at the state level, requiring a set of user-configurable control parameters per state, effectively duplicating the national parameter set for each area of interest. Unfortunately, by requiring that the single model respect the several hundred parameters from each state, we end up drastically increasing the complexity of the resulting models. This has the cost of increasing the in-memory size of our model, as well as increasing the amount of time taken for each model evaluation. More troublesome, though, is that we are sure to run into serious dimensionality issues as a result of this complexity, damaging our prediction accuracy.

A single national-scale model that is able to satisfy the what-if requirements of Sonata must take into account hundreds of user-specified control policies, and as a result, must have enough training data to cover all of the combinations of these policies as well as their complex interactions over time. As the NAADSM and ADSM simulations are stochastic in nature, we would need orders of magnitude more simulation runs in order to collect enough data to reasonably cover all of the possibilities. Unfortunately, the collection time and storage requirements must also increase by several orders of magnitude in response. This scalability issue ultimately makes such a single model solution unfeasible.

5.2 Cooperative Regional Models

In the following section, we explore our solution for implementing outbreak predictions at the national scale. This solution involves the use of hundreds of models at the state and sub-state level, models that are capable of cooperating with each other to produce predictions at a larger scale. We will look at the process of determining these spatial divisions, the impact of using smaller models, and the mechanisms that enable them to share information over time.

5.2.1 Geospatial Sectors

To allow multiple models to cover a large geospatial area, like that of the United States, we first need some method to partition the land area. Using state political boundaries is a good place to start, especially since actual control policies can vary from state to state. The problem with this approach alone is that states have different sizes; the western states are typically larger than the eastern states, for example. The size of the land area being modeled is important in the context of disease outbreaks, as different locations have distinct short and long range disease spread characteristics. This arises due to differences in the distribution of production types, differences in the density of herds and farms, and differences in net commercial imports and exports. To eliminate size differences, we could partition the entire United States into rectangles with a fixed square mileage. With this approach, models are trained to predict the behavior within their specific rectangle. Unfortunately, many of these rectangles will cross State political boundaries and therefore contain multiple states, making it difficult to configure per-state policy parameters. Due to this shortcoming, we start with state boundaries and then partition the states into smaller pieces. Our approach involves selecting sector centers based on the size and shape of a state and then matching production units to the closest center using Euclidian distance based on their latitude and longitude.

We call each of the partitions a sector. We attempt to partition the area as evenly as possible, with large states containing more sectors and smaller states containing fewer. The sectors for a selection of states is shown in Figure 5.1. Currently, a large state like Texas has five or six sectors, while a smaller state like Iowa contains one or two sectors. Ideally, each sector would be the same size, but this is not strictly possible due to variations in the size of states. The number of sectors that we can support depends on the capabilities of the browser. Although modern computers ship with 16GB of RAM, most modern browsers do not allow us to make full use of it. Most browsers have a per-tab memory limit, and this limit is typically around 4GB. This restricts the number of models that we are able to load, especially since models consume more memory once loaded and unpacked. For example, a model with 2600 decision trees and nearly 39,000 nodes, such as the one for clinical in Colorado sector #2 consumes over 2MB of RAM after JIT optimization. Using this number, we consume just shy of 1GB of RAM for the models from just 12 states in the best case scenario. As we currently only have ADSM and NAADSM scenarios covering the South Western and Great Plains states, we can afford to have smaller sectors and more sectors per state. When we obtain more scenarios in the future, we will most likely need to make the sectors larger in order to accommodate the addition of sectors from other regions within the United States.

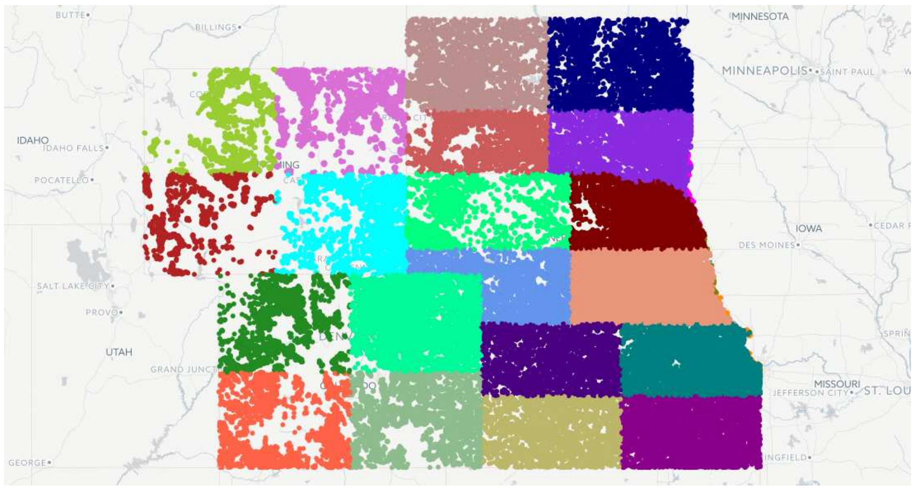


Figure 5.1: *Depicts sector boundaries for five states. The full herd population for Colorado, Kansas, Nebraska, South Dakota, and Wyoming is shown with a different color for each sector.*

Each sector is designed to be trained, loaded, and run independently of the others. Due to this fact, each sector has its own set of models that altogether predict the number of latent units, subclinical units, clinical units, immune units, vaccinated units, depopulated units, direct infections, and indirect/airborne infections in their designated area. This allows the sector-specific models to be fine-tuned for the behavior in the given area, such as through the use of different tree structures or by incorporating parameters that may not be considered significant in neighboring sectors or at larger scales. This fine-tuning increases the accuracy of the model predictions, helping them more closely reflect the source simulation data. Alongside this benefit, the use of independent sectors allows us to limit resource consumption to only sectors involved in an outbreak. As such, if nothing interesting is going on within a specific sector, we do not have to dedicate any CPU resources to it; the remaining sectors will continue to operate without issue and, importantly, without any change in accuracy. We also have the option to leave uninteresting sectors unloaded until something happens, conserving valuable memory space, but doing so leads to a noticeable loading delay when using the tool. Since sectors are independent from one another, we require a mechanism to share influences between them.

5.2.2 Sector-to-Sector Shipments

Sectors in Sonata will operate independently from one another if left to their own devices. They do not require input from outside sources in order to produce results, and Sonata is technically capable of supporting an independent localized outbreak in every sector. Despite this possibility, we are interested in how a localized outbreak turns into a national-scale outbreak, and what steps most effectively combat it once it happens. In order to enable the existence of a national-scale outbreak in Sonata, we require an additional component that permits phenomena in a source sector to influence external sectors. Such a mechanism permits all sectors to cooperate and contribute their predictions toward the visualization of an outbreak at a much larger scale.

The mechanism that enables the disease to spread across sector boundaries is the sector-to-sector shipments component. The shipments component is named for the real-world action of physically transferring animals from one area to another, and as this name implies, this component spreads exposure events between sectors by reproducing the behavior of real-world livestock shipments. In order for Sonata to be able to do this accurately, we need to know two pieces of information: 1) how often do infected shipments originate from a sector given current conditions, and 2) what is the likelihood that a shipment originating from a given sector will end its journey in a given sector.

As for the first piece of information, the likelihood of being the initiator of an infected shipment can be predicted by the inclusion of additional models. The ADSM and NAADSM simulations divide exposure and infection events into three categories of disease spread: direct, indirect, and airborne. The shipment mechanism is implementing the direct spread method, a potentially long-distance source to destination transfer. Using the events in the simulation data, we can train a model to predict the number of direct infection events that occur under a given set of conditions. The models not only take into account the parameters, but also the number of latent, subclinical, and clinically infectious units.

As for the second piece of information, the likelihood of a shipment ending up in a particular sector can be determined through information available in existing shipping datasets. The datasets are derived from real-world shipping records and historical import counts from various states. We use a county-to-county shipment dataset [71] from the USDA that uses Certificates of Veterinary Inspection (CVIs) as the primary data source [72]. CVIs include the source and destination address of the movement among other information, allowing the source and destination county to be determined; counties are then mapped to sectors for use in Sonata. CVIs are required for state-to-state shipments (except slaughters), making them reliable for tracking livestock transfers across state lines. For intrastate transfers, county-to-county connections are generated using a Bayesian distance kernel [73, 74].

The sector-to-sector shipment component starts by predicting the number of direct infections that originate from a sector during a visualized day. For each predicted infection, the destination sector is selected stochastically using a probability spinner. The weights in the spinner correspond to the distribution of shipments originating from the given source sector according to the county-to-county dataset. The number of infections and the selected destinations is stored in the output of the source sector's shipment computation for the day. To ensure that the models within the destination sectors do not attempt to make use of the results before they are fully ready, a synchronization mechanism is needed. Sonata evaluates models for every simulation day, predicting new results used to update plots and other visuals; it turns out that this daily evaluation cycle is a natural synchronization point for allowing sectors to communicate infection events. Instead of a traditional synchronization barrier, Sonata makes use of a Computation Dependency Graph (see Section 6.1.2) for synchronization. Using the graph, computations wait for all of their direct dependencies to finish updating before they evaluate, so in this case a destination sector will wait only for sectors that have the potential to ship infected units. Once it is safe for a destination to collect results, the results from all potential sources are queried and all infected shipments to the given destination are summed. This sum represents newly latent units on the next visualized day, and as such the sum is added to the prediction from the latent model evaluated on the following day. By creating additional latent units, the outbreak can be jumpstarted in new sectors. Infection events generated by the shipments component are visualized on the map views, as seen in Figure 5.2.

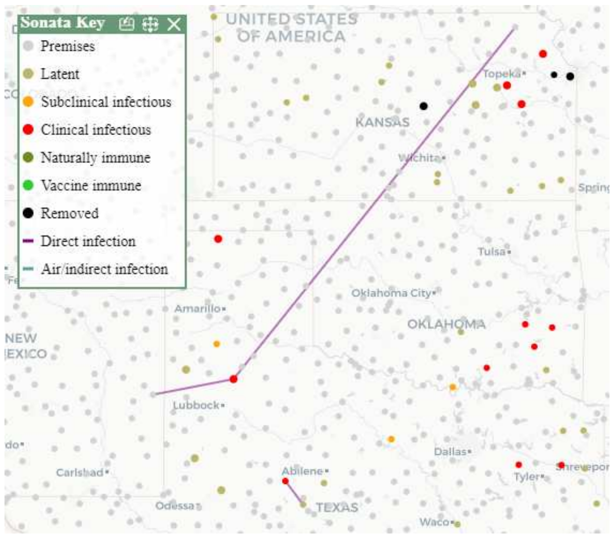


Figure 5.2: *Sonata depicts the source and destination of infection events on the map view. Above, shipments from Texas will result in new latent herds in New Mexico and Kansas.*

Since airborne and indirect are short-range spread methods, we currently capture their behaviors in the per-sector latent model. This means that airborne infections can not currently spread to neighboring sectors, even though there is an area along the edge of each sector where this would be possible in reality. To account for this window, we have built support for the use of an additional destination dataset for airborne and indirect exposures, allowing outbreaks to spread using the same mechanism as direct shipments. Allowed destinations can be computed from the exposure radius of each herd in a given sector's population, tracking the ratio of coverage across various sectors.

We ran into a couple of issues with our earliest attempts to model infection events. For example, we initially trained health-state-specific infection models, having a direct model for infections originating from a latent source, a subclinical source, and a clinical source respectively. Furthermore, when one of the models predicted a shipment, a unit in the given health-state on that day was removed from the source sector and added to the destination sector. After some review, we determined this to be incorrect behavior: the source premises remains infected after a shipment takes place as only a few animals are typically transferred. Furthermore, new infections always start out as latent. We now use a single model for predicting direct infections that uses the number of infected units in various states as input, but only starts latent infections. We also got caught off-guard when we added models for Louisiana. Despite our efforts, the outbreak did not seem to spread into Louisiana, which could have indicated a bug in our approach. It turned out that this behavior is normal due to differences in import, export, and movement patterns on a state to state basis, a lesson learned.

5.2.3 Coping with Additional Load

The cooperative approach that we have developed makes use of hundreds of individual models. This large number of models introduced a number of challenges in terms of loading and evaluation. Because of this, we have to use additional care to ensure that Sonata maintains its ability to produce visuals within a timely manner.

To help cope with the number of models we need to load, we bundle related models together and make use of compression. Each individual model is packed in a binary format, zip compressed, and finally base64 encoded to make the data safe for inclusion in the JSON files. These techniques help reduce the amount of time spent downloading models, but do not impact the amount of memory used by the models within Sonata. All loaded models within Sonata are kept in memory at all times in a fully uncompressed and unpacked form. We keep models unpacked so that they are ready for use immediately upon the user changing a relevant input.

Although we have all models unpacked and ready to evaluate at all times, we do not evaluate every model with each change. To start, only models within sectors with an active outbreak are evaluated. Sectors will become active only once the sector-to-sector shipments component indicates that an infected shipment is to arrive in that sector. Thanks to our Computation Dependency Graph, changes to parameters will only initiate the reevaluation of models that use those parameters, followed by models that use the result. Due to the complex web of interactions between models, however, most of the models will likely be reevaluated in a sector, as model outputs tend to be consumed by the other models on future days. One benefit of this behavior is that evaluation respects the flow of time and will start the day of the parameter change according to the timeline; days preceding the change will not be reevaluated. To help alleviate the strain on CPU resources during model evaluation, we additionally make use of Web Workers. This web API creates separate threads of execution, which enables models to be evaluated in parallel.

Since we only evaluate a subset of models in order to reduce CPU utilization, it is worth exploring whether or not we could reduce the memory consumption of these models as well. We could keep all models packed until use, but this has some undesirable side effects. Temporarily unpacking models from compressed bytes before model use not only requires that we keep the compressed version in memory in addition to the unpacked version, consuming more memory at peak times, but it also requires that we spend additional CPU resources to perform decompression and unpacking while that user is waiting for results. These are resources that would be in high demand at the time of decompression thanks to the many models waiting to evaluate. Furthermore, to reclaim memory used by the temporarily unpacked model code, we must wait for the browser's garbage collator to run. Constantly using and freeing memory is likely to cause additional slowdowns as threads wait for garbage collection. An alternative to this is to leave models compressed until they are used at least once, deferring unpacking from startup until actual use. This still requires additional CPU resources during model evaluation for unpacking, but this becomes a one-time cost. It also no longer suffers from extra memory utilization and is lighter on garbage production. We will likely switch to this approach should memory become tight after the inclusions of new states into Sonata.

5.2.4 Coping with Direct Infections

In the early phases of preparing the sector-to-sector shipment component, a concern arose that use of the component was leading to the double counting of infection events. This is a serious issue, as it leads to outbreaks spreading faster in Sonata than they would in the source simulation under similar conditions. The underlying cause of the issue is the fact that we were using both a shipment component that created new latent units as part of the infection process and a latent model trained on the appearance of latent units that arise through direct infection events within the simulation.

During discussions on how to resolve the issue, it was suggested that within-state direct infections should be handled by the latent model, and cross-state infections should be handled by the shipments component. Unfortunately, this was not a suitable solution for a number of reasons. To start, the two methods make use of different data sources. Specifically, the simulation uses a probabilistic distance-based approach, and uses the same distance distribution for all locations. The county-to-county dataset, on the other hand, starts with a network of real-world sources and destinations along with state-level import data, and fills in the gaps within that network using predictions from a Bayesian kernel model using shipment directions and distances that are specific to each location. Due to these differences, splitting intra-state versus inter-state infections using simulation-generated direct infection destinations would not only be less accurate, but it also introduces two separate behaviors for intra-state versus inter-state shipments. Considering the complex connections between models in Sonata, having multiple infection behaviors would make it harder to diagnose discrepancies in the visualizations produced by Sonata. To add to the confusion, as our sectors and their associated models are sub-state, we would still need to make use of the county-to-county dataset and the shipments component for within-state cross-sector shipments.

The way to resolve these conflicts is to use a single mechanism for all direct infections. As such, we opted for the more accurate shipment component backed by the county-to-county dataset. Unfortunately, the issue of double counting does not simply go away by using the shipment component for infections destined for both in and out of sector: the direct infections generated by the simulation are still included within the data used to train the latent model. To resolve this, we devised a way to exclude direct infections from the training data.

The trick to excluding direct infections from the training data lies in clever manipulation of the daily counts used to train the latent model. For context, we construct a dataset that consists of both parameter values for the scenario variant as well as relevant event counts extracted from the simulation output. To construct these counts, we process the simulation output day by day and track changes in health states on a per-unit basis and record the type, source, and destination of any infection events. After collection, the counts are partitioned by sector, with the infection counts partitioned by the sector of the source unit.

By then partitioning the original latent count (now called true-latent) into two categories, new-latent and existing-latent, we can differentiate between new direct infections and existing latent units on a daily basis. This partitioning is performed by deducting one from the true-latent count for every direct infection event arriving in the given sector on the previous day and adding it to the new-latent count; each one of those infection events resulted in one unit changing its state to latent in the given sector on the current day. The number of units remaining after this operation is the count of existing-latent, the number that existed on the previous day taking into account the natural decline in latent as units change health states plus the expected increase due to new local short-range airborne and indirect infections.

We only make use of the existing-latent partition; the new-latent partition is interesting when analyzing the simulation outputs, but is not relevant to Sonata due to the differences in datasets. The influence of these ignored infections are not missing in Sonata, however, as new latent units are added by Sonata's shipment component at the appropriate time. When we train the latent model, we use the true-latent values from prior days as inputs and the existing-latent count as the target output, as seen in Table 5.1. Doing so effectively removes the portion of latent units originating from a direct infection from the model's prediction.

Table 5.1: *Depicts latent event count partitioning. Note that existing-latent plus new-latent equals true-latent on the same day, and that true-latent from the previous day is used to predict existing-latent on the current day. On day 6, a latent unit became subclinical resulting in a drop from 7 to 6, behavior that the model can capture.*

Day	True Latent	Existing Latent	New Latent	Input → Target
1	0	0	0	0 → 0
2	1	0	1	0 → 0
3	3	1	2	1 → 1
4	3	3	0	3 → 3
5	7	3	4	3 → 3
6	6	6	0	7 → 6

This sort of manipulation is valid due to the way the models are trained and used by Sonata. The models map arbitrary previous states to a resulting current state and do not take into consideration how the previous states arose. This property is important, as otherwise the sector-to-sector shipment component would not be allowed to add to the number of latent units at what would be considered arbitrary times from the point-of-view of any given destination sector. Any cross-day discrepancies that are introduced into the training data are acceptable, as the models should still provide predictions best matching Sonata's current conditions. There is some risk of the model leaning to predict direct infections if we include the true-latent count from multiple prior days, so the latent model is restricted to only using one previous day as input to prevent leaking information.

Thanks to this approach, Sonata now uses just one consistent approach for all direct infections: the sector-to-sector shipment mechanism. In addition, since the sector-to-sector destination probabilities that we use for shipments are computed from real-world shipments, this approach is also more accurate. Furthermore, no changes to how Sonata uses the models were required: Sonata runs a model to predict the number of outgoing infections from a source sector and for each of the infections, Sonata stochastically determines destination sectors using a probability table (a source and destination can be the same). Sonata then runs the latent model to see how existing latent units oscillate, and to this output it adds the infections from other sectors (and itself).

5.2.5 Coping with Out of Sector Influences

It turns out that the solution we used to prevent double counting also allows us to solve a separate issue. When we partition simulation output data into sectors, there is still the chance that airborne and indirect infections have been introduced from neighboring sectors. If these are not accounted for, the models within Sonata are more likely to predict ghost outbreaks, that is, outbreaks that start or ramp up without proper cause. Thanks to the direct infection remedy, the fix for outside influences is straightforward: in addition to hiding direct infections from the latent model, we also hide indirect and airborne infections that originated from outside of the sector. Short-range, within-sector airborne and indirect shipments are accurately predicted by the latent model, as before. We also train a model that predicts the number of airborne and indirect infections originating from a sector, so that we can properly implement infection spread to neighboring sectors should it be required in the future.

Since we now hide all out-of-sector influences from the latent model, we experimented with a new approach to running scenarios. Previously, we were partitioning multi-state scenarios into their individual states and starting the outbreak in one sector at a time. Partitioning the scenario allows it to complete faster (useful when running millions of simulation runs), but there can be slight behavior differences due to fewer units at long distances. As we can eliminate out-of-sector influences, we are able to run the simulations with an initial infection in each sector, allowing us to run the entire scenario at once and dramatically cutting down the number of simulations that we need to run. Unfortunately, this approach skews the outputs toward larger and faster outbreaks. As such, there are fewer instances of small-scale outbreaks in the training data, so the models are less prepared to handle them as they arise in Sonata.

5.3 State Loading and Selection

Before Sonata is able to predict how an outbreak unfolds within a state, information about that state must first be loaded. Within this section, we look at the interface controls that allow users to check on the status of states and to select them for various operations. Furthermore, we describe the directory layouts, file structures, and various mechanisms that underpin the loading process.

5.3.1 Scenario Loading and Unloading

Sonata is designed so that state scenarios can be loaded and unloaded individually. This is made possible due to the independence of the sectors that make up the various states. Neighboring states do not need to be loaded for the sectors within a state to operate, and the predictions will be updated as neighboring states load in. The main benefit of this approach is that users are able to mix and match states originating from different simulation scenarios, even those developed by different subject matter experts and organizations. Thanks to this, users can load states from scenarios that target their particular planning needs, perhaps those that include a different production unit population, production types, base parameter probability distributions, or zoning rules. This is a powerful and necessary feature due to the dimensionality limitations inherent in the models we use; many of the scenario properties can only be modified accurately by changing the scenario. For example, Sonata only allows users to explore the values of certain probability-based parameters by stretching and translating the distribution, they are unable to change the underlying shape without loading a different scenario. Similarly, the importance and impact of various parameters is locked into the models at training time, and changing scenarios allows for models that are tuned specifically for the circumstances.

Sonata loads all of the information used by a state from JSON files. To start the state loading process, Sonata first loads a single JSON file that lists all of the available scenarios for each state along with path information so that the state scenarios can be requested and loaded later on. The file specifies the default scenarios to load when the tool starts and alternatives that are available for loading on user request. With this list of defaults, XMLHttpRequest is used to issue GET requests for the state JSON files. Since the loading process takes time by nature, the call to load a scenario accepts a callback function so that other components can be notified when loading completes. The JSON is parsed upon arrival, and information about parameters, models, shipment destinations, and the herd population is unpacked and stored in objects for each state scenario. Users are able to load and unload scenarios at runtime, through Sonata's interface. This is accomplished from a section in the options menu, which lists the unloaded state scenarios that Sonata knows about from the initial JSON file, as well as all the state scenarios that are currently loaded.

5.3.2 Scenario and State File Structure

To simplify the loading process, each state scenario has a single self-contained JSON file. This JSON file contains all of the information needed to start processing the state. By having just one file for each state, we eliminate the need to handle situations where loading waits for multiple files to finish, the files load out of order, or some of the files fail to load at all. To allow Sonata to mix and match states without requiring state-specific code, the state scenario JSON files have a well-defined structure. They include the name of the state, a boundary polygon for use with maps, and default parameter values that apply to the entire state. This is followed by a list of sectors defined within the state. Each sector contains the center point, a list of models, direct shipment-destination sector ids with probabilities, airborne and indirect spread-destination sector ids with probabilities, and population information. Each model entry includes the model in compressed form and a list of inputs that it makes use of. The production unit information includes the population size within the sector, a sample of units of various production types, and K-D trees for those samples. Finally, the files also contain information needed to estimate the number of depopulated animals from the number of farms/premises, an input needed for calling an external economic impact model.

All of the JSON files used by Sonata are generated by Python script from a variety of source files. To enable these scripts to operate without issue, we enforce a specific naming convention and scenario directory structure, with each scenario having its own directory. Thanks in part to these conventions, the scripts that generate Sonata JSON files only require the path to a scenario directory as input, and will automatically scan for, find, and load the input files required for processing.

We initially planned to only offer one version of each state early in development. Under this model, we used directories that would contain production units and models for each state, and parameters shared by all states. Whenever states from a newer scenario became available, we would completely replace those from the older scenario. Over time we ran into a few problems with parameters, as parameters from different scenarios are keyed differently due to differences in production types. For simplicity, we hardcoded which states used each parameter set as states were updated and replaced. This is no longer required, as the use of per-scenario directories along with the inclusion of local parameter defaults in each state's JSON file resolved these complications.

5.3.3 State Selection Map

Sonata allows users to select states for the purposes of configuring parameters, data views, and user roles. The original plan was for this to be done from the main map view: in selection mode, when users hovered over a state, the borders of that region would be highlighted and the area clickable. This turned out to be a poor choice, however, as the map view can have its viewport adjusted or closed by the user. Instead, we introduced a new mini-map to the right of the timeline, placing the interface for state selection next to that for day and branch selection. Like the timeline, selected states on the mini-map are colored teal and multiple states can be selected at once. When multiple states are selected, actions performed apply to all the states simultaneously. With so many states, constantly changing which ones are selected is a tedious task, so we plan on implementing configurable preset buttons and a history so temporary selections can be reverted. In addition to allowing selection, the mini-map also shows the status of the states. These include the availability of the states for loading, showing which ones are unavailable, unloaded but available, and loaded. The mini-map also has the ability to show which states have had their parameters changed, similar to the timeline.

5.4 Tiling and Clustering Herds

Being designed to produce visualizations for spatiotemporal data, Sonata must be able to visualize the entities that interact with and propagate spatiotemporal events in that data. Sonata must be able to accomplish this at the national scale, meaning that we must be able to handle upwards of millions of entities in a browser. In this section we explore the techniques we used to support the visualization of these entities.

5.4.1 Sampling Herd Entities

Computers are limited in their processing capabilities, with mobile devices generally being further limited in order to preserve battery life and reduce heat. Since we aim for Sonata to be deployable on tables for use in the field, this places a notable restriction on how much data can be visualized while maintaining responsiveness. Any hardware limitations are compounded by the memory and rendering limitations of current web browsers. Although it is difficult to reliably quantify the number of herds that can be displayed at once due to large variation in hardware and software configurations, browsers tend to start showing signs of struggle when the number of rendered entities (i.e. herds) exceeds 2000, at least when drawn using scalable vector graphics. This means that only a small subset of the population can be visualized at a given time, a subset that is significantly smaller than the 516,266 non-overlapping premises across 12 states that we currently have available to us from subject matter experts.

The naive solution to this problem is to simply sample the dataset, and this was our initial approach. We uniformly sampled 100 premises per sector totaling about 400 per state. With the 12 states that we had NAADSM or ADSM scenarios for, this is a total of 4,800 herds. For each of the selected herds, location, type, and size information is written to a population JSON file that is loaded by Sonata at runtime. Sampling the population prevents the need to download a large population file and helps the browser cope with interactive rendering, but has a critical disadvantage: a permanently low unit resolution. Maintaining 4,800 herds is a tradeoff between rendering performance and resolution. We make the assumption that the users will not always be viewing the entire United States, allowing us to have more herds available to us for computing visualizations. Since the browser has no knowledge of the remaining unsampled units, only the loaded sample is available for use.

It is desirable to sample the Interaction Graph, as we only need information about herds that we are able to visualize. Unfortunately, uniformly sampling the graph removes nearly all interactions between herds, as it is unlikely that a linked herd will be included in the sample. For similar reasons, visual patterns that tend to emerge in the full dataset such as clusters of nearby infected herds or vaccination rings are not clearly identifiable when the dataset is sampled down. To compensate, we began to weigh our sample toward interesting herds, i.e. those that can become infected or change state. Preferentially selecting herds in this manner has a downside in that it does not accurately reflect the distribution of healthy versus infected herds across the landscape.

5.4.2 Tiling Herd Entities

Due to the permanently low resolution and skewed interactions of the sampled population, we opted to switch to a combination of herd clustering and data tiling. Tiling is a well-known technique used commonly within online map visualization software. Tiling has the benefit of allowing a small subset of tiles to be stitched together to display an extremely detailed area without needing to download the area unseen. The process involves dividing the area into tiles addressed by geospatial coordinates (x,y) and zoom level (z). The coordinates are mathematically defined and closely related to zoom level. At zoom level 0, the entire Earth is projected to fit inside a single tile at (x,y) equals (0,0). For each increment in level from there, the number of tiles in both the x and y direction double respectively, therefore tiles cover one fourth of the area at 4 times the resolution.

Static images are the most commonly tiled medium. In fact, Sonata already made use of an open tile service to obtain static image tiles to display as the map background. It is also possible to tile geospatial datasets like our Interaction Graph, a technique that we use to our advantage. We tile unit data from zoom levels 4 to 12; this ranges from fitting the entire United States comfortably on a 1080p screen to fitting a single town or city. The unit data we currently have covers 39,502 tiles, but the beauty of the tiling approach is that the browser only has the ~32 most relevant tiles loaded for rendering at any given time. Figure 5.3 shows the tile address and entity counts for a selection of tiles loaded by a map view.

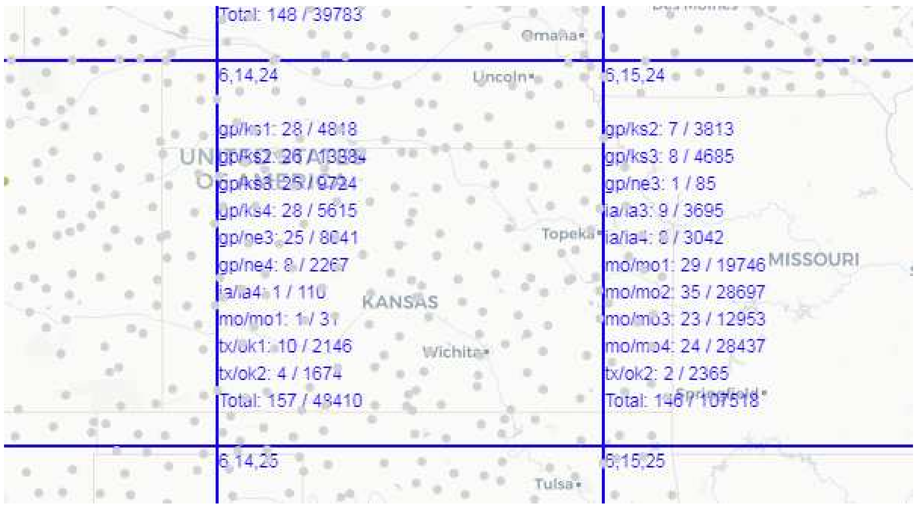


Figure 5.3: *A map view with tile boundaries enabled. The blue text details the number of visible entities out of the total population in the area covered by the tile.*

Typically, tiles are written to individual files that can be accessed by the browser on demand. Instead of writing the tiles to files, we instead insert the prepared unit data into a database indexed by tile address and separately by unit id. When Sonata queries for a tile, a server-side script quickly builds one from units in the database. Using a database is beneficial since we can include or exclude data for specific states or scenarios conditionally, and we can also query for the interaction data of specific herds by id. The time taken to generate and download a tile is roughly 100ms, but tiles can be loaded in parallel. The extra processing time does add to the delay experienced when loading new tiles as the user pans and zooms, although the loading time can be largely hidden by prefetching tiles of nearby areas.

5.4.3 Clustering Herd Entities

Tiling provided the opportunity to load localized data at a resolution tuned for the current viewing conditions. As such, to maintain balance between high resolution and resource utilization the goal is to keep the number of visible herds relatively constant: as more land becomes visible, the number of herds shown per unit of land needs to decrease to compensate. Unfortunately, the process of tiling itself does not limit the amount of data that can be included within a single tile. Without some form of sampling or clustering, every herd that falls within a given tile's assigned area will find itself included in the tile's data.

To achieve the goal of a relatively constant number of visible entities, we make use of clustering. Clustering algorithms group data points that are similar to each other, in our case where the entities are physically nearby. This is particularly helpful when revealing or removing additional herds while zooming in, as we can collapse and expand herds that are clustered together instead of random sampled herds. Alongside this, clustering allows us to visually represent entire clusters with a single entity at distant zoom levels. There are many clustering algorithms available, but we opted for Mean Shift [75, 76]. This particular algorithm clusters based on density and scales well with many clusters, making it an ideal choice for clustering a large herd population. Conceptually, Mean Shift creates a surface that is higher where data is denser. Data points climb up this surface to find the cluster they belong to, as depicted in Figure 5.4.

Our clustering and tiling approach is closely integrated. All tiles for a single zoom level are created at the same time, working from the highest level, 12, to lowest level, 4. Level 12 includes the full herd population and is the starting point for tiling and clustering operations. For each level after that, new tiles are generated using data from relevant tiles in the prior level. Specifically, data included in the four tiles from the prior level corresponding to a given tile on the current level is clustered and processed. After clustering, the herds closest to their assigned cluster's center are selected for inclusion in the new tile data, resulting in only one entity representing each cluster. References to the other herds within the cluster are maintained by this chosen entity for use later, forming a tree across levels.

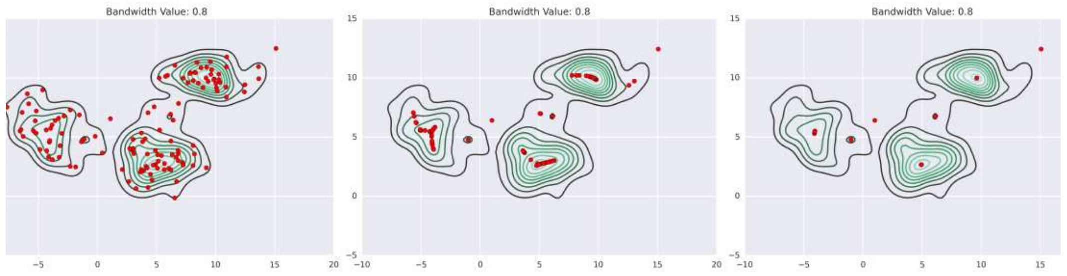


Figure 5.4: *Mean Shift clustering in action. Mean Shift clustering allows us to represent multiple herds in a densely populated area with a single entity.*
Image credit: https://spin.atomicobject.com/2015/05/26/mean-shift-clustering/

Following the tiling and clustering at all zoom levels, new herd ids are assigned. Starting with the herd entities contained within the lowest zoom level, we recursively assign ids by following the cluster membership reference tree depth first. Once all child entities of a given entity have been assigned an id, the range of ids is recorded in the parent entity. This range is guaranteed to be consecutive due to the depth first traversal, providing Sonata with a means to quickly identify the entity that is tasked with visually representing any arbitrary herd at any zoom level. Figure 5.5 shows an example id hierarchy for an entity. As can be seen in the figure, new entities split off from their parent at different times, relating to how physically close the represented herds are to each other. Because Sonata operates on models at the sub-state sector level, units are tiled, clustered, and assigned ids per sector.

5.4.4 Making use of Tiled Data

When designing the tiling and clustering solution for Sonata, we had three primary requirements to satisfy. These requirements were as follows: 1) allow dynamic herd resolution at various levels of zoom, 2) show and remember which herds are involved in an outbreak at all zoom levels, and 3) allow the propagation of various interactions beyond the in-view area. The first of these requirements is satisfied by the combination of tiling and clustering, but this solution ultimately made meeting the remaining two requirements a bit of a challenge.

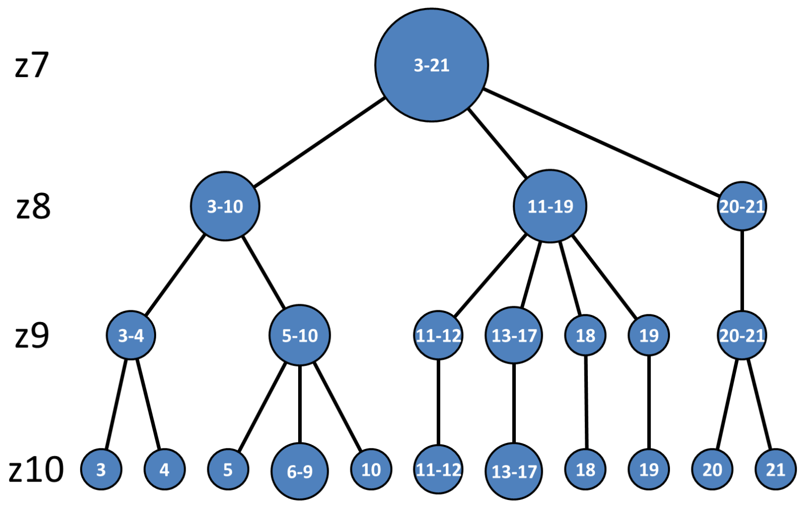


Figure 5.5: *The id hierarchy for an entity selected from Colorado, zoom level 7 to 10. Circles are entities displayed at the given zoom level and the numbers within are the id ranges covered. Note how new entities split off their parent at different rates, depending on how physically close the represented herds are to each other.*

We accomplished showing the presence of infected herds at all zoom levels through a combination of techniques. After clustering, single entities represent multiple herds visible at closer zoom levels, building a hierarchy spanning level to level. These collapsed entities have enough information that Sonata can make visual decisions that are valid even as new tiles become loaded or unloaded. Among this information is the range of all consecutive herd ids that the given entity is representing at this zoom level. In addition to this, Sonata maintains a list of involved herds independent of which tiles are loaded. These involved herds are referenced unambiguously by their sector-unique herd id. Using this list of unique ids and the ranges associated with the entities contained within the currently loaded tiles, we can determine the correct visual state of every visible entity. For example, Sonata can determine that a specific entity should be displayed as vaccinated, even if the location information for the actually vaccinated unit has not been loaded yet. If the user pans and zooms to the location where that unit becomes visible, it will be displayed as vaccinated. Sonata is able to perform this matching efficiently for large populations by sorting both the visible population and involved unique ids. A binary search on the visible population is performed to find the correct entity, and then the list of involved herd ids is iterated to collect the health states that need to be represented by the entity. The visual state of herds is updated every time a new set of tiles is loaded or unloaded.

Propagating interactions beyond the visible area was the biggest challenge. We start off by ensuring that we always have enough information available to make global visualization decisions. As such, we always have all units at zoom level 7 loaded. This is about four times as many units as were loaded with our sampling approach, except these units are not drawn on the map so we do not suffer a performance penalty for their inclusion. This is enough information for some interactions, like vaccination, as the destination can be unambiguously determined by stochastically selecting an entity within a given radius and stochastically selecting an id from its range. We optimize the search for units within a particular range through the use of K-D trees [67], a space partitioning data structure that supports efficient nearest neighbor search. If use of the Interaction Graph is desirable, we can query for the full-resolution interaction information of already involved herds directly from the tile database. Such background querying poses an issue, however, as map computation processing and the resulting visual updates must wait for interaction data to be downloaded. The problem is compounded by the fact that the map data of the prior day is required as input to compute the next day's data. Fortunately, infection events are relatively rare and can originate from the same source multiple times, allowing for data reuse. Loading the infection information in the background is important, as we cannot rely on the user to load adjacent regions through panning. Furthermore, we cannot ignore or defer processing of infections that exit the view area, as there is a chance that an involved out-of-view source unit could infect a unit within view on a later date. We considered dynamically reducing the resolution of interaction data and including it within the tiles at the various zoom levels, but this introduces ambiguity in which herds are likely infection candidates and does not solve the issue of infections spreading out of view. Our current solution is more accurate in this regard, and works in both cases.

6 Computation and Model Evaluation

In order to make timely predictions, Sonata needs to evaluate hundreds of outbreak models as quickly as possible, ideally finishing in under a second. To make this possible, Sonata includes a framework for evaluating models and other tasks as computations that are evaluated in order and only as needed. The following chapter describes our techniques for computation and model evaluation.

6.1 Data Computations

Sonata divides the workload into small computational tasks that can be scheduled and run when needed. In this section, we explore the useful features of Sonata's computations and how we ensure they run in a safe order.

6.1.1 Computations as a Data Source

Sonata includes a variety of ways to visualize computed data, including maps, charts, and tables. Each one of these views requires data in order to create and update the elements that make up the visualization. In order to obtain the data that they need, views subscribe to a number of computations, which behave as data sources. When a computation evaluates, any views that subscribe to the results will be notified, allowing them to update their contents. We have designed our computations to perform lightweight tasks that produce generalizable results. The production of generalizable results allows the computation outputs to be used by a variety of view types.

Computations are not only used as a data source for views; they can also produce inputs used by other computations. In this scenario, computations subscribe to one or more other computations, allowing them to be reevaluated whenever inputs change. For example, the computations that are used to evaluate our outbreak models subscribe to parameter values on their assigned day as well as model outputs from previous days. Altogether, the computation subscriptions result in a web of dependencies starting from the open data views and connecting back through every computation whose value is required to ultimately construct those views. We call this web of dependencies the Computation Dependency Graph.

6.1.2 Computation Dependency Graph

Due to the scale of the model evaluation problem within Sonata, we need to use techniques beyond model efficiency improvements to ensure that we make the most of the limited computing resources available to us. For this reason, we make use of our Computation Dependency Graph within Sonata, a data structure that enables proper and efficient scheduling of computation evaluations. The Dependency Graph consists of a directed acyclic graph denoting the input requirements for each computation accompanied by bookkeeping information. Internally, the Dependency Graph is permitted to contain multiple graph partitions, which indicate that there are no ordering requirements between computations appearing in separate partitions.

The Dependency Graph is constructed automatically as computations are introduced and removed at runtime. When a computation or view is created, it obtains a reference to computations that it requires so that it can collect input values during evaluation. If a requested input computation does not yet exist, it is created (recursively) at this time. Shortly after a computation or view obtains a reference to an input computation, it will register itself with that computation so that it may be notified of changes to input values. This registration process creates edges in the graph. Computations remain in the Dependency Graph as long as they are being referenced by any other computation or view. Orphaned computations remain temporarily in order to prevent deleting cached results when releasing and re-registering inputs, as can happen when loading or unloading sectors. Sonata periodically deletes unused computations, unregistering any inputs in the process.

By tracking these dependencies, Sonata is able to only run models and other processing code that are needed for an open view, and then only once, even if it is used multiple times. The bookkeeping information allows Sonata to determine how many upstream input computations still need to be evaluated before a given computation can be safely evaluated, preserving ordering guarantees. Additionally, if a computation evaluation results in no changes, we can short circuit evaluation of large sub-graphs of downstream computations. Thanks to the Dependency Graph, we have the ability to process updates from the bottom up in the background, or from the top down when data in needed immediately by a view.

6.1.3 Output Caching and Reuse

Needing to continuously reevaluate models and the visual components that rely on them puts strain on the limited resources of the host browser. Thankfully, caching and reusing outputs is a significant way to combat this. In Sonata, the result value produced by every computation is cached, allowing it to be used by other computations and views at a later time. By using this method alongside dependency tracking, we only need to evaluate a model once even when multiple views request the value at different points in time. Some of the intermediate results used by views such as aggregations, filters, and other transformations are computed as computations, allowing these intermediate values to be cached and shared with other views. Any views for the same branch-day can simply access the cached value instead of reprocessing the data. In Sonata, the result value produced by every computation is cached indefinitely. This policy results in more memory consumption, but the amount required is tiny compared to other memory consumers in the tool. The benefits of caching appear to greatly outweigh any negative impact the memory consumption has on application performance.

In addition to the performance benefits of only-once computation evaluation, maintaining cached values provide further benefits by short-circuiting unnecessary re-evaluations. Computations in Sonata are stateless, meaning that if there is no change in inputs, there will be no change in outputs. Producing the same output value during a re-evaluation is a common occurrence, especially since model outputs are rounded. In such a case, re-evaluating any downstream computations would be a waste of resources, so Sonata relies on the existence of the cached value instead. Sonata uses a lightweight freshness counter to track whether input computations have changed, requiring re-evaluation.

There is another opportunity for short-circuiting when conditions are the same on two separate timeline branches. In such a scenario, two separate computations with the same configuration except for the branch are provided identical input values (these may be from separate computation instances). When two computations that perform the same operation receive the same inputs, they must produce the same output and therefore the value can be shared between them. By using a quick and lightweight method to identify a matching computation with the same input values, computations can copy their counterpart's cached value by reference during evaluation time instead of performing the full operation. Unfortunately, scanning through other computations' configuration and inputs to find valid counterparts is prohibitively expensive. Instead, an evaluating computation creates a branch-agnostic key to be used in a dictionary of cached computations. The key is unique for every operation and distinct set of input values. If the key already exists, then the result can be copied, otherwise the computation adds itself to the dictionary for others to find.

Caching has another powerful use within Sonata, thanks to the ability for users to advance and rewind time at will using the timeline slider. Many of the views utilize conditions on a previous day and perform additional computation to bring it to an updated state, notably the maps. In the forward direction of time, we only need to evaluate models for timeline days including and after the day of the changed parameter. This is obvious since events in the future cannot influence events in the past. Unfortunately, due to lossy operations, the computation logic is difficult to perform in reverse. This means that without caching, we would need to restart the computation from the beginning for every time we reversed the visualized day. By caching the results of previously visualized days, we are able to rewind the visualization in milliseconds by recalling the stored state.

6.2 Computation Evaluation

Determining which computations are evaluated and when they are evaluated is the job of the computation scheduler. In this section, we look at our various scheduling approaches and the challenges that influenced them. We also explore the techniques and optimizations we employ to ensure that Sonata provides an interactive experience.

6.2.1 *Initial Scheduling Algorithm*

When we initially introduced computations into Sonata, we had not yet encountered many scheduling challenges. This was largely thanks to our reliance on a single threaded environment at the time, as well as the fact the computations were guaranteed to return results when run. Due to these conditions, we implemented two lightweight methods for obtaining updated values.

The first of these methods was by direct request. A direct request occurred when either a computation or view requested the value of an upstream computation. If that value was cached, it would be returned immediately. If the value was missing, however, the upstream value would be evaluated at request-time. This would in turn recursively request values from computations further upstream, leading to the immediate processing of all necessary dependencies. This top-down processing tended to occur when the tool first started or as views were opened, and it was the fastest method of providing initial results to the views in question.

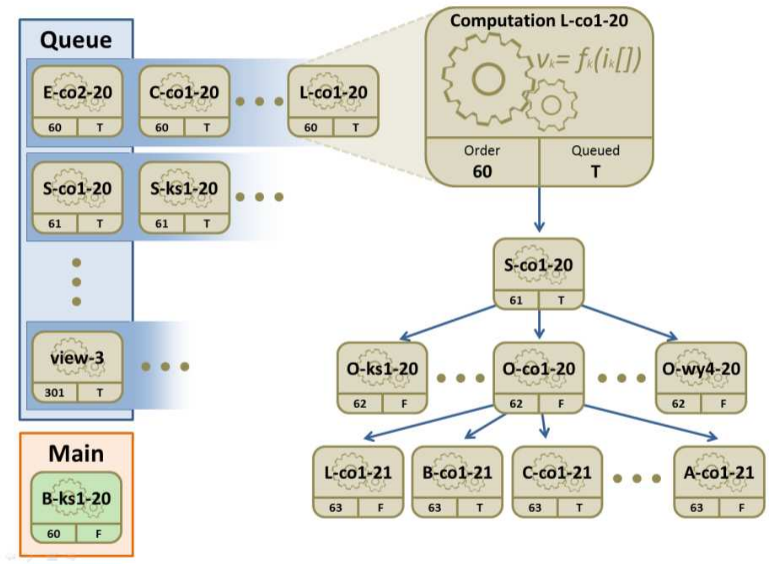


Figure 6.1: *A snapshot of day 20 evaluation activity using our first computation scheduling algorithm. Tan boxes represent computations, with bookkeeping information at the bottom. Computations are queued according to their order value. Only one computation can be evaluated at a time.*

The second evaluation method was background scheduling, shown in Figure 6.1. As computations registered their use of various upstream computations, in turn building the Computation Dependency Graph, an extra “order” bookkeeping value was tracked. The order value of a computation was set to the maximum order of all dependencies plus one. Computations were always scheduled to evaluate from lowest to highest order value, guaranteeing that all the upstream computations required by any given computation finished prior to its evaluation. This was accomplished through the use of a priority queue. To prevent constant sorting whenever computations were enqueued, the queue consisted of a sorted list of buckets assigned an order value such that the next computation to be evaluated gets obtained from the first bucket. The computations within a single bucket all had the same order value but are themselves unsorted. When a computation evaluation resulted in a value change, all immediate downstream computations were queued for re-evaluation. No computations get queued if the value did not change, reducing resource waste.

6.2.2 Thread Limitations and Parallelism

There is an important behavior of web browsers that we have to take into account when designing the computation scheduling algorithm relating to a user’s ability to actively interact with the tool. By default, all JavaScript is run in a single main thread, and all code interacting with the window must be run in the main thread. This means that mouse and keyboard event handling, the rendering of animations, and other important maintenance tasks have to compete with our JavaScript code for time on the main thread. If care is not taken when running computations, we can end up starving other tasks and make the browser window completely unresponsive. To handle this, our background scheduling algorithm will yield after a one thirtieth of a second has passed; allowing other tasks a chance to run.

Early in Sonata’s development, we used the recursive direct-request computation evaluation technique to obtain results during tool startup. This had the benefit of reducing computation overhead during initial loading, but had the notable downside of not being interruptible, i.e. no other tasks had the chance to run. This ended up making the tool completely unresponsive to the point where interface elements were not getting rendered during the first few seconds as there were several thousand model evaluations to complete before the first results were displayed. We initially attempted to remedy this by completely initializing all interface components (maps, timeline, and menus) before running any computations. To our surprise, this initial change did not fully remedy the problem, and we ended up switching to iterative background scheduling for initial model evaluation. With this change, initial model evaluation was taking upwards of 20-30 seconds, a duration that we considered unacceptable. To improve evaluation speed, we explored running models in parallel.

6.2.3 Parallelism with WebWorkers

As we increased the number of sectors available in Sonata, it became increasingly apparent that it was no longer feasible to run the model evaluations within the main JavaScript thread. As such, we looked into the possibility of having model evaluations distributed across multiple background threads for extra parallelism. JavaScript allows the use of additional threads through the Worker interface, where each worker runs in a separate system level thread allowing computations to be evaluated simultaneously on the hardware.

Unfortunately, the worker threads come with some notable downsides compared to those of other languages such as C or Java. The primary complication is that there is no memory sharing between threads, most likely to prevent concurrency issues. Instead, all data is communicated via message passing, resulting in extra overhead in terms of a serialization and deserialization phase as well as the resulting deep copy of all objects being transferred. This places an unavoidable limitation on which threads can handle the evaluation of which models. In Sonata, we do not have the luxury of having multiple copies of models distributed across multiple threads due to the memory constraints we face combined with the number and size of our models. In other languages where objects are accessible across threads, we would have been able to have any thread evaluate any model from any sector. Unfortunately, in Sonata we have to distribute each model to just one thread. Only that one thread will be able to run the model, even if other threads happen to be idle.

Since the copying of input and output combined with thread scheduling results in a high overhead, we only push heavy computations such as model evaluations to the workers. Lighter computations, including aggregations, are still run in the main thread. Because only the main thread has access to the DOM and window, results from workers must be copied back to the main thread. The main thread is notified of results via registered event handlers, requiring that the scheduler yield and therefore increasing the total overhead. Despite these shortcomings, the performance boost from using workers has outweighed the increased overhead. The introduction of WebWorker threads cut initial load and evaluation times from 20-30 seconds down to 2-5, with the interface ready nearly instantly.

6.2.4 Current Scheduling Algorithm

As we moved from single threaded to multi-threaded computation evaluation, it became clear that our previous scheduling algorithm was bottlenecking model throughput. In a single threaded environment, we could only evaluate one computation at a time, and all stale upstream computations are reevaluated before anything that depends on them. This fact allowed us to guarantee that any computation queued after the current computation would have up-to-date inputs by the time it ran, allowing us to queue computations before they were technically ready to run. In the multi-threaded environment, however, we need to be able to distribute computations across threads for parallel evaluation. This poses a conundrum: 1) if we wait for computation results to be received in order to proceed, otherwise ready computations are blocked from running while waiting for the unrelated results; 2) if we immediately proceed to the next queued computation, then we lose the guarantee that the computation's inputs are up-to-date, as there are computations ahead of it that have not yet finished. We needed to take a different approach to scheduling.

We solved the issue by adding additional bookkeeping information to the computations, and upgrading the algorithm to compensate. We now associate wait counters and freshness counters with each computation, as seen in Figure 6.2. The wait counters prevent computations from running early while upstream computations are still stale, and the freshness values are used to prevent a chain of re-evaluation when the inputs did not change. As these values need to be maintained over time, the new algorithm increases the queuing overhead slightly.

In order to prevent a computation from running that is several hops downstream from a stale computation, the wait counters must be incremented on all downstream computations. Updating the wait counters is the responsibility of the queuing algorithm, which is now iterative so that it can continue updating computations until no bookkeeping changes are required. The queuing algorithm is broken down into two simple cases per computation (i.e. per iteration). 1) If the up-wait counter is zero (no stale upstream computations) and the up-fresh counter (incremented if an upstream value changes) is less than or equal to the computation's freshness counter, we decrement the up-wait counter of the immediately-downstream computations and push them onto the check-computation stack. Computations entering this case are ready to run, but do not need to; its dependents are checked instead. 2) Otherwise, if the up-wait counter is zero and the up-fresh counter indicates a change, then we mark the computation as ready to queue. No matter the previous check, each of its immediately-downstream computations has its up-wait counter incremented and is then added to the check-computation stack. The next computation is popped off of the check-computation stack and the next iteration begins. The scheduling lifecycle of a computation is depicted in Figure 6.3.

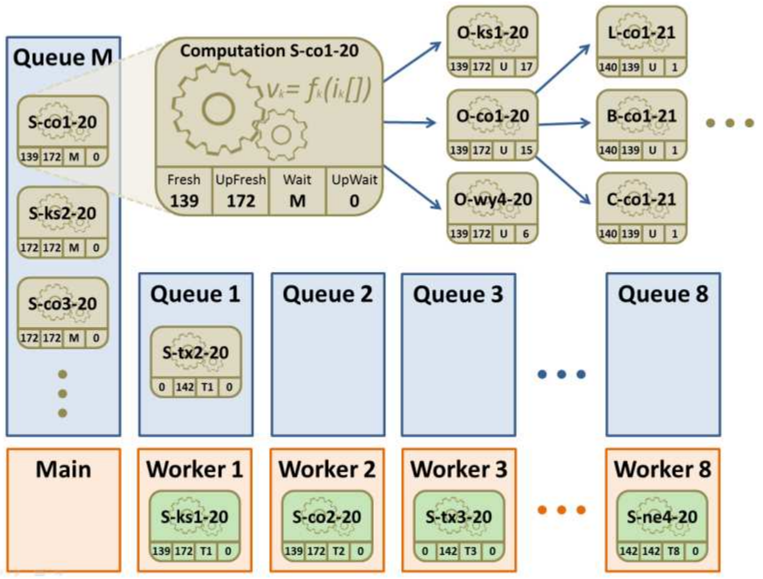


Figure 6.2: *A snapshot of day 20 evaluation activity using our current computation scheduling algorithm. Computations are only queued when all upstream computations have finished. Freshness determines whether or not to queue or skip a computation.*

It should be noted that some of the logic in the description above has been omitted for ease of reading. In reality, the up-wait counters only indicate the number of immediately-upstream inputs that are stale, not the total number of all upstream computations. This means that computations are only pushed onto the check-computation stack when the wait increment or decrement has not already been performed. Computations keep track of when they have already marked downstream computations, and when they are already in the queue.

With proper updates to the freshness counters, the queueing algorithm will properly mark and/or queue the submitted computation and all eligible downstream computations.

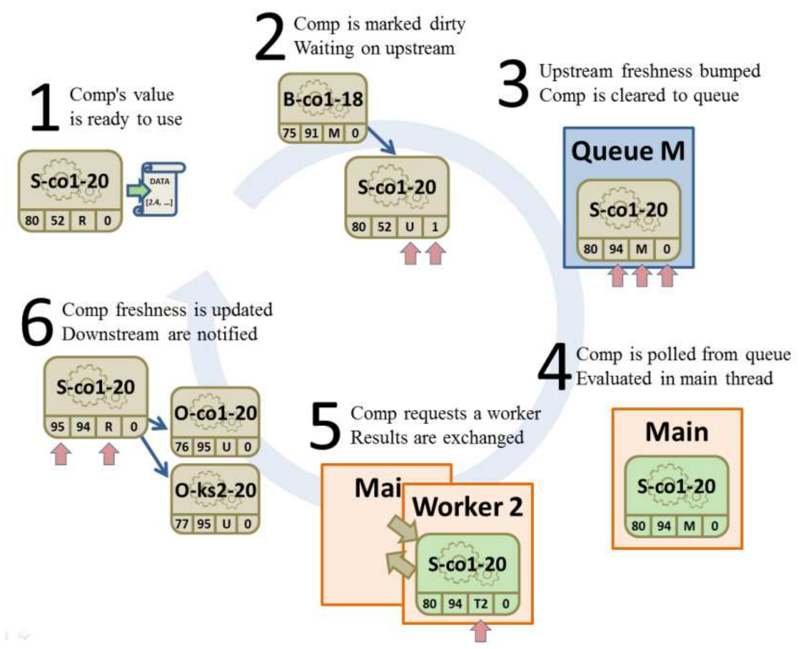

1 Comp's value is ready to use
S-co1-20
80 52 R 0
DATA
2 Comp is marked dirty Waiting on upstream
B-co1-18
75 91 M 0
S-co1-20
80 52 U 1
3 Upstream freshness bumped Comp is cleared to queue
Queue M
S-co1-20
80 94 M 0
4 Comp is polled from queue Evaluated in main thread
Main
S-co1-20
80 94 M 0
5 Comp requests a worker Results are exchanged
Mai
Worker 2
S-co1-20
80 94 T2 0
6 Comp freshness is updated Downstream are notified
S-co1-20
95 94 R 0
O-co1-20
76 95 U 0
O-ks2-20
77 95 U 0

Figure 6.3: *The scheduling lifecycle of a computation. A computations' bookkeeping information is used to determine how it advances. If upstream freshness remains unchanged, computations will advance directly from step 3 to step 6.*

Queued computations are scheduled to evaluate in queue order. Even though all computations were ready to run when queued, this may not remain the case for the entire duration that a computation remains in the queue. For example, if a user changes a parameter that is several steps upstream of a computation in the queue, the queued computation will need to wait for its inputs to be re-evaluated. For this reason, the up-wait counter is checked immediately prior to evaluating a computation. If it is non-zero, then the computation is ignored, and will have to be re-queued at a later time.

6.2.5 Rapid Results for Views

The computation queuing system is designed to run each computation only once, and then only after all prerequisite up-stream computations have been evaluated. This behavior eliminates the need to perform duplicate work, but also means that users must wait for all computations to finish before seeing any change in results. Waiting is not always desirable, as users want to know that the tool is actively responding to their request. To alleviate this wait time, Sonata will periodically pre-evaluate computations that are directly utilized by a view, provided that at least one of its up-stream computations has finished re-evaluation since the last redraw.

Pre-evaluating and redrawing views gives the impression that the tool is more responsive, but care must be taken as it actually slows the arrival of final results. The slowdown is due to the inability to schedule meaningful work while we are performing the pre-evaluation and redraw. In addition to the redundant processing, the browser must perform extra work to determine how the changes we made should be rendered. This extra work is divided into three categories: recomputing styles, positioning the layout (i.e. reflow), and repainting the screen. These operations usually take place in the main thread (delaying scheduling), and for a large change, the latter two can take upwards of 100 milliseconds. To reduce the impact of pre-evaluation, we impose a time delay on views after they have been redrawn, limiting the number of times that a view can be redrawn per second. We also perform redraws within animation frame callbacks, allowing us to cut down on the number of reflow/repaint cycles that occur. This affords us more time to evaluate models and squeeze in additional frames.

6.2.6 Optimizations for Branches

Users have the ability to compare and contrast potentially diverging outcomes via Sonata's timeline functionality. Whenever the timeline is branched, all parameters and computations get duplicated to permit outputs to diverge. If care is not taken, however, this can end up wasting a good deal of our limited resources as there is the chance that the computations will be performing identical work on two or more branches, as can be seen in Figure 6.4. This reveals a clear opportunity for optimization at the branch and computation level, improving response times in cases where branches are used.

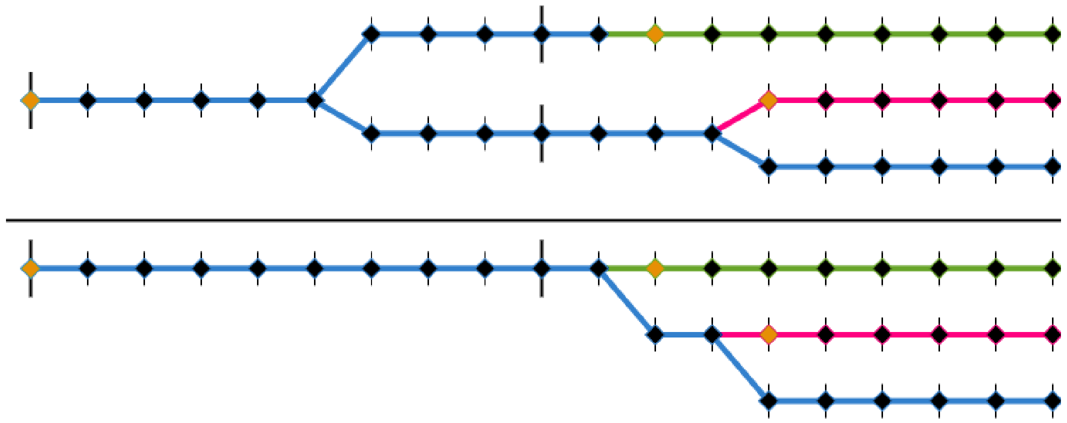


Figure 6.4: *Timeline branches (above) versus computational branches (below). User-defined branches can result in the same operation being performed on two or more branches (indicated by color). Computations are ideally evaluated once per unique set of inputs.*

One way we reduce the resource impact is by allowing timeline days to be shared by multiple branches. When a timeline branch is freshly split, days before the split point are shared with at least one other timeline branch, while days after the split only exist on the new branch. As a result, each timeline branch has references to the already exiting timeline days before the split point, and newly created timeline days after it. Each timeline day is assigned a unique id upon creation, generated from its original branch and the day number. In turn, any computations reflecting the events on a given day are keyed with that day's unique id, resulting in the creation and use of just one computation if the day happens to be shared. After the split point, however, the newly created timeline days with new ids result in the creation of duplicate computations not shared with any other branch.

Branch-level optimizations after the split point do not have the ability to fully eliminate redundant operations. Since different computations rely on different subsets of inputs, logically, only a subset of the computation outputs will actually diverge across branches when specific input parameters are adjusted. As such, we looked into the possibility of sharing specific duplicated computations that were guaranteed to be performing identical work due to current input configurations. A potential solution appeared in the form of splitting and merging the Computation Dependency Graph at optimal points. This was particularly promising as we could use the existing computation scheduling algorithm, shared dependency control, and value caching that already works well in a single branch scenario.

In the considered solution, the system maps input subscription sets to branches. Each unique set of subscriptions corresponds to a new computation instance for the given operation, causing the computational branching seen in Figure 6.4. For this purpose, uniqueness is determined by comparing the branch-specific values of all source parameters either required directly by a computation or required indirectly through a subscribed input computation. A computation instance is mapped to each branch that has identical values for all direct and indirect parameters and will subscribe to computations also mapped to the given branch. The subscription process works no matter the number of branches a computation is mapped to since all subscribed inputs require some subset of the indirect parameters and the values are therefore identical by definition; each subscribed computation instance is guaranteed to be mapped to all relevant branches.

Despite the promise of this approach, we decided to go in a separate direction. The models we use for predictions are tightly coupled, and the divergence of one model will cause the divergence of all other models in two or three days. Because of this, we do not significantly benefit from computation-level lineage optimization. What is much more useful to us is the short-circuiting of computations when the exact operation has already been performed, regardless of the branch. As such, we create duplicate computations (i.e. one per branch) and simply copy sharable cached results from other computations by reference instead of performing a full evaluation, as described in Section 6.1.3.

6.3 Outbreak Prediction Models

Sonata relies on predictions from machine learning models and values derived from those predictions to inform the visualization. In this section, we explore how models are evaluated and some of the complications that arose from their use.

6.3.1 Model Building Pipeline

Sonata allows users to interactively visualize the flow of disease outbreaks over time. In order to obtain results fast enough to support interactive use, we use predictions generated in real-time by machine learning models. The models used by Sonata are the final product of a multi-phase analytics pipeline [36]. This pipeline was initially designed for Cadence, but has been adapted and enhanced to support the requirements of Sonata. The framework is currently able to produce neural network [77], multiple linear regression [78], random forest [79], and gradient boosting [65, 66] models. Of the four types, we have found that an ensemble of decision trees [80] fitted using gradient boosting produce the most accurate results for our use case.

In phase I, parameter ranges are mined from expert-produced simulation scenarios. These ranges limit the space of explorable values to reasonable quantities probable in real-world outbreak scenarios. Using the ranges, the analytics framework samples the entire multi-dimensional parameter space using Latin Hypercube Sampling (LHS) [81], producing one million modified scenario files. This method of sampling ensures that the entire parameter space is covered by scenarios, including parameter values that are uncommon in real outbreaks.

Phases II and III of the pipeline distributes and runs the simulations across a cluster of machines, reducing the amount of time taken to run the 32 million simulation iterations. As simulation runs complete, the raw outputs (each several gigabytes in size) are encoded in our binary sparse replay format, recording what events occurred and when they occurred. From these replay files, we generate daily output counts that the models are later trained on, as well as Interaction Graphs that Sonata can use to reconstruct the outbreak's behavior visually.

Phase IV prepares the simulation data for the model training process. The first step of this phase is to compute the variance and normalize the input and output space. Following this, dimensionality is reduced by finding the set of inputs most correlated with each output using Pearson product-moment correlation coefficient (PPMCC). Finally, collinearity between inputs is analyzed, further reducing the input space used during model creation.

Phase V of the analytics process involves training the models using the prepared simulation data. The trained models are tested for accuracy using k-fold cross-validation [82]. The scikit-learn Python library [83] is used for many of the algorithms mentioned above. The training process produces a JSON file containing vectors and thresholds for the models, depending on the type. These models are then converted to a binary format, compressed, and base64 encoded. The resulting strings are included in the state JSON files, allowing Sonata to load and decode the models at runtime.

6.3.2 Model Evaluation

In order for Sonata to obtain a new value from one of the models, the model needs to be evaluated. This is a multistep process that takes place as part of a computation scheduled by Sonata's data manager. A model computation is created whenever any other computation or view subscribes to a specific model output from any given sector-day. Upon creation, the model computation subscribes to various input parameters or model outputs from previous days. The list of inputs a computation needs to subscribe to is included in the metadata for the model, which is packaged along with other sector data in each state's JSON file. The input list uses a relative notation for time, specifying inputs by name and a number of days in the past. The models are able to subscribe to the same type of output on different days, for example the number of latent units one and two days ago relative to the day the model computation is assigned. Subscribing to specific inputs ensures the computation (and therefore the model) only runs when at least one of its inputs changes and then only once all inputs are ready to be read.

When the model computation is scheduled on the main thread, perhaps due to a change in input values, Sonata starts by collecting the values of all subscribed inputs. These are collected into a single object keyed using the same notation used in the metadata input list (objects act as a dictionary in JavaScript). What happens next depends on the number of threads in use. If only the main thread is available, the values object is passed to the model instance, and the model is evaluated immediately. For our ensemble models, this process involves navigating several trees, comparing the keyed input values to trained thresholds along the way, shown in Figure 6.5 . If multiple threads exist, on the other hand, the computation looks up the worker that holds the given model instance and sends an evaluate message to it, copying the values object to the worker in the process. The model computation will then return undefined, a signal to the scheduler that is has not yet finished. This allows non-conflicting computations to be processed while the worker is busy, but will not mark downstream computations as ready to run. On the worker thread, when an evaluate message is received, the duplicated values object is passed to the model instance. The result is then copied back in a response message which includes the origin computation's id and freshness value. Provided that the inputs have not become stale since the evaluate message was submitted (checked by comparing freshness values), the scheduler will look up the model computation by id and pass the response message, completing the evaluation process. The model computation will process the response and return true (changed) or false (unchanged), allowing downstream computations to be scheduled appropriately.

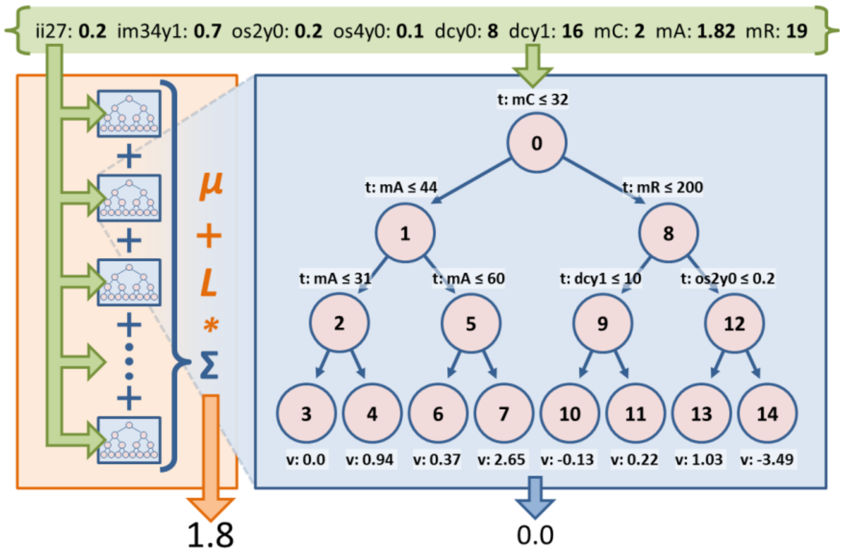


Figure 6.5: *The inner workings of a gradient boosting model. The depicted values are from the clinical model from Colorado sector-2, showing predictions for day 20.*

Due to resource limitations combined with the large number of models that need to be evaluated, we have invested some effort in improving the computational efficiency of our JavaScript model implementations. One area of improvement was making parameter access times as small as possible. Given that we are using JavaScript, the best way that we have found is to simply rely on standard object property accesses. We require two property accesses per parameter value used by a model, one to obtain the parameter value object, and another to obtain the specific numeric component. Early in development, parameter inheritance was resolved at evaluation time, but we quickly replaced this with other pre-computed reference-based approaches (see Section 4.2.4). In addition to these changes, we have made some modifications to help the browser produce optimized code more quickly, as modern browsers are capable of compiling commonly run JavaScript code into faster machine language at runtime. Code eligible for optimization needs to be predictable, meaning that care must be taken to not unduly change data types or object structures.

6.3.3 Bundling Model Evaluations

There is a great deal of communication and thread scheduling overhead when evaluating models using workers. This is particularly the case when a majority of models from the same sector-day are distributed across different worker threads, a common occurrence due to the way models feed into each other. We have alleviated some of this overhead by bundling all model evaluations for a given sector-day into a single computation, leading to some interesting tradeoffs.

With this approach, each bundle computation subscribes to the complete set of inputs used by the constituent models. Since the computation will be scheduled any time at least one input changes, we can no longer rely on the computation itself to determine which models are stale. As such, each input is checked for freshness as the values for all models are collected into the values object. Each input is assigned a binary value where each bit position corresponds to a different model, allowing fast bitwise operations to be used to determine if a model is stale without needing to search through model input lists. Once completed, the old result and input values object is copied to the worker as part of an evaluate message. On the worker side, each model is reevaluated if stale and a response message is returned.

The requirement to send previous results in the bundled setting increases the message size, but only one message is sent and there is only one potential thread scheduling delay per sector-day. This is compared with nine smaller messages distributed to various threads when evaluating models independently. Additionally, there is lower parallelism with the bundled approach, as only one or two threads will be in use (at least during a light outbreak). As a side effect, however, there is no time spent waiting for results from other models associated with the same sector-day, such as when generating map visualization data. In fact, we included map data generation within the bundle, since it has to be recomputed anytime any of the models change. This has the benefit of getting it off the main thread, but the data is now always generated instead of only when needed by a view. So far, bundling the models in Sonata does not appear to have had a significant impact on performance, but there is still room for adjusting the approach and the potential for improvement.

6.3.4 Tuning Predictions: Predicting for the Future

Our first models for Sonata were trained to predict values several days in the future. Specifically, each model was fed inputs from day d and in turn predicted the output value that should exist on day d+5. As described previously, these inputs included scenario parameters along with outputs from models evaluated on preceding days. When it came time to add prototype plots to Sonata, shown below in Figure 6.6, they actually revealed an interesting problem with the way we were training the models. We assumed that all output values would always be available to Sonata for all days for which the model was to be evaluated. When used in Sonata, however, we realized that the first five days of values would be missing, since there was nothing to generate them. As a side effect of the missing values, the first five days of predictions all had the same output values. When these constant model outputs are then used for model evaluations over the following five days, a step pattern emerges. The use of missing values (zeros) for five consecutive days results in the same prediction for the following five days, which in turn results in the same prediction for the five days after that. The process repeats for the remaining five day intervals.

As a solution to this issue, we initially considered packaging the first five days of values within the model metadata so that Sonata could initialize the inputs when the tool starts. With this approach, the model generation tools would need to be adjusted slightly so that this information could be included from the training stage. We also figured that this would be a good opportunity to include other metadata in the models, such as a model-specific number of days in the future that each model is trained to predict (as opposed to just five), or the average number of days that the outbreak is expected to last. Before we had the chance to implement this approach, however, we found a technique that resolved the problem entirely. Our chosen solution was for the models to always predict one day in the future, and use inputs from a variable number of days in the past. The selection of inputs as well as the number of days in the past for each input is determined during the analytics and training process. The use of inputs across several days helps to capture more complex behavior, and also prevents repeating predictions over future days.

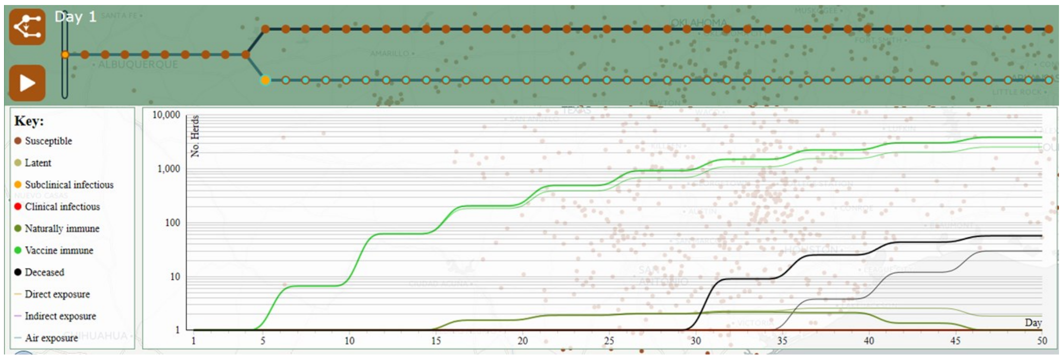


Figure 6.6: *A plot showing diverging intermediate predictions from two sets of input parameters. The predictions step every five days, likely due to a lack of initial conditions.*

6.3.5 Tuning Predictions: Percentage Based Models

Over the course of Sonata's development, we experimented with a couple of model prediction approaches in an attempt to make our predictions more stable when using multiple models. Our original approach shows high accuracy for individual models operating on their own, but the models do not always agree when working together. Our original models independently predict the number of latent, subclinical, clinical, naturally immune, vaccinated, and depopulated units, as well as infection rates on each visualized day. Each of these models is optimized to solve its own problem and therefore has strengths and weaknesses in different areas of the input space. As a consequence of this, we sometimes see a timing or behavior mismatch between different models given the same input sets.

To remedy these inconsistencies, we devised a new mechanism for model predictions. Instead of predicting how many units there are in each health state, we tried predicting what happens to units already in a given health state. For example, given current conditions a model might predict that 60% of latent units stay latent and 30% become subclinical. Sonata will then track the progress of individual units from day to day, using these percentage predictions to inform changes to a unit's health state. It is also worth noting that since the models predict the advancement of individual units, the resulting data can be visualized right away without further processing, unlike what is required with the original model approach. Using this method, the number of units active in an outbreak is always properly maintained over time, ensuring that the output counts agree across all outputs. As the model outputs are used to advance the health state in accordance with the cyclic graph defined in NAADSM and ADSM, we run the models in that same order back to back in a bundled computation.

Unfortunately, as the newly trained models became available, it turned out that they had unacceptably low accuracy. There could have been multiple reasons for this, including the fact that the new models needed to predict nearly 10 times as many distinct values as our original models. Furthermore, the models predicted value tuples, so errors may have been introduced in the conversion from predicted value to tuple. Due to the low accuracy, we ended up abandoning this technique in favor of other corrections.

6.3.6 Tuning Predictions: Simplifying Production Types

The original design goals for Sonata involved the tool being able to predict outbreak conditions for each of the various production types. The number of production types varies by simulation scenario. For example, in the Great Plains scenario, the production types are small and large cow-calf, small and large dairy, small and large swine, small and large feedlot, and small ruminants. There are additional cattle and feedlot types in some of the other scenarios. Unfortunately, there are a few issues that arise when working with so many production types, especially when the type variants differ between scenarios. Of primary concern is the amount of extra processing that needs to be performed when evaluating production-type-specific outputs. For each production type that we predict for, we must run the models an additional time. In the Great Plains scenario, this results in nine times the CPU utilization of our current all-type predictions. Furthermore, the number of type-specific parameters included in the models drops per production type as more production types are included. This is a result of only including the most significant parameters overall in the model. As a final point of concern, the differences in production types in different sectors add a layer of complexity when setting parameters simultaneously across multiple states.

As a solution, we have been actively working on reducing the number of production types used by Sonata down to five: 1) cow-calf, 2) dairy, 3) swine, 4) feedlot, and 5) small ruminant. The first technique that we attempted was merging production type variants within the simulation scenario files. This results in simulation outputs that only include the desired five production types, outputs that are then used to train our models. Unfortunately, the merge process seems to have drastically changed the simulation behavior, resulting in poor accuracy compared to the original in our initial tests.

We want the simulations to be as accurate as possible, so it may be best to include the full set of production types in the scenarios, including small and large variants. In such a case, we can bridge the gap when we produce the scenario variants that explore the parameter input space. Currently, we adjust the values of small and large production types independently. Instead, we can adjust the parameter values for an associated pseudo-type; the values written to the scenario variant file for the small and large version of each production type will be separate, but both tied in a well-defined way to the value of the associated pseudo-type. At training time, the models will be provided pseudo-type values as training inputs (both parameters and health-state counts), and will therefore predict the outcome of simulations run using the bound values for the associated small and large type variants. Within Sonata's parameters menu, planners will be able to adjust parameters specific to the pseudo-types. At the same time, we can make use of color to simultaneously display the true shape of parameters for large, small, and other variants associated with a given pseudo-type, even though only the pseudo-type value is used with the models. This work is still ongoing.

6.3.7 Performance Evaluation of Models

Below, we present some timing experiments we performed on Sonata using the non-bundled evaluation approach. Results are collected for two browsers, Firefox 59 and Google Chrome 66 running on Windows 7 (i7-3630QM @ 2.4GHz, 8GB RAM, SSD). Chrome has better performance when running Sonata.

We start off by measuring various components of load time, shown in Table 6.1. In our tests, we are loading 12 states: Colorado, Kansas, Nebraska, South Dakota, Wyoming, Iowa, Missouri, Oklahoma, New Mexico, Texas, Arkansas, and Louisiana. In total, Sonata running in Chrome spent 5.2 seconds waiting for all files to be loaded and made available to the main thread and Firefox spent 5.6 seconds waiting. In reality, 18.8 seconds were spent loading files in Chrome (1.5 seconds per file on average) and 31.7 seconds in Firefox (2.6 seconds per file on average). This time includes receiving the file, decompressing it, parsing JSON to objects, serialization for transit between threads, and initialization. Most of this time is hidden from users because it is done in parallel across many WebWorker threads. Some of the initial loading time is spent drawing incomplete results so that users know that progress is being made even when that progress is relatively slow due to load, disabling this feature reduces loading times in Firefox by 3 seconds.

Table 6.1: *Loading times in two browsers. Files are loaded in parallel, resulting in a lower perceived loading time.*

Event	Chrome 66	Firefox 59
Loading time per file (average)	1571 ms	2648 ms
Model initialization (per model)	107 ms	71 ms
Parameter initialization (per state)	2 ms	6 ms
Total time to load	5.2 s	5.6 s
Total time to evaluate models	2.4 s	7.4 s

The main body of the test involved timing how long it took to finish processing all models after adjusting parameters over an increasing number of branches. Each test captured this duration after setting some parameters and then repeated the process after resetting the values to default. The number of branches increased from one to four, and for each a small parameter change was made followed by a large parameter change. The small change targeted vaccination in Nebraska on day 11 of the top branch, although the change can potentially cascade into other states. The large change adjusted the depopulation capacity and probability of observing clinical signs in feedlots on day 11, and also adjusted movement control for indirect infections from dairy to feedlots on day 22. The large change, shown in Figure 6.7, was designed to trigger the recalculation of a majority of models. The timer begins once parameter changes are made, and ends once all computations have finished updating. To end the timer, we make use of a computation that subscribes to all leaves in the Computation Dependency Graph, guaranteeing that it will be run last. Ten chart views and one map view are opened per branch to consume and draw the results of the model updates. A two second cooldown period exists between measurements to allow internal browser processes to stabilize.

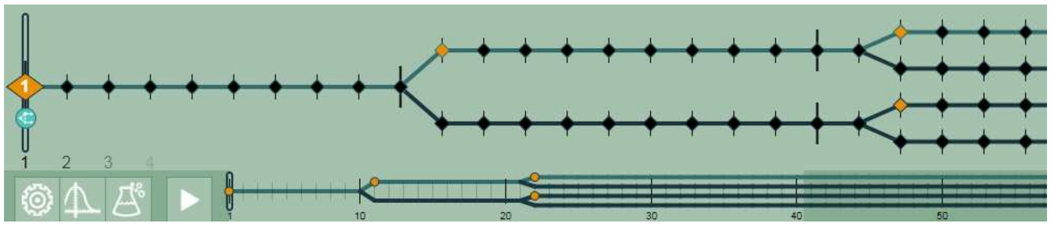


Figure 6.7: *The branch setup for a large parameter change during our multi-branch timing experiments.*

Results from this experiment are shown in Table 6.2 and Table 6.3. In addition to increasing the number of branches, we also make measurements for three computation configurations: 1) Each model has its own computation; 2) Each model has its own computation, but a barrier is inserted at the end of each day; 3) Each sector has a single computation evaluating sector models back-to-back, also with a barrier. Chrome finishes processing in a reasonable time even as we increase the number of branches. It is important to note that the measured times also include processing delays introduced by pre-evaluating the models for early view updates.

Table 6.2: Model evaluation times in Chrome using various computation configurations with an increasing number of branches. The experiment is run with small and large impact parameter changes. The barrier waits for all daily updates before proceeding.

Branches	**Individual Models**		**Individual + Barrier**		**Bundled + Barrier**	
	Small	**Large**	**Small**	**Large**	**Small**	**Large**
1	0.3 s	1.1 s	2.5 s	2.5 s	2.4 s	2.4 s
2	0.7 s	2.4 s	3.5 s	7.5 s	2.9 s	5.7 s
3	0.6 s	4.6 s	9.6 s	16.7 s	8.4 s	13.4 s
4	0.9 s	5.2 s	13.3 s	20.6 s	10.6 s	16.8 s

Table 6.3: *The frequency and duration of various events during the timing experiment.*

Event	Freq. (Chrome)	Chrome 66	Firefox 59
Main computation loop	3,687	4 ms	<1 ms
Worker model evaluate	177,796	<1 ms	<1 ms
Map data generation	50,944	<1 ms	<1 ms
Map update/redraw	24	3 ms	3 ms
Chart update/redraw	1,460	1 ms	2 ms

6.4 Paarlberg Economic Model

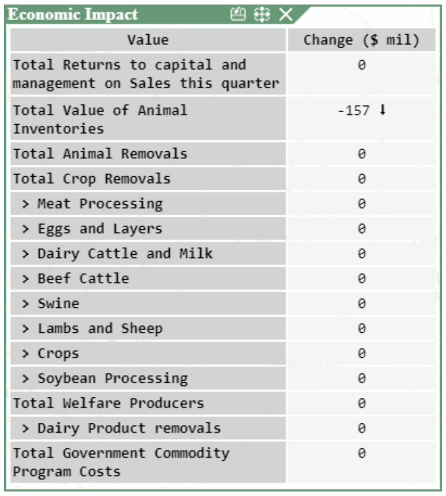
Some of the results that we would like to display in Sonata come from external software. Since it is not always possible to adapt existing software for use in a browser, we have added support for querying external scripts from a data computation. We currently utilize this functionality to call the Paarlberg Partial Equilibrium Model, a model that predicts the economic consequences of outbreak control activities across several commodities. Although currently only used for the Paarlberg model, it would be possible to extend this functionality to other services, or potentially offload less essential model evaluations.

6.4.1 Querying External Scripts

Sonata makes use of its existing computation framework to submit queries to external services. Having the economic model queried from a computation has a number of benefits, including automatic updates whenever its inputs change and the potential for results sharing between branches. Since network communication overhead combined with uncertain server-side processing times can delay results significantly, we require a mechanism that allows other non-conflicting computations to run while we wait for a response. To perform the query without delaying other work, we make use of the same functionality that allows computations to wait for results from worker threads.

When the computation runs, Sonata opens a new connection to the server in question and sends any required input data. In the case of the Paarlberg model, this data includes the number of depopulated animals from each production type. Since we predict depopulation counts in terms of herd or farm production units, we make use of statistics collected for each sector to compute the expected number of animals. After sending the input data to the server, the computation returns undefined, an indication to the scheduler that results are still pending. In this case, the computation's waiting status will remain set, preventing downstream computations from being scheduled while still allowing non-conflicting work to continue.

The computation also registers a response callback when opening the server connection. When results arrive, the callback packages the results into a message similar to those returned by workers. This allows the existing worker-response code to properly deliver results to the requesting computation. Should the arrival of results be successful, then any subscribing downstream computations can access the results like they would with any other type of computation, allowing the production of visuals seen in Figure 6.8. If, however, the original inputs happen to change again before results are received, then Sonata is able to discard the expired results before they are made available to subscribing computations. This is possible as each request-result pair has a freshness value associated with it, the same as worker results. Prior to the arrival of stale results, Sonata will make an attempt to cancel the previous query by closing the existing server connection, an action which typically results in a hang-up signal being sent to the server-side script.


Economic Impact

Value	Change ($ mil)
Total Returns to capital and management on Sales this quarter	0
Total Value of Animal Inventories	-157 ⬇
Total Animal Removals	0
Total Crop Removals	0
> Meat Processing	0
> Eggs and Layers	0
> Dairy Cattle and Milk	0
> Beef Cattle	0
> Swine	0
> Lambs and Sheep	0
> Crops	0
> Soybean Processing	0
Total Welfare Producers	0
> Dairy Product removals	0
Total Government Commodity Program Costs	0

Figure 6.8: *Predictions from the economic model based on the losses suffered during an outbreak predicted by Sonata.*

7 Interaction Graph

Providing real-time access to visual and data analytics using data produced in long-running stochastic simulations has been a major focus of this work. To accomplish this, we have utilized machine learning models trained on datasets encompassing outputs from over 1,000,000 ADSM and NAADSM simulation runs. These models enable us to predict what the simulation would have produced given a set of scenario parameters with reasonably high accuracy, at least for simple numeric values like the expected duration of an outbreak.

When we considered obtaining changes in individual herd health states over time from these models, we realized that they would not be able to satisfy that request with the level of accuracy that we desired. While the information required to reproduce state changes is available from the output of individual simulation runs, it would have been difficult to train a lightweight model that captured the infection behavior of the original simulation. Due to the stochastic nature of the simulation combined with the sheer number of herds and simulation days (over 360,000 herds in a region and over 50 simulated days), any such model would have extremely high dimensionality, size, and complexity and would therefore be unsuited for use in Sonata. It was clear that we needed another technique to capture herd-level outbreak behavior. In response, we opted for a graph that was capable of capturing interactions between herds.

7.1 Graph Structure

Our Interaction Graph attempts to make suitably accurate predictions using a format small enough to be downloaded within a fraction of a second by Sonata. The Interaction Graph is a directed graph that captures the probability that one herd will infect another. We have adjusted the format and building methods multiple times, changing our approach to meet the emerging demands of the Sonata tool.

7.1.1 Current Graph Format

In the Interaction Graph, herds are represented by vertices and the probabilities of specific interactions between herds are represented by edges. On the edges are event tags that include information about which events occurred, when they occurred, and how often they occurred. We are primarily concerned with capturing the infection and exposure events as determined by the simulation. As such, the structure of the graph enables the preservation of infection spread locality, solving a crucial visual accuracy concern.

We make use of the Interaction Graph when visualizing an outbreak, so we include metadata about the herds within the corresponding vertices. Some basic information that we store includes associated ids, latitude, longitude, number of animals on the premises, and type of animals on the premises. Furthermore, we include whether or not the herd is classified as a super-spreader [84], enabling Sonata to make more intelligent decisions when an outbreak is active in the area. This classification is determined through analysis of the original simulation outputs, comparing the frequency of infections originating from the given herd compared to all other herds and the location of hubs in the transmission network.

Probabilities are captured through the use of event counts. We can determine the relative probability of an event occurring between a source herd and destination herd by looking up the source vertex and comparing the counts along its edges. For this operation, we divide the event count along the destination edge by the sum of relevant event counts along all of the sources edges. More complex querying is also possible; this is described in Section 7.3. In addition to infection events, we also capture event counts that are only relevant to one herd, which we store alongside the other metadata. An example of this is the probability that a herd will change from one health state to another after a given number of days, as this can be useful when querying the graph.

Figure 7.1 depicts a simplified view of our initial interaction graph structure. We also have a variation of this format where the day information is not included, and the events are aggregated accordingly. We primarily use this variant to reduce the size of the graph when timing information is not required.

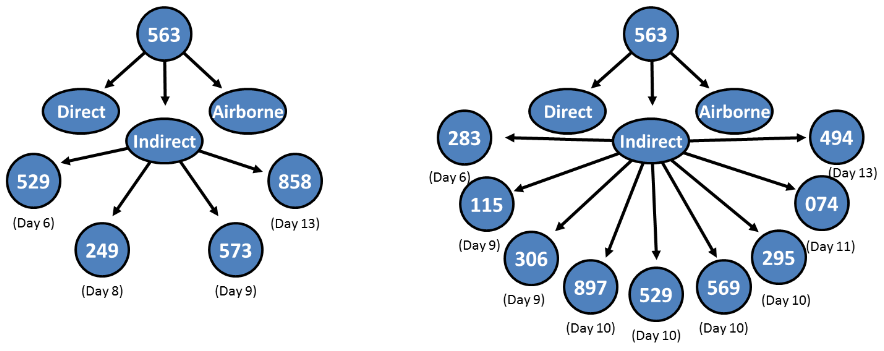


Figure 7.1: *Interaction Graphs depicting infection events from one source herd during two separate simulation runs. Numbers within the nodes are herd ids, and tags on edges represent type, day, and event count. Counts from all simulation runs are eventually summed into a single graph.*

7.1.2 Replay Format

We now convert and archive simulation outputs within binary replay files. The format, which was inspired by early versions of our Interaction Graph, records relevant daily events that occurred within a simulation run, allowing Sonata to visualize the events in the correct order. These replays feature a sparse format that captures events relevant to Sonata in one thousandth of the space of the original outputs, and can be read in under a second. The replay files include all of the events needed to build an Interaction Graph, and we now use them exclusively when constructing the graphs and extracting model training data.

7.1.3 Reducing Graph Size

As the graph is primarily meant to be used within Sonata for informing the visualization, maintaining a small graph size is critical for reducing load times when using Sonata. As such, we have taken steps to remove less relevant information from the graphs used directly by Sonata. Table 7.1 details the growth in file size as the graph is built, as well as average shortest path length and average clustering coefficient.

In an effort to reduce the graph size, we have removed extra events that were relevant to queries, but not relevant to the visualization. First off, we no longer include any state change events, and we only include aggregated counts for other event types, as opposed to daily counts. We also no longer include direct farm-to-farm shipments in the graph, instead only including airborne and indirect infections and exposures produced by the simulation. This change is largely due to differences between the simulation and our county-to-county shipments data, which happens to be of too low resolution to associate infection probabilities with individual herds.

Table 7.1: *Shows the growth in file size for Interaction Graphs covering the Great Plains scenario. The number of vertices and the number of edges per infection type is included. The file is in a JSON format.*

Replays Included	File Size (GB)	Number of Nodes	Average Degree	Average Path Length	Average Clustering
112,900	1.03	95,683	546.25	2.30	0.84
225,708	1.40	95,701	737.03	2.16	0.85
338,508	1.66	95,701	873.22	2.12	0.85
451,308	1.88	95,701	983.17	2.08	0.85
521,054	2.00	95,701	1043.57	2.06	0.85
559,444	2.06	95,701	1073.52	2.06	0.83
594,444	2.11	95,701	1099.46	2.06	0.85
629,025	2.16	95,701	1124.24	2.05	0.84
659,955	2.20	95,701	1145.43	2.04	0.84
684,658	2.23	95,701	1161.75	2.04	0.85

In addition to these changes, it has become apparent to us over the course of developing and using Sonata that we do not need to provide complete knowledge of every possible infection destination in order to provide compelling and accurate visuals. As such, we now set a limit of eight destination edges per source herd vertex. The top eight destinations are determined after incorporating all event counts, allowing the destinations with the highest probability of spread to be selected. This limit does not apply to herds classified as a super-spreader, allowing major infection hubs to remain intact. Because there are so many herds that could potentially be a source and relatively few herds that get infected each day, it is unlikely that a user will notice that the number of destinations is capped.

7.1.4 Graph Size Tradeoffs

We made a few tradeoffs between size and accuracy with our choice of graph structure. As a side effect of collecting events across all of the various simulation runs, the Interaction Graph only captures the average case outbreak. This means that the graph does not directly respect any scenario parameters set within Sonata that can influence how often and how far the disease spreads and how quickly herds change state. We could incorporate parameter range information into the graph through the use of decision trees in order to provide better tuned probabilities, but doing so would greatly increase the size of our graph and our download times in turn.

Instead, we have opted to use the graph for base probabilities and adjust these probabilities with values obtained from other parts of Sonata, primarily predictions from other models. For instance, we predict the number of herds in each of the seven health states using machine learning models and then update individual herds to match the prediction. We incorporate the graph's base probabilities to determine which herds are most likely to need updating. Combining these prediction methods allows us to increase the accuracy of predictions obtained from relatively small models.

7.2 Graph Production

Our approach for building the Interaction Graph has evolved over time as Sonata's requirements came more clearly into view. In this section, we discuss these approaches and explore the reasons why they were improved. We also explore some of the ways the code for building the Interaction Graph was adapted for other purposes.

7.2.1 MapReduce Production Approach

Building the graph is a computationally expensive process. When using a single machine, this process takes several days, emerging from the fact that the simulation outputs are cumulatively in excess of 100TB, for a single geographic region. The slow rate at which the data can be read from the disk and decompressed in memory presents a bottleneck; to cope, we decided to distribute the process of graph generation.

Our first attempt at distributed graph generation involved MapReduce. We chose to use MapReduce since we were processing the raw output files line by line, a pattern MapReduce handles very well. Our mapper takes a line from the simulation output and extracts the herd to herd infection events from it. The events are split and each one is emitted to a single reducer, with an intermediate combiner counting duplicate herd to herd events before transit. At the reducer, the events are collected and counted, keeping direct, indirect, and airborne infections separate. Probabilities are generated from these event counts and the source herds are matched to build a directed graph. We distributed the output files evenly across our cluster in order to take full advantage of the algorithm.

7.2.2 Local versus Global Query Approach

After work on the MapReduce based solution, we experimented with adding advanced query support to our Interaction Graph (described in Section 7.3). Although MapReduce worked well for our original goals, it did not afford us the flexibility we desired when it came to the new query features we wanted to include, so we needed to revise the build process. The second process for building an interaction graph first involves clustering the simulation variants by feature space, and then subsequently distributing the simulation output files to different machines assigned to each given cluster. On each machine, we decompress the simulation output files in memory and process them line by line in order to produce a Local Interaction Graph. This local graph captures the desired interactions for all variant outputs stored on that machine, and each is similar in functionally to the single graph we were previously producing with MapReduce. Due to the decompression and line processing being mostly CPU bound, we are able to process eight output files concurrently per machine with negligible disk contention, loading a new file from the disk every half second on average. We were also able to feed infection events to a single graph update thread through the use of concurrent queues, allowing us to avoid synchronizing the readers and slowing the graph update process.

Each processing node has its own local graph that captures the files on that machine, but we also wanted each node to hold a Global Interaction Graph that captures all of the local graphs as it allows us to have a view of the average state of the system. We accomplish this by passing parts of the local graphs to other machines, and then combining those parts with the global graph when they are received. Specifically, while each node is building its local graph, it also builds a smaller update graph that will be sent to other nodes. If we organize our nodes in a ring, one machine that we will designate as the leader can initiate graph synchronization rounds. During a synchronization round, the leader sends its update graph around the ring. At each hop, the node combines its own update graph into the message and forwards it to the next node. This involves unmarshalling the update graph from the previous node, cumulatively combining its own local updates with the update graph, and then remarshalling the modified update graph. One convenient feature of the Interaction Graph is that it stores probabilities as event counts, so combining two graphs is as simple as summing the counts on matching edges. Due to the reuse of edges, there is a limit to how large the graphs will grow when combined. In our testing, they asymptotically approached 15MB for the Texas region with 360,000 herds. Once the update containing all of the other nodes' updates is received back at the leader, it is once again passed around the ring. This time, each node that received the message combines the completed update with its global graph, allowing the nodes to commit the update to their global graph without any double counting. This update method allows us to cut down on the number of transfers in the system, just 2N where N is the number of nodes.

This process appears to work best when the local updates are small in size, i.e. when recent updates only involve a small subset of the herd space. Unfortunately, due to the stochastic nature of the simulations and the exploration of different areas of the input space, it is common for the updates to become quite large, often several megabytes in size. Larger updates pose problems on many fronts, all resulting in global graph synchronization times upwards of 30 seconds per node. First of all, large updates require more time to transfer over the network. Surprisingly, this is the smallest component of the update process, taking only 1 or 2 seconds to send a large, nearly complete local graph and significantly less time if it is smaller. The unmarshalling process contributes a larger chunk of time. The binary message format must be converted into a graph format so that it can be combined later in the process. This means that memory must be allocated for each of the small objects internal to the graph and their state initialized. We use Java to build the graph, and that means that we may need to spend additional time running the garbage collector at this time to free needed memory. During the following combination process, every attribute on every edge of the update graph must be matched and their associated occurrence counts summed. This requires traversing every node in the graph to obtain its edges. The final step, marshalling, takes longer than the unmashalling step due to the addition of new herds, edges, and attributes. We made some optimizations to our internal graph structure to help reduce synchronization times. Some of these optimizations included reusing key strings wherever possible, adding indexes for quick herd and attribute access, and using object with mutable values. These made a noticeable impact, shaving off several seconds from each node.

7.2.3 Current Production Approach

We have reduced our reliance on the Interaction Graph due to complications that arose with our long-range data sources. Our original versions of the graph relied on herd-level long-range shipment events obtained directly from simulation outputs. While working on connecting local models in order to predict behavior at a national scale, we needed a separate data source that extended past the regional boundaries of the simulations. The dataset of choice was a county-to-county dataset constructed by enhancing real-world cattle shipment data obtained from Certificates of Veterinary Inspection (CVIs) with a probabilistically generated mesh to fill in gaps in the CVI data [71, 72]. Unfortunately for our use case, this dataset has a low resolution at the county-level in order to protect the privacy of the farm owners listed on the original CVIs. Combined with the fact that the simulation herd population we use in our graph is randomly offset (also for privacy protection), we are unable to exactly match the county-to-county data to the herds used in our

Interaction Graph. We would need to probabilistically generate our own shipment mesh, likely at the cost of accuracy.

Altogether, both of the data sources that we could use for our Interaction Graph were generated probabilistically. With this in mind, it became clear that we could simply reproduce these distributions directly within Sonata at little cost instead of using the same distributions to fill out the Interaction Graph ahead of time. Even the short-range infections can be generated at runtime by Sonata, since the target destinations are determined using simple probability density functions within ADSM. Despite this, the graph still helps us produce the visualization for short-ranged phenomena, as it is good at capturing base probabilities that are influenced by farm size and farm production type (e.g. cattle versus swine). It is important to note that the graph is not limited to short-range phenomena, and could still be of use in other application domains. The issues described here arise from a lack of accessible high-resolution real-world data for our particular use case.

The production of our reduced Interaction Graph is fairly similar to our previous query-focused approach. One notable difference is that we no longer read raw simulation outputs when constructing the graph, we rely on pre-processed simulation replays instead. Furthermore, instead of partitioning the replay files by input-space, we simply distribute them equally across machines. During graph construction, each machine reads the assigned replays and counts relevant events in order to build its own full-resolution, but incomplete version of the Interaction Graph. When all machines are finished, the partial graphs are collected by a single machine and merged into a single Interaction Graph covering the entire input space. File size reduction measures are also applied at this time, such as limiting the number of destinations per herd.

7.3 Graph Queries

We looked at whether the Interaction Graph was a good fit for answering more advanced queries about an outbreak. Some of the queries we can answer are "If herd H is infected, which neighboring herd is likely to be infected next?", "If herd H is infected, what is the expected disease duration?", and "If herd H is infected, how many days until it results in the infection of 50% of its neighbors”. Variants of these queries are also allowed, specifying additional herds, start days, and spread methods. One of the most interesting results we can obtain from the queries is obtained when we run a query on both the local and global graphs. When we do this, differences in the results reveal how the feature space reflected in the local graph influences outbreaks compared to the global average.

Using our graph structure for querying had mixed results. The simple queries that are useful for informing Sonata visuals are able to be answered in milliseconds. Unfortunately, due to the structure of the graph only including one-hop spread probabilities, using the graph to compute more complex queries requires more computation time as multi-hop probabilities need to be computed by traversing the graph. Some herds have the potential to infect half of the herd population; in this worst-case scenario, queries can take between five seconds to five minutes. The time taken rules out the ability to interactively query the graphs within Sonata, at least when using this method. An alternative approach is to use the information captured within the graph to perform lightweight stochastic mini-simulations. This method should produce results much faster since only a subset of the graph is needed, but it is likely to produce less accurate results. The ability to directly query the graph does present some potential outside of Sonata, although care must be taken as the results reflect the average of all outbreaks.

8 Collaboration in Sonata

Sonata is designed to be a tool that enables the spatiotemporal visualization of disease outcomes at the national scale as a planning exercise. In a real-world scenario, the planning process is a collaborative effort involving multiple participants from a range of departments and with specific areas of expertise and jurisdiction. We want to enable all participants to be involved in the Sonata planning exercise so that they are able to contribute according to their expertise. This will make Sonata a collaborative planning tool.

We will need to ensure that other users will be able to interact and improve upon each other's changes, both in a cooperative and competitive manner. This includes cases where changes are made remotely and the client is not actively viewing the modified region. There are multiple facets to this problem, including new requirements for both interface and backend components. We will explore these new requirements and our solutions throughout the remainder of this chapter.

8.1 Synchronizing Multiple Clients

We envision a team planning exercise as several planners working together within a room, each with their own tablet computer. As the planners will be working together to contain the same outbreak, they should be able to see the influence of their peers' actions on their own system in real-time. This requirement naturally brings the need for a networking component into focus, but introducing a naive design could come at the detriment of the rest of the system. In this section, we detail our approach for synchronizing multiple clients while ensuring that the participating clients remain responsive.

8.1.1 Connecting to a Collaborative Session

Users are able to join a collaborative session by using the controls available in the options menu. Using this interface, participants provide both a session id and a user id. The session id is used to determine which collaborative session to join; supplying an id that does not yet exist will create a new session. The user id is displayed to other users in a number of places. The first of these is on the collaboration interface, where a list of all users connected to the current session is displayed. Furthermore, when a user performs a shared action such as changing a model parameter, their user id is recorded on the collaboration server alongside information about the change. The name and action is then relayed to the other users, enabling everyone to be notified about what action was taken, and importantly, who made the change.

Coming from a distributed systems background, it is easy to assume that we can just plug in our favorite publish/subscribe framework and call the problem finished, but our decision to build Sonata using web technologies within a browser limits our choices. To simplify the problem of communication, we send messages back and forth between clients and an intermediate collaboration server, which is capable of forwarding messages to interested clients. We use AJAX long-polling for return communication, in which the browser repeatedly opens an HTTP connection to the server with a long timeout so that the server may reply with any new messages when they become available.

8.1.2 Submitting Update Events

Before any synchronization can take place, participating clients must send information about the state of any sharable objects. Currently only timeline branch configuration, set parameters, and view configurations are sharable in Sonata, but this area of the tool is still in active development. Shareable data is transferred to the collaboration server as a JSON list of key-value tuples. Each key uniquely identifies a branch, parameter, or other sharable object, and closely resembles the format used for keying data computations. The value does not have a generic structure and can be any valid JSON value. Each sharable object is responsible for reading and writing their given value storage format.

How much data is transferred to the collaboration server depends on what triggered the transfer to take place. In the most common case, update data will be sent to the server anytime a shared object is changed by the user. We want to ensure that participation in a collaborative session does not degrade the responsiveness of the tool. As such, we prioritize the visualization of changes made locally, meaning that any user-updated values will immediately be reprocessed by subscribed data computations, potentially finishing before other clients receive the update. Besides single key-value update messages, Sonata will also send values in bulk in certain circumstances. This happens when a user decides to share an object (primarily a timeline branch) that was not previously shared, or when they join a collaborative session and they have already designated objects as shared. The reason we give users the ability to decide which shareable objects are actually shared is twofold: 1) we can support more users if not everyone is submitting their own branches, and 2) it allows users to have private branches for experimentation. When (re)joining a session, objects already designated as shared might not be synchronized with other clients. For this reason, Sonata waits until after the initial update request completes before sending differences to the server.

8.1.3 Receiving Update Events

In order to receive collaboration updates submitted by others, clients must register their interest with the collaboration server. This is accomplished by opening a connection to the server and specifying the range of updates desired. The HTTP connections are opened with a long timeout in order to allow the server to respond in the distant future, a technique called AJAX long-polling. If the connection is closed before results are received, the client will repeat the request.

Upon first joining a collaborative session, a client will request all past events and values from the server. This allows them to become completely synchronized with all other clients, regardless of the time that they joined the session. Like update submissions to the server, results arrive in the form of an ordered list of tuples in JSON format. Instead of just key-values pairs, however, extra information about the user that submitted the update and the time it arrived at the server is also included. When updates are received, the client processes them in order to ensure that shared objects exist before they are used, such as timeline branches being created before values are set on their days. Once all update entries have been processed, the client initiates a new connection with the collaboration server, this time requesting any entries not already received.

During processing, each update entry key is deciphered to determine the action that needs to be performed. Some actions involve the creation, update, or removal of timeline branches and others involve looking up sector and day objects to perform a parameter value update. In addition to updating outbreak related values, other actions include updating active user lists, displaying chat messages sent from other clients, and "sharing screens" by synchronizing the current view configuration. Most actions will result in a notification message being displayed in the lower right-hand corner of the screen specifying what was changed and who changed it. The notification messages stack vertically and automatically fade after six seconds (by default), but the software can also issue messages that must be clicked to be dismissed.

One concern we had while planning the collaboration component was that excessive updates could have a negative impact on the responsiveness of other clients. This turned out to not be the case for a couple of reasons. First of all, parameter values are updated using the same API that is used by various interface components, so the same logic applies to determine if computations should be evaluated. Sonata will only re-evaluate computations if there is at least one open view that makes use of a given value. This means that there will not be a performance impact for any updates that the user is not interested in visualizing. Secondly, all expensive operations like model evaluations take place on WebWorker threads. As a result, collaborative updates are performed in the background without impacting the responsiveness of the user interface. Users may experience a sub-second delay in the completion of visual results if they make changes while a collaborative update is being processed, but only if the changes are on unrelated timeline branches. Changes a local user makes will interrupt and restart affected computations, resulting in the user experiencing the same time to visual results that they would have outside a collaborative session.

8.1.4 The Collaboration Server

The collaboration server is responsible for accepting update events from clients and determining who should receive them. Since connecting clients may join late into a collaborative session, the server stores all events in a database. We use SQLite 3 for this purpose, and each collaborative session is assigned its own database file determined by hashing the session id. The database contains a table for events and a table for active users. When a user connects to the collaboration server, either to submit events or to request them, the server will update the given user's activity timestamp.

Any time the server receives update events from a client, they are stored within the database before being forwarded to other clients. To ensure that all clients see the same results, each event entry is assigned an incrementing id when it is inserted into the database. Database ACID guarantees ensure that the order is valid even if multiple clients are inserting events at the same time. The entries are indexed by their assigned id to enable fast lookup and retrieval later. In addition to providing each entry with a unique id, the event arrival time and submitting user id are also stored, although this is purely informational and not used for ordering.

Clients request events by connecting to the collaboration server and specifying an event id. When the server receives a request for update events, it will scan the database for any events with an id greater than the specified id. Because of this, newly joining clients can specify an id of zero and receive all available events. If any events happen to exist with a greater id, the server begins constructing a JSON response. With the information from the database, the server creates event entries in the form of (key, value, user, time) tuples. The entries are added to the response in ascending id order, with one caveat: each key can only appear in the response once. If the key is not yet included in the response, the entry is appended to the end. If it does exist, however, the latest entry replaces the older entry without changing the response order. Events have the same key when they involve the same sharable object, such as changing the same parameter multiple times. We only need to include the latest such value in the response, but we must still maintain the correct order so that sharable objects such as timeline branches are created before their days are configured. To conclude the response, the server records the maximum id of the included events, allowing the requesting client to reconnect at a later time and request subsequent events.

If there are no events with a greater id available in the database, then the server will hold the connection and wait for at least one to be inserted. While waiting, the server will also periodically write whitespace to test the connection and check if any listed users have been inactive for too long (inserting a disconnect event if one is found). To prevent constantly polling the database and wasting server CPU time, the server will sleep for 500 milliseconds between attempts. Due to the sleep time, clients will typically receive new events submitted by other clients within 500 milliseconds. We, of course, also have to contend with network latency on top of this. We cannot assume that planners will have the similar network and server capabilities, and likewise, it would be unwise to assume that all planners are in one physical location. Fortunately, slight delays like this are not a problem since users do not know exactly when their peers have submitted events, and as such they have no reference point for determining slower response times. Save for severe network issues out of our control, Sonata will fully synchronize participating clients before the contributing participant has a chance to verbally explain what they changed.

8.1.5 Supporting Larger Groups

It is important that we consider the collaborative experience when more than a couple of participants are involved. Originally, each Sonata client would automatically share all existing timeline branches when joining a collaborative session. This was put in place so that other users could see what they have been working on. Unfortunately, this feature has drawbacks if you are connected to a session with many participants: Sonata will have more branches open than it can fit on screen. Sonata was not designed to have more than four branches open at a time, and when every participant contributes at least one branch, the total number of branches open quickly adds up.

We improved the experience by changing two behaviors related to branches and views. First off, branches are no longer automatically shared with other participants. That means that users are able to have their own private branches in a collaborative session, and can choose which of those they would like to share with others. Second, creating or receiving a new branch no longer automatically partitions the view area and opens new views. Instead, users have the ability to choose which branches are visualized, whether private or shared. Users are able to configure both options from the list of branches in the view configuration menu.

8.2 Future Work

Collaboration in Sonata still has a ways to go before it fully meets the requirements of a multi-agency planning session. Much of this work will focus on role assignment, where the lead planner can limit which parameters other planners can change. These limitations will apply to broad groups of parameters in a given state or geographic region. As examples, consider vaccination in Colorado or zoning in Texas.

There is also an opportunity for us to add additional ways for collaborators to share their thoughts with each other. If possible, we would like to include some of these directly into the visualization itself. Examples of this include allowing users to draw on the map, chart, and table views. This functionality would enable planners to visually mark areas of interesting behavior and areas of focus. In addition to drawing, we would like to add functionality for pinning text notes and labels on various parts of the tool. At the very least, notes could be pinned to days on the timeline, augmented with convenient modification when setting parameters. This would enable collaborators to record their thought process on an optional basis when using Sonata, serving as inspiration to others or even themselves. All of the drawings, notes, and labels would be shared with other participants through the collaboration server.

9 Conclusions

In this dissertation, we have described our methodology that enables interactive, exploratory, and scalable visual analytics of voluminous spatiotemporal phenomena. Here, we briefly overview these accomplishments.

Sonata visualizes data produced by the real-time evaluation of machine learning models. The visualization is composed of several distinct views of the data, providing the user a complete view of the phenomena at hand. Users are able to interact with the visualization by configuring views to explore aspects of the data they desire, by interacting with views to review different locations spatially and temporally, and by influencing the models themselves by adjusting hundreds of input parameters. The sub-second result times provided by the models encourage continuous and interactive exploration of the dataset.

To manage the temporal dimension of the data, we provide users with a timeline abstraction. This timeline organizes the daily intermediate predictions of the models, predictions that are fed back into the models on subsequent days. Input parameters can be tuned each day, giving users finer control over predictions over time. The timeline underpins a key exploratory power of Sonata: contrasting. The timeline can branch on any day allowing users to visually compare and contrast diverging outcomes resulting from both small and large changes.

To cope with the vast spatial dimension of the data, we partition the geographical extent into smaller sectors. Each sector has its own set of models trained specifically on local behavior within the sector, allowing not only more accurate prediction of local spatiotemporal phenomena, but also enabling scalable localized outbreaks. The sectors influence each other over time, cooperating to form a complete picture of events at multiple scales beyond the scope of the source simulation, including the national scale.

To provide an interactive experience at scale, Sonata makes effective use of limited resources. Sonata's internal computation framework exists to perform this task. Through the framework's use of computation dependency tracking, progress bookkeeping, and result caching, we ensure that work is only performed when a relevant change of inputs is made and then only when the result is actually used. The framework's scheduler negotiates with WebWorker threads, improving response time though parallel model evaluation.

To enable multiple users to more easily contribute to the same planning exercise, we incorporate collaborative functionality into Sonata. The use of events to facilitate collaboration results in a substantially lower network footprint (in contrast to display-based collaboration) and allows a process to reconcile multiple concurrent actions in a collaborative environment. By exchanging events with a collaboration server, participants experience a visualization that is synchronized across several clients. Synchronization occurs in under a second and does not interfere with a participant's ability to interact and explore the data on their own terms.

In summary, the tasks that guided this research enabled the development of a framework that is capable of:

1) Capturing and reproducing spatiotemporal phenomena through the use of an Interaction Graph;
2) Visualizing spatiotemporal phenomena across multiple linked maps and graphs;
3) Influencing distinct spatiotemporal outcomes with support for direct comparison and contrast;
4) Intelligently executing model and visual code to make the most of limited resources;
5) Supporting higher-resolution visualization of millions of entities and their spatiotemporal interactions through the use of geospatial data tiling;
6) Enabling multiple small, but accurate regional models to collaborate in order to accurately model a significantly larger geospatial area;
7) Enabling multiple people to participate in visualization exercises with multiplayer functionality.

Furthermore, this research preserves the timeliness and fidelity of results in resource constrained environments, something beneficial to all visual analytics applications. Sonata, our what-if style spatiotemporal disease outbreak visualization tool for livestock populations, serves as an implementation of these ideas. Beyond this implementation, this research is applicable in a wide range of spatiotemporal visualization settings, including but not limited to city traffic planning, predicting climate phenomena, and managing commercial inventory and supply-chains.

References

[1] N. Harvey, A. Reeves, M. A. Schoenbaum, F. J. Zagmutt-Vergara, C. Dubé, A. E. Hill*, et al.*, "The North American Animal Disease Spread Model: A simulation model to assist decision making in evaluating animal disease incursions," *Preventive veterinary medicine,* vol. 82, pp. 176-197, 2007.

[2] *Apache Hadoop*. Available: http://hadoop.apache.org/

[3] *Apache Pig*. Available: http://pig.apache.org/

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax*, et al.*, "MillWheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment,* vol. 6, pp. 1033-1044, 2013.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax*, et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment,* vol. 8, pp. 1792-1803, 2015.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley*, et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2-2.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser*, et al.*, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135-146.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *NSDI*, 2010, p. 20.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, 2007, pp. 59-72.

[10] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment,* vol. 3, pp. 285-296, 2010.

[11] S.-M. Chan, L. Xiao, J. Gerth, and P. Hanrahan, "Maintaining interactivity while exploring massive time series," in *Visual Analytics Science and Technology, 2008. VAST'08. IEEE Symposium on*, 2008, pp. 59-66.

[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao*, et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data mining and knowledge discovery,* vol. 1, pp. 29-53, 1997.

[13] L. Lins, J. T. Klosowski, and C. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 19, pp. 2456-2465, 2013.

[14] Z. Liu, B. Jiang, and J. Heer, "imMens: Real-time Visual Querying of Big Data," in *Computer Graphics Forum*, 2013, pp. 421-430.

[15] D. Fisher, "Incremental, approximate database queries and uncertainty for exploratory visualization," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, 2011, pp. 73-80.

[16] D. Fisher, I. Popov, and S. Drucker, "Trust me, I'm partially right: incremental visualization lets analysts explore large datasets faster," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012, pp. 1673-1682.

[17] S. Joshi and C. Jermaine, "Materialized sample views for database approximation," *Knowledge and Data Engineering, IEEE Transactions on,* vol. 20, pp. 337-351, 2008.

[18] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS),* vol. 11, pp. 37-57, 1985.

[19] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 29-42.

[20] D. Rafiei, "Effectively visualizing large networks through sampling," in *Visualization, 2005. VIS 05. IEEE*, 2005, pp. 375-382.

[21] G. Niemeyer. (2008). *Geohash*. Available: http://en.wikipedia.org/wiki/Geohash

[22] M. Malensek, S. L. Pallickara, and S. Pallickara, "Galileo: A framework for distributed storage of high-throughput data streams," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, 2011, pp. 17-24.

[23] M. Malensek, S. Pallickara, and S. Pallickara, "Fast, ad hoc query evaluations over multidimensional geospatial datasets," *IEEE Transactions on Cloud Computing,* vol. 5, pp. 28-42, 2017.

[24] M. Malensek, S. Pallickara, and S. Pallickara, "Analytic queries over geospatial time-series data using distributed hash tables," *IEEE Transactions on Knowledge and Data Engineering,* vol. 28, pp. 1408-1422, 2016.

[25] M. Malensek, S. Pallickara, and S. Pallickara, "Evaluating geospatial geometry and proximity queries using distributed hash tables," *Computing in Science & Engineering,* vol. 16, pp. 53-61, 2014.

[26] C. Tolooee, M. Malensek, and S. L. Pallickara, "A scalable framework for continuous query evaluations over multidimensional, scientific datasets," *Concurrency and Computation: Practice and Experience,* vol. 28, pp. 2546-2563, 2016.

[27] W. Budgaga, M. Malensek, S. Lee Pallickara, and S. Pallickara, "A framework for scalable real-time anomaly detection over voluminous, geospatial data streams," *Concurrency and Computation: Practice and Experience,* vol. 29, p. e4106, 2017.

[28] S. L. Pallickara, S. Pallickara, M. Zupanski, and S. Sullivan, "Efficient metadata generation to enable interactive data discovery over large-scale scientific data collections," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010, pp. 573-580.

[29] S. Jensen, B. Plale, S. L. Pallickara, and Y. Sun, "A hybrid XML-relational grid metadata catalog," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, 2006, pp. 8 pp.-24.

[30] Y. L. Simmhan, S. L. Pallickara, N. N. Vijayakumar, and B. Plale, "Data management in dynamic environment-driven computational science," in *Grid-based problem solving environments*, ed: Springer, 2007, pp. 317-333.

[31] S. L. Pallickara, S. Pallickara, and M. Pierce, "Scientific data management in the cloud: A survey of technologies, approaches and challenges," in *Handbook of Cloud Computing*, ed: Springer, 2010, pp. 517-533.

[32] T. Buddhika, M. Malensek, S. L. Pallickara, and S. Pallickara, "Synopsis: A Distributed Sketch over Voluminous Spatiotemporal Observational Streams," *IEEE Transactions on Knowledge and Data Engineering,* vol. 29, pp. 2552-2566, 2017.

[33] T. Buddhika and S. Pallickara, "Neptune: Real time stream processing for internet of things and sensing environments," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 1143-1152.

[34] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Transactions on Parallel and Distributed Systems,* vol. 28, pp. 3553-3569, 2017.

[35] S. L. Pallickara and M. Pierce, "Swarm: Scheduling large-scale jobs over the loosely-coupled hpc clusters," in *SWARM: Scheduling Large-Scale Jobs over the Loosely-Coupled HPC Clusters*, 2008, pp. 285-292.

[36] W. Budgaga, M. Malensek, S. Pallickara, N. Harvey, F. J. Breidt, and S. Pallickara, "Predictive analytics using statistical, learning, and ensemble methods to support real-time exploration of discrete event simulations," *Future Generation Computer Systems,* vol. 56, pp. 360-374, 2016.

[37] M. Malensek, W. Budgaga, R. Stern, S. Pallickara, and S. Pallickara, "Trident: Distributed Storage, Analysis, and Exploration of Multidimensional Phenomena," *IEEE Transactions on Big Data,* 2018.

[38] M. Bender, R. Klein, A. Disch, and A. Ebert, "A functional framework for web-based information visualization systems," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 6, pp. 8-23, 2000.

[39] M. Bostock and J. Heer, "Protovis: A graphical toolkit for visualization," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 15, pp. 1121-1128, 2009.

[40] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 17, pp. 2301-2309, 2011.

[41] S. Shekhar, C. Lu, X. Tan, S. Chawla, and R. Vatsavai, "Map Cube: A visualization tool for spatial data warehouses," *Geographic data mining and knowledge discovery,* p. 73, 2001.

[42] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 8, pp. 52-65, 2002.

[43] G. McKenzie, K. Janowicz, S. Gao, J.-A. Yang, and Y. Hu, "POI Pulse: A Multi-granular, Semantic Signature-Based Information Observatory for the Interactive Visualization of Big Geosocial Data," *Cartographica: The International Journal for Geographic Information and Geovisualization,* vol. 50, pp. 71-85, 2015.

[44] *Leaflet*. Available: https://leafletjs.com/

[45] C. A. Steed, D. M. Ricciuto, G. Shipman, B. Smith, P. E. Thornton, D. Wang*, et al.*, "Big data visual analytics for exploratory earth system simulation analysis," *Computers & Geosciences,* vol. 61, pp. 71-82, 2013.

[46] A. Inselberg and B. Dimsdale, "Parallel coordinates: a tool for visualizing multi-dimensional geometry," in *Proceedings of the 1st conference on Visualization'90*, 1990, pp. 361-378.

[47] B. Smith, D. M. Ricciuto, P. E. Thornton, G. Shipman, C. A. Steed, D. Williams*, et al.*, "ParCAT: Parallel climate analysis toolkit," *Procedia Computer Science,* vol. 18, pp. 2367-2375, 2013.

[48] D. Kozma, I. Simon, and G. E. Tusnády, "CMWeb: an interactive on-line tool for analysing residue–residue contacts and contact prediction methods," *Nucleic acids research,* vol. 40, pp. W329-W333, 2012.

[49] S. K. Badam, J. Zhao, S. Sen, N. Elmqvist, and D. Ebert, "TimeFork: Interactive Prediction of Time Series," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 5409-5420.

[50] E. Horvitz, "Principles of mixed-initiative user interfaces," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1999, pp. 159-166.

[51] P. C. Wong, H. Foote, P. Mackey, G. Chin, H. Sofia, and J. Thomas, "A dynamic multiscale magnifying tool for exploring large sparse graphs," *Information Visualization,* vol. 7, pp. 105-117, 2008.

[52] C. Walshaw, "A multilevel algorithm for force-directed graph drawing," in *International Symposium on Graph Drawing*, 2000, pp. 171-182.

[53] L. Nachmanson, R. Prutkin, B. Lee, N. H. Riche, A. E. Holroyd, and X. Chen, "Graphmaps: Browsing large graphs as interactive maps," in *International Symposium on Graph Drawing and Network Visualization*, 2015, pp. 3-15.

[54] U. Dadi, C. Liu, and R. R. Vatsavai, "Query and Visualization of extremely large network datasets over the web using Quadtree based KML Regional Network Links," in *Geoinformatics, 2009 17th International Conference on*, 2009, pp. 1-4.

[55] S. Havre, E. Hetzler, P. Whitney, and L. Nowell, "Themeriver: Visualizing thematic changes in large document collections," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 8, pp. 9-20, 2002.

[56] W. Peng, M. O. Ward, and E. A. Rundensteiner, "Clutter reduction in multi-dimensional data visualization using dimension reordering," in *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, 2004, pp. 89-96.

[57] I. Jolliffe, *Principal component analysis*: Wiley Online Library, 2002.

[58] J. B. Kruskal and M. Wish, *Multidimensional scaling* vol. 11: Sage, 1978.

[59] M. Eichner, M. Schwehm, H.-P. Duerr, and S. O. Brockmann, "The influenza pandemic preparedness planning tool InfluSim," *BMC infectious diseases,* vol. 7, p. 17, 2007.

[60] K. R. Bisset, J. Chen, X. Feng, V. Kumar, and M. V. Marathe, "EpiFast: a fast algorithm for large scale realistic epidemic simulations on distributed memory systems," in *Proceedings of the 23rd international conference on Supercomputing*, 2009, pp. 430-439.

[61] (2009). *Didactic*. Available: http://ndssl.vbi.vt.edu/didactic/

[62] W. Van den Broeck, C. Gioannini, B. Gonçalves, M. Quaggiotto, V. Colizza, and A. Vespignani, "The GLEaMviz computational tool, a publicly available software to explore realistic epidemic spreading scenarios at the global scale," *BMC infectious diseases,* vol. 11, p. 37, 2011.

[63] A. Perez, M. AlKhamis, U. Carlsson, B. Brito, R. Carrasco-Medanic, Z. Whedbee*, et al.*, "Global animal disease surveillance," *Spatial and spatio-temporal epidemiology,* vol. 2, pp. 135-145, 2011.

[64] W. Budgaga, R. Stern, M. Malensek, C. Christensen, N. Harvey, S. Pallickara*, et al.*, "Cadence: Supporting Real-time What-If Explorations for Disease Outbreak Planning," *Under review*.

[65] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics,* pp. 1189-1232, 2001.

[66] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis,* vol. 38, pp. 367-378, 2002.

[67] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM,* vol. 18, pp. 509-517, 1975.

[68] R. Likert, "A technique for the measurement of attitudes," *Archives of psychology,* 1932.

[69] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: defining gamification," in *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*, 2011, pp. 9-15.

[70] S. G. Hart, "NASA-task load index (NASA-TLX); 20 years later," in *Proceedings of the human factors and ergonomics society annual meeting*, 2006, pp. 904-908.

[71] T. Lindström, D. A. Grear, M. Buhnerkempe, C. T. Webb, R. S. Miller, K. Portacci*, et al.*, "A Bayesian approach for modeling cattle movements in the United States: scaling up a partially observed network," *PLoS One,* vol. 8, p. e53432, 2013.

[72] M. G. Buhnerkempe, D. A. Grear, K. Portacci, R. S. Miller, J. E. Lombard, and C. T. Webb, "A national-scale picture of US cattle movements obtained from Interstate Certificate of Veterinary Inspection data," *Preventive veterinary medicine,* vol. 112, pp. 318-329, 2013.

[73] T. Lindström, S. A. Sisson, M. Nöremark, A. Jonsson, and U. Wennergren, "Estimation of distance related probability of animal movements between holdings and implications for disease spread modeling," *Preventive veterinary medicine,* vol. 91, pp. 85-94, 2009.

[74] T. Lindström, S. A. Sisson, S. S. Lewerin, and U. Wennergren, "Bayesian analysis of animal movements related to factors at herd and between herd levels: Implications for disease spread modeling," *Preventive veterinary medicine,* vol. 98, pp. 230-242, 2011.

[75] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE transactions on pattern analysis and machine intelligence,* vol. 17, pp. 790-799, 1995.

[76] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on information theory,* vol. 21, pp. 32-40, 1975.

[77] M. H. Hassoun, *Fundamentals of artificial neural networks*: MIT press, 1995.

[78] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological),* pp. 267-288, 1996.

[79] T. K. Ho, "Random decision forests," in *Document analysis and recognition, 1995., proceedings of the third international conference on*, 1995, pp. 278-282.

[80] L. Breiman, *Classification and regression trees*: Routledge, 1984.

[81] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics,* vol. 42, pp. 55-61, 2000.

[82] F. Mosteller and J. W. Tukey, "Data analysis, including statistics," *Handbook of social psychology,* 1968.

[83] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel*, et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[84] N. Shah, H. Shah, M. Malensek, S. L. Pallickara, and S. Pallickara, "Network analysis for identifying and characterizing disease outbreak influence from voluminous epidemiology data," in *Big Data (Big Data), 2016 IEEE International Conference on*, 2016, pp. 1222-1231.