FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Reactively Querying an Immutable Log of Relational Facts

Design and Implementation of a Bitemporal
EAV Database with a Subscription Query Language

Master Thesis

Submitted in partial fulfillment of the requirements for the degree of

Master of Science in Engineering

to the University of Applied Sciences FH Campus Wien

Master Degree Program
Software Design and Engineering

Author:

Albert Zak

Student identification number:

1810838004

Supervisor:

Priv.-Doz. Mag.rer.soc.oec.
Dipl.-Ing. Dipl.-Ing. Dr.techn.
Karl Michael Göschka

Date:

2020-05-31

Declaration of authorship:

I declare that this Master Thesis has been written by myself. I have not used any other than the listed sources, nor have I received any unauthorized help.

I hereby certify that I have not submitted this Master Thesis in any form (to a reviewer for assessment) either in Austria or abroad.

Furthermore, I assure that the (printed and electronic) copies I have submitted are identical.

Date: Signature:

Abstract

Growing customer demands lead to increased incidental complexity of data-intensive distributed applications. Relational Database Management Systems (RDBMS), especially those implementing Structured Query Language (SQL) possess performant but destructive-by-default write semantics. Modeling domains such as healthcare in classic RDBMS leads to an explosion in the number of columns and tables. Their structure has to be known in advance, discouraging an explorative development process. Requirements of real time collaboration, auditability of changes, and evolution of the schema push the limit of established paradigms.

The core of the data layer presented in this thesis is a simple relational model based on facts in the form of Entity-Attribute-Value (EAV) triples. A central append-only immutable log accretes these facts via assertions and retractions. Transactions guarantee atomicity/consistency/isolation/durability (ACID) and are themselves first-class queryable entities carrying arbitrary meta facts, realizing strict bitemporal auditability of all changes by keeping two timestamps: transaction time t_x and valid time t_v. Changes are replicated to clients which subscribe to the result of a query. Multiple incrementally maintained indices (EAVT, AEVT, AVET, VAET) grant efficient direct access to tuples, regarding the database analogous to a combined graph, column, and document store. The database itself is an immutable value which can be passed within the program and queried locally.

This work demonstrates the feasibility of implementing various desirable features in less than 400 lines of Clojure: bitemporality, audit logging, transactions, server-client reactivity, consistency criteria, derived facts, and a simple relational query language based on Datalog.

Key Terms

EAV
Database
Bitemporal
Immutable
Datalog
Lisp

List of Abbreviations

t_v	valid time
t_x	transaction time
6NF	6th normal form
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CQRS	Command Query Responsibility Segregation
CRDT	Conflict-free Replicated Data Type
CRUD	Create, Read, Update, Delete
DDD	Domain-Driven Design
DDP	Decentralized Data Protocol
DRP	Distributed Reactive Programming
DSL	Domain-specific language
edn	Extensible Data Notation
ES	Event Sourcing
FRP	Functional-Reactive Programming
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
lvar	logic variable
MUMPS	Massachusetts General Hospital Utility Multi-Programming System
ORM	Object-Relational Mapping
OT	Operational Transform
RAD	Rapid Application Development
RDBMS	Relational Database Management Systems

RDF Resource Description Framework

REST Representational State Transfer

RPC Remote Procedure Call

SaaS Software-as-a-Service

SPA Single Page Application

SQL Structured Query Language

UI User Interface

UUID Universally Unique Identifier

XML Extensible Markup Language

Contents

1 Introduction **1**
- 1.1 Problem . . . 2
- 1.2 Contribution . . . 2
- 1.3 State of the art . . . 3

2 Related Work **6**

3 Design **9**
- 3.1 Problems . . . 9
- 3.2 Goals . . . 11
- 3.3 Non-goals . . . 12
- 3.4 Conceptual model . . . 12
- 3.5 Query language . . . 14

4 Implementation **16**
- 4.1 Data model . . . 18
- 4.2 Writing . . . 20
- 4.3 Querying . . . 24
- 4.4 Publishing and subscribing . . . 27

5 Discussion **29**
- 5.1 Advantages . . . 29
- 5.2 Limitations . . . 30

6 Future Work **32**

7 Conclusion **33**

1 Introduction

This thesis describes a possible design and a proof-of-concept implementation of a data layer for near real time distributed collaborative applications targeted at operational workloads in small to medium sized businesses with strict auditability requirements.

Context. The ideas presented in this work are based the author's experience running a Software-as-a-Service (SaaS) business designing and operating an information system in the medical sector for the past five years (see figure 1).

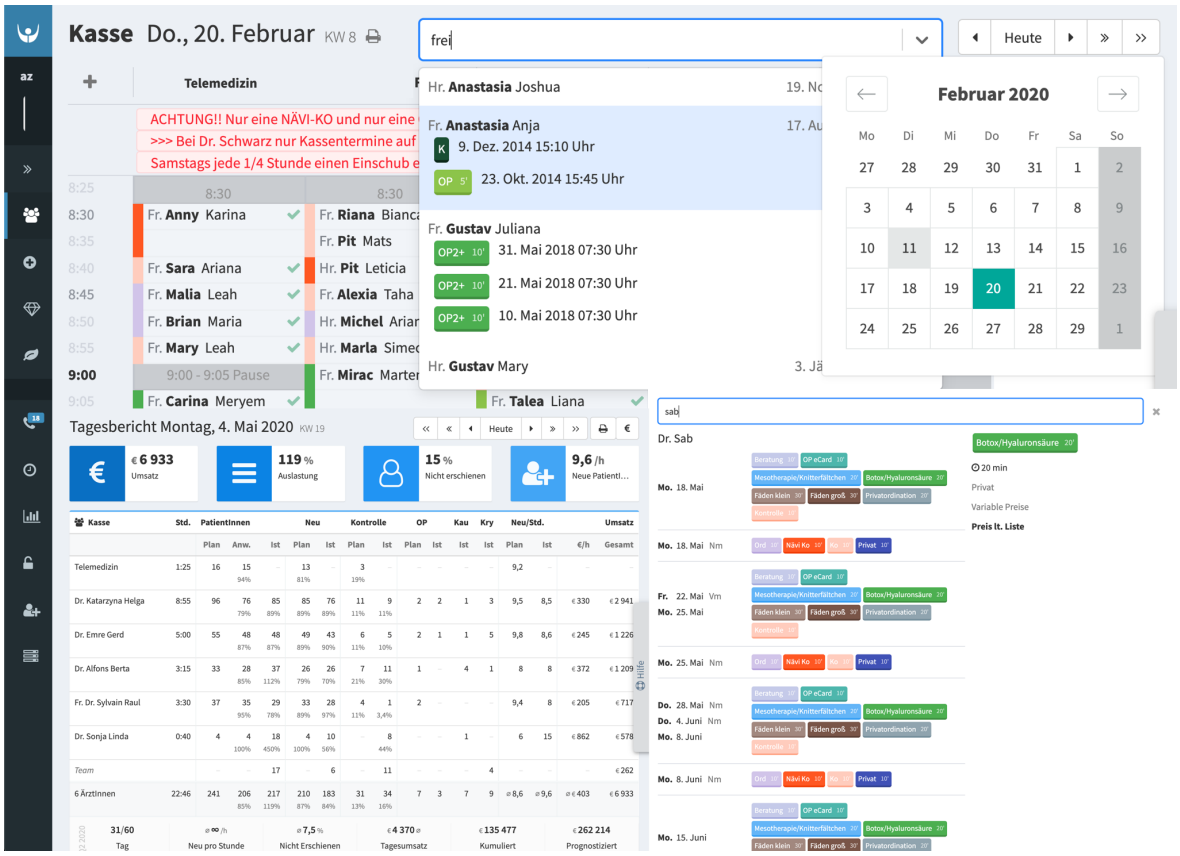

Figure 1: The author's medical information system

Outline. Section 2 presents related work. Section 3 exhibits common problems experienced in data-intensive business applications and shows a different approach towards solving some of them in a functional, immutable, and reactive way. A proof-of-concept implementation is described in section 4 and its merits and deficiencies are discussed in section 5.

1.1 Problem

Classic Relational Database Management Systems (RDBMS) are not suitable for modeling sparse, hierarchical or multi-valued data in domains where evolution and dynamism are hard requirements. They also lack a notion of memory or change over time because writes are destructive by default. Attempts to add concepts of temporality to Structured Query Language (SQL) are complicated and as a result not widely used. Auditing changes to a mutable black box with no history is hard. Developers fear the networking aspects of sending queries on a round trip "over there" [Hic12a] to the database, as opposed to having the data in memory and being able to query it directly. On top of all, customers demand increasingly interactive distributed collaboration environments, which pushes the limits of established request-response mechanisms.

1.2 Contribution

This thesis explains a possible design of a data layer for business applications and describes its prototypal implementation. The core is a simple relational data model based on facts in the form of Entity-Attribute-Value (EAV) tuples similar to the 6th normal form (6NF) or the Resource Description Framework (RDF) as seen in Semantic Web literature. Such EAV tuples are accreted in an append-only (immutable) log as assertions and retractions together with two timestamps: transaction time (t_x) at which the fact was added to the log, and valid time (t_v) at which the fact became true in the wider context of the system [Sno92], i.e. the real world.

Goal. The original goal of this work was to demonstrate the feasibility of an implementation of various desirable data layer features in less than 1000 lines total of readily comprehensible Lisp (Clojure/ClojureScript [Hic08]): Bitemporality, practically immutable audit logging, transaction metadata, server-client reactivity, schemalessness, consistency criteria, derived facts, versatile indexing, transactions with guarantees of Atomicity, Consistency, Isolation, Durability (ACID), and a simple relational query language based on Datalog.

Method. The thesis explains the decisions made while iterating on the design and its implementation and includes a discussion on limitations and advantages.

Results. Embracing the EAV+t_t+t_v data model together with the expressive programming language ClojureScript reveals that it is possible to implement a proof-of-concept data layer and query language for business applications with various valuable features not commonly seen in mainstream databases. The implementation is realized using less than 400 lines of code, the majority of which appears to map fairly well to the conceptual design.

1.3 State of the art

The system presented in this thesis builds on various ideas around better ways of dealing with of data, state, and process in distributed information systems.

Entity-Attribute-Value (EAV). Representing information in *attribute-value pairs* dates back to Lisp from the late 1950s [McC60]. Together with a value identifying the *entity*, the EAV data model is able to represent arbitrary data in a single table of just three columns (and many rows) by forcing normalization to the 6th normal form (6NF). The basic unit of information in EAV systems is the *fact*, which is a triple `[e a v]` consisting of:

(i) `e`, a value identifying the *entity* to which the fact is related,
(ii) `a`, an *attribute* value, and
(iii) `v`, the value itself.

The EAV data model is also known as *open schema*, or *vertical database* [JP15]. Medical information systems TMR [SHS83] and HELP [HHS+94] were the first to bring this way of knowledge representation to a clinical context in the 1970s.

EAV later gained popularity through the Semantic Web and the Resource Description Framework (RDF) [DMVH+00] where facts are called object-attribute-value triplets. In domains where where the number of attributes related to an entity is large and modeling such data in a classical Relational Database Management Systems (RDBMS) would require dealing with the overhead of defining and storing many columns, the EAV model provides an efficient and simple way to deal with sparse data.

```
[[:patient/91 :name "Hye-mi"]
 [:patient/91 :room :room/32]
 [:room/32 :building "A-12"]
 ...]
```

Listing 1: A sequence of EAV facts

While all *current* state of the system can be represented conceptually as in listing 1 as a single (long) sequence of facts, it would be useful to know exactly *how* the system got into its current state.

Logs. 16th century seafarers invented the *log* [MH73]. With a piece of wood – the log – attached to a string and thrown into the water, they could track its speed relative to the ship over time and record the measurements in a logbook, along with other events of interest [Kak19].

In data-intensive applications, the log is today often treated as a side product of mutating central state in a database, to aid in understanding situations after something has gone wrong. However, ideas such as Event Sourcing (ES) have reversed the roles of log and database and dictate that state is inferred from an accretion of events.

Event Sourcing. Systems which derive their state by consuming an ordered sequence of events are called *event sourced*. In practice, event sourcing is commonly associated with, and used in conjunction with Domain-Driven Design (DDD) [Eva04] and the architectural pattern of Command Query Responsibility Segregation (CQRS) [KJB12].

DDD tends to be employed in domains where the shape of the data is mostly known in advance and generally remains rather static over the lifetime of the system, because domain events are generally tied to the code because they are modeled as classes closed for modification.

CQRS is an architectural separation of the read paths from the write paths. In the case of classic event sourcing, this implies that when new data comes in as part of an event, the event is first appended to the global log. Afterwards, one or more materialized views (in a regular RDBMS) are updated from which the application finally performs its reads, as reading from the unwieldy log directly would be inefficient and complicated.

Immutability. Tying these ideas together with the EAV data model yields an event sourced database, where events are the most granular write operations on each field, akin to the transaction log of a classical database. To better convey the immutability of the log, the terms *assertion* and *retraction* are used in place of creation and deletion. To change a fact, one issues a transaction which simultaneously retracts the previous value and asserts the new one. An immutable EAV database consequently is the accretion of assertions and retractions of facts over time.

Value orientation. Instead of thinking about the database as a "place" where to send queries to fetch data from and write data to, the idea pioneered by Datomic is that the entire state of the database at any given point in time should be an *immutable value* which can be passed around in the program. Because it is immutable and a local value, it never changes. This has the implication that subsequent queries on the same value always result in the same results, giving a *stable basis*.

(Bi-)temporality. When extending EAV databases and the idea of Event Sourcing (ES) with a first-class concept of time and/or transactions, such systems are sometimes referred to using the ambiguous abbreviation *EAVT*, with the *T* variously referring to either the addition of transaction time, valid time or other *domain time* values (or a combination thereof) [HC13] or the *T* may hint at the concept of first-class transactions, where a fact additionally carries the entity ID of its originating transaction to allow attaching arbitrary metadata to events. Listing 2 shows a fact with a pointer to a *transaction entity*, about which bitemporal and other transaction metadata exists as first-class facts. This example demonstrates that the patient was actually (t_v) moved to room 32 at 18:30, but the fact was recorded in the system (t_x) later at 21:12 by user 43 as part of transaction 4.

```
[[:patient/91 :room :room/32 :txe/4]
 [:tx-id/4 :t_v #inst "2019-05-31T18:30:00"]
 [:tx-id/4 :t_x #inst "2019-05-31T21:12:00"]
 [:tx-id/4 :tx-by :user/43]
 ...]
```

Listing 2: Transaction metadata are first-class facts

While uni- or monotemporal databases allow querying along the transactional sequence of historic database states (called t_x), support for bitemporality adds another separate time axis t_v, meaning the time at which the fact came into existence in the *context* of the system. Note the there is no limit on the storage of arbitrary additional time-related facts related to an entity or a first-class transaction, which are referred to as *domain time*. A (bi-)temporal database only provides efficient and convenient indexing and query capabilities for t_x and t_v values. Users querying for any other values, including domain time, must use the general querying mechanisms.

Functional-Reactive Programming (FRP). Originally conceived to describe graphical animations [EH97], the FRP paradigm applies nicely to distributed data-intensive systems where it yields composable building blocks which are relatively easy to reason about [RDP14].

FRP fits particularly well inside the view layer of a Single Page Application (SPA) to construct the view out of (nested) pure functions $v = f(s)$ which take a global state value and return a description of what to render. Each component function specifies what slice of the state it needs, recursively composes smaller components, and is automatically re-run when the state changes.

Apart from web development, [SDM13] gives an overview of the literature on the more general Distributed Reactive Programming (DRP) paradigm.

2 Related Work

Rapid Application Development (RAD). Scaling developer productivity as the team size goes $n \to 1$ on data-intensive near real time collaborative line-of-business applications goes back to the RAD movement of the 1990s sparked by tools such as the Delphi suite [MCBDT00], PowerBuilder [Zub97], Lotus Notes [Zub97], and FileMaker [CA10].

MUMPS. On the aspects of near real time state synchronization combined with RAD tooling, a notable mention is the Massachusetts General Hospital Utility Multi-Programming System (MUMPS) [Bow79] dating back to the year 1966. An extremely terse programming language combined with a runtime system featuring built-in persistence mechanisms. Its "global database" is a subscripted array which replicates and persists *code and data* across machines in near real time. Major hospitals and financial institutions continue to run their daily operational workloads on various evolutions of MUMPS systems today [AR18].

Firebase and Parse. These services brought real time state synchronization to web and mobile applications starting in 2011. Firebase is a proprietary managed service operated by Google. Its data model is based on a global mutable structure in JavaScript Object Notation (JSON) and allows only rather simplistic filtering and navigational queries. Parse was a similar real time Backend-as-a-Service based on MongoDB and Redis which was active from 2011 to 2017. [WRG19]

Meteor. This thesis is an evolution based on the author's experience running a SaaS business built on the open source Meteor [SMDG14] framework and the Decentralized Data Protocol (DDP) [SWO12] released in 2012. Meteor allows clients to transparently subscribe to the results of predefined MongoDB queries updating live over arbitrary durations. Synchronization is achieved by the server watching the operations log of the database (called "oplog tailing") [WRG19], keeping track of the documents and queries of each connected client, and computing delta changes to send back to the client. See table **??** for a comparison of Meteor with other data management approaches.

Clients cache the received documents in a local instance of MongoDB implemented in JavaScript (called Minimongo), enabling arbitrary local queries with no latency. Combined with a functional-reactive view library such as React to bind the management of a view's lifecycle with its data subscriptions, this setup gives the impression of data "just being here" on the client with almost no programmatic overhead.

The major problem of Meteor is its absolute dependence on MongoDB and the ensuing lack of a relational data model, i.e. no joins and requiring to denormalize data, leading to incidental complexity and the $n + 1$ query problem.

Other databases which allow subscribing to the result set of a query include RxDB for JavaScript [WRG19], Supabase which is built in Erlang/Elixir and adds query-subscription capabilities to Postgres [CWdlR20].

	DBMS	Real-time database	Data stream management	Stream processing
data	persistent collections		persistent & ephemeral streams	
processing	one-time	one-time & continuous	continuous	
access	random	random & sequential	sequential	
streams	structured			structured & unstructured
	PostgreSQL MongoDB	Firebase, Meteor, RethinkDB, Parse	Influx, PipelineDB, SQLStream	Storm, Samza, Flink, Spark

Table 1: Comparison of Meteor with other database and stream processing systems [Win17]

Datomic. Most of the design tenets for the project described in section **??** originate from the database Datomic [Hic12a] which was created by the author of the Clojure programming language in 2012. Its data model is a monotemporal (transaction time t_x only) and immutable log of assertions and retractions of EAV facts, with first-class transactions being reified as regular queryable facts themselves. Datomic requires a minimal schema to be specified upfront.

Instead of thinking of the database as a single *place* to send queries to and receive tuples from, Datomic separates its components as follows to achieve horizontal read scalability, turning the classical model of a database "inside out" [Kle17]:

- There is a *generic* stateful distributed storage layer, which may be backed by any pluggable implementation (SQL, Redis, Riak, Kafka, files, memory...). The storage layer keeps all data in the form of tuples and various indices. [HHG19].
- Application servers (called "peers") operate on a working set of tuples in memory. They perform queries themselves *locally on their own copy of the data.* Tuples are cached locally as needed and are lazily loaded from the storage layer over the (local) network, where disk locality is considered irrelevant [AGSS11]. As put by the author: "perception does not require coordination [Hic12b]", yet consistency and caching are trivial because tuples are immutable.
- All write transactions – represented as data structures – go through a *single* instance of the *transactor* component, which is the only part of the system that may write to the storage backend. Because it is one single threaded instance, it can impose a global order on all incoming transactions. Conversely, the transactor is the bottleneck for write throughput, as well as a single point of failure for which a second instance should be operated in a hot failover configuration. Peers can place first-class functions into the database, and call them as part of a transaction to enforce arbitrary data invariants [HH$^+$19].

Crux. Rooted in the same core principles as Datomic but having taken some different architectural choices, Crux is an efficient *bitemporal* schemaless unbundled document database. For a detailed comparison with Datomic, see [PTM+19].

Datascript et al. Similarly inspired by Datomic, DataScript [PFH+14] is an immutable schemaless in-memory EAV store. It lacks notion of time because it is designed to manage frequently changing client-side state of a SPA within the browser for the lifetime of a session, using limited memory.

Several attempts exist to bridge the gap between a Datomic-like database on the server and a DataScript instance on the client used simultaneously for client-only view state as well as for caching tuples from the server [S+16]; as well as the gap between that database and the view layer [PK+15, K+19]. Datahike [K+18] is a port of DataScript back to the server, where it adds durable persistence. It aims to be a single-node replacement for Datomic suitable small projects.

Mozilla Mentat [ATK+18] was an attempt at a performant embeddable implementation in Rust of a combination of ideas from Datomic, DataScript, and SQLite.

Eva [PHD+19] is an open source monotemporal EAV database very similar to and compatible with the Application Programming Interface (API) of Datomic, though the project appears to be abandoned in a feature-complete "alpha" quality stage.

Ittyon [Ree16] is a Clojure library to manage distributed state in games based on ideas from Entity-Component Systems (ECS) combined with the EAV+t_x data model. Its client/server architecture via *channels* inspired the implementation of the prototype described in section 4.

LogicBlox. A notable proprietary commercial RAD self-serve database system with an integrated programming language based on Datalog, LogicBlox [AtCG+15] similarly aims to reduce incidental complexity of developing line-of-business applications with a focus on probabilistic and predictive analytics of large business data sets. Its developers have contributed a novel efficient algorithm for incremental view maintenance for Datalog systems [Vel12].

Hyperfiddle. Built on Datomic to interactively create a User Interface (UI) for database applications, the ongoing Hyperfiddle project [Get18] aims to be a Domain-specific language (DSL) to fully describe an application using only plain expressions of Extensible Data Notation (edn).

Eve. Ambitious but ultimately abandoned, the Eve project [CGM+16] was an attempt to reinvent programming for "humans first" through a combination of development environment, database, and a novel relational and reactive programming language where e.g. the concept of identifier scope is abandoned, and the order of statements has no semantics.

3 Design

The contribution of this work is divided into two main sections, design and implementation. The subsequent parts of this section first present various design problems of commonly employed data layer technologies. Deriving from their limitations, the next part paints a blissful picture of what an ideal data layer would look like (subsection 3.2), while the following demarcates the scope of the contribution (subsection 3.3). Finally, the conceptual model (subsection 3.4) and the query language (subsection 3.5) are presented.

3.1 Problems

Traditional RDBMSs enforce a structural rigidity of fitting data into "tables", "rows", and "columns", entailing a lack of flexibility in dealing with sparse data, irregular data, hierarchies, or multi-valued attributes [Hic12a]. With the shift to data-intensive Single-Page Applications (SPAs), clients become limited peers to the database. Requirements of real time collaboration, evolution of the schema, and auditability of changes, together with limited request-response data loading mechanisms work together to increase the incidental complexity of application code.

Structured Query Language (SQL). Despite the widespread use of SQL, it is not, in any version, an accurate reflection of the relational model [Cod90] [MM06]. Modeling domains in common RDBMS requires distinguishing between entity and relationship [Che76]. A system in which choosing the structure for the data involves setting up "routes" between data instances (such as from a particular employee to a particular department) is access path dependent. A pure relational systems would need no problematic distinction between entity and relationship [MM06].

SQL within the application's main programming language is usually treated as a second class citizen by resorting to concatenation of strings with a thin layer of sanitization. (Microsoft's LINQ [MBB06] is a notable exception.) Features such as materialized views are cumbersome to use because the language lacks powerful means of abstraction and composition [ASS96] which would enable recursively composing complex queries out of smaller queries.

Object-relational mismatch. Object-Relational Mapping (ORM) of tuples from the database onto whichever data types are available in the backend programming languages, and increasingly also to the language in the front end in SPA, as well as the choice of appropriate communication interfaces between at least three layers pose additional considerations. Duplicating structure in schema definitions and again in ORM code leads to a proliferation of types.

Time. Temporal databases are commonly optimized for analytics performance over large numeric datasets, e.g. sensor readings over time. Temporal extensions to SQL [KM12] are convoluted and do not address the fundamental problem at hand, which

is that typical mutable-state systems obscure the sequence of events that brought the world into its current state.

Distribution. Thinking that data resides "over there" on a remote server breeds a "fear of round trips" and causes developers to accumulate incidental complexity in the form of performance optimizations related to caching [Hic12a].

Yet, one cannot disavow the fundamental complexities of distributed systems: Network failures (latency, disconnection, offline context), consistency and availability choices taken by underlying technology which are not made explicit, and how different representations of data influence and determine what is and is not possible in terms of concurrent activity by different actors [Eme14].

Imperative fetching. Performing data loads via classic request-response semantics is not expressive enough because the act of fetching is made explicit and requires imperative calls to various endpoints. A developer of a SPA with a Representational State Transfer (REST) backend must decide along which boundaries to split the data endpoints. In traditional RESTful style, each *resource* dictates its own endpoint with its own Create, Read, Update, Delete (CRUD) actions. This approach places the burden of orchestrating a decomposed rich interaction on the clients [Cal15], leading to the $n + 1$ query problem when performing client-side joins, and an over-fetching of data which is not needed to display but happens to be provided by the endpoint.

At the other extreme, there exists one distinct endpoint providing the exact required data for each screen (notwithstanding that the concept of a *screen* is vague in SPAs) which collects, filters, and joins the data on the server before serializing it as one tree, and sending it back to the client. This style causes incidental complexity through a proliferation of endpoints and ad-hoc schemata.

A middle-ground mixture of both styles is common, muddling the architectural waters and bringing both disadvantages together with even more code bloat. In any case, the act of fetching the data must be initiated by the client in an imperative way instead of data just being "here" in the client's view layer when needed.

Propagation of updates. The traditional understanding of queries for data at rest as in listing 3 is that data exists at the database in one place, and queries are passed over there, usually in the form of SQL strings, and unroll themselves into a dataflow tree of operators, pulling data up the tree when they reach the leaves [Alv15].

```
(Π [:name :department]
   (⋈ (σ :name "Scott" employees)
      departments))
```

Listing 3: Querying data at rest [Alv15]

When the underlying data changes, queries do not get re-run automatically and clients are left looking at stale data. On the flip side, re-running the same query at different times is not guaranteed to yield the same results, as the database might have changed in the mean time. There are no facilities to temporally stabilize a basis for queries.

3.2 Goals

To reduce some of the complexities in question, [MM06] recommend adopting functional and declarative programming constructs, along with a fully relational data model. The described system should provide a flexible *data layer* for small to medium size internal business applications for distributed collaboration in near real time, optimized for developer productivity but still backed by relational guarantees.

Notion of memory. The requirement of strict auditability of *everything* implies keeping a queryable history of structured facts. An event sourced data model with explicit time and memory based on an accretion of facts fits this requirement. Especially when structured in 6NF/EAV, their combination affords a "save everything, query later" way of thinking by being access path independent.

Streaming relational queries. Recall the querying semantics for data at rest in listing 3. What is needed for rich interactions is a different model of streaming queries for data in motion: What is "just there" is not the data, but the query — which is *instantiated in the network*, and the data flows through the query instead of the other way around. Listing 4 gives an example of a streaming query.

```
(Π [:ip]
   (⋈ (σ (classifier email :spam)
         email-source)
      contacts-source))
```

Listing 4: Querying data in motion [Alv15]

Transactional guarantees. A proper data layer for an information system must afford the same transactional guarantees of Atomicity/Consistency/Isolation/Durability (ACID) as classical relational databases.

Adaptivity and dynamism. Yet, such a system must also be malleable, adaptive, and can not require developers to settle e.g. on a fixed database schema design upfront. Instead, the system must be able to quickly co-evolve with the domain it is serving.

Distribution. The handling of data between clients and servers should be as declarative as possible: Clients declare what data they are interested in, and the infrastructure takes care of fetching, caching, handling updates, and managing the lifecycle of the subscription. The same infrastructure must be written in a way such that it is able to run on servers, web browsers and mobile devices with no modification.

3.3 Non-goals

Efficiency. No attention is paid to the efficiency of compute and memory usage. Tradeoffs are almost always made in favor of clarity concerning the mapping between conceptual model and implementation of the proof of concept. The only major optimization is the fact that the triple index structure exists, leastwise it doubles as the simplest possible way to access arbitrary data without the overhead of parsing and executing a query.

Custom indexing strategies, e.g. ways to maintain a phonetic index to query for people's names, do not need to be part of the database design, because the triple indexing scheme combined with the ability to derive facts (explained in section 4) is general enough to allow arbitrary access to the data in an efficient-enough manner without having to declare indices upfront.

First class bitemporality. Another non-goal is the efficiency, primacy, and expressive power of the provided bitemporal affordances. A vast amount of previous work exists on relatively complex attempts to add efficient bitemporal semantics to relational databases [SBJS96, JS99, KM12] notably for use in bitemporal constraints [DFG+97] or within complex queries, and of bitemporality as a concept in production rule systems [AtCG+15]. Bitemporality in the described system is only secondary. Access paths are optimized for the most recent view of the data, while bitemporality is meant to be used for infrequent auditing purposes. There is no bitemporal index, consequently issuing queries with temporal modifiers is allowed to cause a sequential scan of the log.

Standard protocols. Lastly, the design explicitly avoids compliance with existing proliferated standards around the handling of data such as SQL, Extensible Markup Language (XML), JSON, REST, etc. to instead allow quick exploration into different paradigms unburdened by past decisions.

3.4 Conceptual model

The data model of the presented system is extremely simple. There is no requirement to design a schema or to differentiate between entities and relationships. Yet, *all* structured data can be represented in the system as long as data is fully normalized (6NF).

- The basic unit of information is a *fact*, a triple `[e a v]` containing values representing *entity*, *attribute*, and *value*.
- Over time, facts are *asserted* and *retracted*, accreted as part of a *transaction*.
- The database and all changes to its state over time are fully described by the transaction log of assertions and retractions of facts.

index	name	feels like	good for
EAVT	"entity-oriented"	document store	accessing various attributes of a known entity
AEVT	"attribute-entity-oriented"	column store	accessing the same attribute of various entities
AVET	"attribute-value-oriented"	filtering a column store	finding entities by the value of a specific attribute
VAET	"value-oriented"	searching everything	searching over all values, regardless of attribute

Table 2: Impact of the index sort order on the area of application

Indexing. EAV systems commonly keep a number of sorted indices (see table 2) to allow the data to be retrieved from multiple "angles" or directions, depending on the need of the query. Index structures are named after the *nesting order* in which the elements of the facts are arranged. Not all database systems maintain the same indices. In this case, the system keeps four indices covering the following common use cases:

- EAVT, the canonical order, which *maps* an entity to its attributes like a document,
- AEVT, for finding entities which *have* a certain attribute set
- AVET, for *filtering* entities by a known attribute set to a known value,
- VAET, for *searching* over all attributes of all entities by a known value.

For example, here is a simple example to pull out the name of a known patient, using only the `get-in` function of the Clojure core library on the `:eavt` index:

```
(get-in db [:eavt :patient/91 :name])
```

One can also leave out the attribute, and get back a map (as a conceptual *document*) containing all known attributes related to that entity.

```
(get-in db [:eavt :patient/91])
```

Performing a search by name over all patients is similarly trivial using the `:avet` index, with the result

```
(get-in db [:avet :name "Hye-mi"])
```

3.5 Query language

The query language of the system is a greatly simplified language modeled after the pattern matching relational query language used in Datomic, which is in turn a Lisp variant of the Datalog [AV88] language expressed in of Clojure's edn.

The choice of language is arbitrary – any relational language would suffice – and the core of the database does not depend on any query language capabilities Modeling the language after the one used in Datomic was chosen because because not only has the edn notation become a de-facto standard for other EAV databases like Crux, EVA, and Datascript, but because the shape of each query clause maps naturally to the representation of a fact in canonical EAV order.

See listing 5 for an query consisting of four query clauses (the `:where` part) performing an implicit join, and a final projection (`:find`) to extract the values bound to the *logic variable (lvar)* symbols `?name` and `?location`. For example, the query clause `[?p :name ?name]` applied to the fact `[:person/123 :name "Hye-mi"]` would result in *binding* the lvar `?p` to the value `:person/123`, and the lvar `?name` to the value `"Hye-mi"`. Other clauses are bound likewise. Note that multiple occurrences of the same lvar prompt *unification* with the same value, creating an implicit *join*. The order of the query clauses has no semantic meaning.

Performing a query entails applying the `q` function to a database value and a query. Clients can thus decide whether to leverage the query language via loading a library, or just access the data via the index structures directly.

```
'[:find [?name ?company]
  :where [[?p :works-for ?e]
          [?e :name ?company]
          [?p :name ?name]
          [?p :location "Ulsan"]]]
```

Listing 5: "Who from Ulsan is working for whom?"

Temporal and bitemporal queries. As stated in section 3.3, the (bi-)temporal aspects of the described system are secondary – they are to be used for infrequent auditing purposes. Consequently, the design of the indexing and query mechanisms can be greatly simplified be forgoing bitemporal indexing strategies such as [NEE95].

As the query function simply takes a database as a *value*, a *filtering function* can be applied the the database beforehand. The `keep` function in listing 6 returns a structurally shared and lazy copy of the database filtered by arbitrary bounds of the relevant timestamps t_x and t_v.

```
(q (keep
     (λ [t_x t_v]
       (and (> t_v 300) (< t_v 500)
            (< t_x 700)))
     db) query)
```

Listing 6: Applying a temporal filter before querying

Per-entity history. A common use case in auditing is to retrieve the *history* of all attributes related to a given entity over time. The `history` function takes a database value (optionally composed with a filtering function as described above) and an entity value, and returns an ordered slice of the log with transactions relevant to the requested entity. Note that it does not make sense to create a new database value from a history log, because that would just result in only the latest values being present in the index yet again.

Publication and subscription One of the goals states that clients should be able to declaratively subscribe to the *live result set* of a query. The results and the query itself will change over the duration of a client's session. Each change triggers an immediate re-render of the UI. Conceptually, clients *install* their *subscription queries* on the server, and the infrastructure will re-run the subscription query whenever the underlying data changes and notify the client of the changed results. The design does not prescribe whether or not to replicate past (i.e. superseded or retracted) facts, thus greatly simplifying the proof-of-concept implementation by deferring concerns such as diffing, authorization, and the decision of what exactly to replicate to the clients to the developer customizing this data layer to their use case.

Security. While extreme dynamism may be warranted in a high-trust environment, a real-world application may interact with some malicious entities and thus needs a means to restrict queries on the server side. In a real-world application, clients would need to authenticate themselves and the server would authorize publication based on access rules. Yet, there is no simple way to statically analyze queries submitted by the client for safety properties, but the server can control which facts are allowed to be replicated to a client. A publication might, for example, choose to not replicate facts with specific attributes, or transform facts to censor parts of the value.

4 Implementation

The following subsections describe the implementation of the design explained in the previous section. The conceptual data model from section 3.4 is translated to Clojure in section 4.1. Next, a guided tour follows the *data path* from a new fact originating on a client all the way through the server and to the other clients, showing and explaining the simple functions that transform it along the way.

Functional core, stateful shell. While Clojure does not enforce functional purity, it is idiomatic to push impure computation and state towards the edge of the system. In fact, core functionality described in the next subsection is implemented as pure functions operating on plain immutable data structures. Such purity allows the same core of the database to run unmodified on servers and on clients. Clients and server obviously need to keep some state, but each does so within one single place only. Three pure namespaces make up the functional core of the system:

(i) `core` implementing the basic pure functions and data structures to create and transact facts,

(ii) `index` implementing pure functions to place facts into indices,

(iii) `query` implementing pure functions to provide basic querying capabilities with a language based on edn Datalog.

The minimal impure parts stay within the `client` and `server` namespaces. Their task is to handle the asynchronous connection with each other (`connect!`, `disconnect!`, `receive`, `broadcast!`) and mutate their local copy of the database (`transact!`)

Figure 2: Two connected clients in the testbed

The testbed. To aid quick experimentation while developing, all interactions between the server and the clients are simulated within a single web page. The scaffolding in the `testbed` namespace presents a simple UI to add and remove client connections, to transact facts, and to perform queries, all while providing insight into the full current state of each simulated node, see figure 2. The testbed can inject arbitrary delay into Clojure's `core.async` *channels* used to simulate WebSocket connections between nodes. The channels implementation of the testbed is based on [Ree16].

4.1 Data model

The implementation relies heavily on the lazy immutable default data structures [Hic09] provided by the Clojure core library, which remain performant even when used in strange ways thanks to structural sharing [Oka99]. This section extends and clarifies the design from section 3.4 and translates it into Clojure.

- A *fact* is represented as a vector, its elements can be of any Clojure value type.

```
[e a v]
```

- A *transition* is either an *assertion* or *retraction*, a vector containing the splatted fact as its first three elements, along with an indicator whether to assert (`:+`) or retract (`:-`) the preceding fact.

```
[e a v :+]
```

- A *transaction* is a vector containing an arbitrary number of transitions, which are to be atomically applied to the log *at some point in the future.* Note that a transaction does not mutate the database, it is at this point just a data structure expressing intent to assert or retract facts. A transaction may contain *meta facts* about itself, using the "magic" placeholder value for the transaction entity `:tx-meta`.

```
[[e a v :-] [e a v :+] [:tx-meta a v :+]]
```

- A *commit* (listing 7) is a transaction that was successfully applied to the database, meaning that any consistency criteria succeeded and a new database value containing the updated state was produced. It has similar shape and contents as the transaction it represents. Each transition of the committed transaction contains as its last element the same newly-generated transaction entity reference `txe`. There is also now at least the t_x transaction meta fact added and optionally any derived facts.

```
[[e a v :+ txe]
 [e a v :- txe]
 [txe :t_x t_x :+ txe]]
```

Listing 7: Structure of a commit

- The *log* is an ordered linked list of all commits. All changes over time to the database are fully described by the log, with the newest commit being appended to the beginning: `'([...] [...] ...)`.

- An *index* (listing 8) is an associative nested structure (map). There are four indices, each three layers deep: `:eavt`, `:veat`, `:avet`, named after the nesting order of their keys with the last `t` referring to the transaction entity `txe`.

```
{:index
  {:eavt {e {a {v txe}}
          e {a {v txe}} ...}
   :aevt {a {e {v txe}}
          a {e {v txe}} ...}
   :avet {a {v {e txe}}
          a {v {e txe}} ...}
   :vaet {v {a {e txe}}
          v {a {e txe}} ...}}}
```

Listing 8: Structure of the indices

- Finally, the *database* (db) is a map containing the log list and the index map.

```
{:log '() :index {}}
```

Commits do not mutate the database. Note that a commit itself has no notion of place or state, and does not mutate anything, it only signifies that a transaction was successfully applied to *some* database value *somewhere*. A commit *may* mean that the server has successfully updated its state and broadcast the commit to all connected clients, or, since the database value is immutable, it may just be the result of any local *"as-if"* dry run.

Transaction meta facts. A client may supply arbitrary transaction metadata, except t_x which is always set by the server. To send metadata, the client adds transitions to the transaction, which have the magic value `:tx-meta` set as their entity. Before committing a transaction, the server will replace this value in the transitions with a newly generated entity.

4.2 Writing

This subsection follows a simple fact in canonical EAV format, originating on a client, through its transformations until it ends up on the server's canonical source-of-truth database. Starting with the fact

```
[:patient/91 :name "Hye-mi"]
```

and calling `assert` on it yields an *assertion transition*, which is created through *conjoining* (appending as last element) the `:+` keyword with the fact vector. Similarly, calling `(retract fact)` conjoins `:-`. The result, the example case, is an assertion vector:

```
[:patient/91 :name "Hye-mi" :+]
```

The database only accepts transitions to be atomically applied as part of a transaction vector. For example, to update a value, simultaneously retract the previous value and assert a new value for the same combination of entity. The variadic `transact` function only applies its arguments to `vector`, creating a transaction vector:

```
[[:patient/91 :name "Hye-min" :-] [:patient/91 :name "Hye-mi" :+]]
```

On cardinalities. What would happen if one were to assert multiple values for the same combination of entity and attribute? Should any previous values be retracted automatically, resulting in one value being "true" at a time? Should all values stay current until retracted? The path chosen in this implementation was to only support the more general case of *defaulting to multi-valued cardinalities on all attributes* (which also happens to be simpler to implement). This may lead to unexpected results in queries. A production system would need to define a schema and enforce cardinalities of *many* or *one* on each attribute depending on the requirements of the domain model.

Server interface. This transaction can now be committed to a database value, either locally to create a new database (sometimes also referred to as a "what-if" transaction, as it does not affect the value on the server), or globally for all clients by transacting on the server. In production systems, a server should likely not accept arbitrary transactions from clients. Instead, the database is protected by regular Remote Procedure Call (RPC) / REST endpoints which, after authentication, authorization, and sanitization accept only specific parameters to transact.

The testbed server accepts one type of message, a vector beginning with the keyword `:transact` containing the entire transaction vector nested as its second element:

```
[:transact [[:patient/91 ... :-] [:patient/91 ... :+]]]
```

When such a transaction message is received, the server applies the transaction and afterwards swaps the global state atom (atomically mutates the reference via a call to `swap!`) to the new database value. Clients do not attempt to *optimistically* update their local database but rather wait for the server to confirm the transaction and broadcast the final *commit* back to all clients. Safe means for performing optimistic updates remain a topic that needs further research. Clients simply listen to a message tagged `:commit` and upon receipt proceed to swap their local database value. Be aware that the server may send different commits to each client, depending on their subscription query (explained later in subsection 4.4).

ACID transactions. Thanks to Clojure's immutable data structures, implementing transactions that respect ACID guarantees is almost trivial. The pure function `(transact db transaction txe now)` takes a database value, a transaction vector, a new unique entity `txe` and the current time which is to be used as transaction time t_x. The `txe` value is usually either a globally incrementing number or a randomly generated string like a Universally Unique Identifier (UUID) version 4. It is up to the impure calling code on the server to generate and supply these two values. The `transact` function returns a *new* database value with the transaction committed, i.e. appended to the log and with the indices updated. Transacting is a nested multi-step process:

- `commit`: Generates another data structure which describes the commit about to happen. The same arguments as to `transact` are passed along, resulting in the commit data structure depicted in listing 9:

```
[[:patient/91 :name "Hye-min" :- :txe/1]
 [:patient/91 :name "Hye-mi" :+ :txe/1]
 [:txe/1 :t_x t_x :+ :txe/1]]
```

Listing 9: A commit of one retraction and two assertions

To generate this data structure, `commit` performs the following steps:

  - `(conj transaction [txe :t_x now :+])`: Conjoins the non-optional assertion of the t_x meta fact with the supplied transaction vector.
  - `(map #(conj % txe) transaction)`: Conjoins the `txe` value as the last element of each transition in the amended transaction vector returned from the previous step.
  - `(derive-transitions transaction db txe now)` Some previously added facts in the database may be of a special attribute which indicates that the value is a *derivation function*. These functions receive a pending database value and may return additional assertions or retractions, for example to keep a phonetic index on people's names updated. The behavior of `derive-transitions` is described later.
  - `(validate db)` Finally, the pending commit is applied to a new database, and its installed constraint functions are checked. The behavior of `validate` is described later.

- `apply-commit`: When the previous steps generated a description of a commit, the last step is to apply the commit: Append it to the log and update all indices. Since the commit being applied was generated from the same basis value of the database, it is guaranteed that all consistency criteria are honored and that this call cannot produce an invalid database.

Derivation functions. Functions are first-class values in any functional language, even more so in a Lisp where all code is represented as data structures, and all data structures can be evaluated as code – and passed around and stored in the database. These affordances allow the database to *react to changes to itself, on itself*, without side effects.

Users can transact *derivation functions* into the database, which are just facts with the special `:db/derive` attribute and a function as a value. This function must have the signature:

```
(λ [db commit])
```

In the process of creating a commit these derivation functions receive the value of the database and the most current accumulated value of the commit being generated. They are expected to return a *transition vector*, of which any transitions returned will be added to the current commit, before the updated commit vector is passed on to the next derivation function.

Derivation functions cannot mutate the commit vector, but may issue *immediate* retractions of facts contained therein. The order in which installed functions are evaluated is not specified. Since there is no way to enforce functional purity, and they may, but are not recommended to, have side effects.

Consistency constraints. Enforcing consistency of the database after all derivation functions have been evaluated is similarly trivial to implement: All current facts with a `:db/validate` attribute have as their value a function with the same signature as derivation functions. For the transaction to commit, every validation function is expected to signal consistency by returning `true`.

Note that there are no constraints which need to be *disabled* during a transaction, but rather the final database value (after deriving facts and adding them to the current commit) is checked for consistency just once.

Indexing. The four indices `:eavt`, `:aevt`, `:avet`, and `:vaet` are created by *folding* over the reversed commit log. Listing 10 shows the entirety of the indexing logic. Transaction boundaries are irrelevant because creating the index is atomic anyways, so the reversed log is flattened before being passed to the fold. Updating an index happens after a commit is finalized, and has the fundamental property of being *incremental*, meaning that to update an existing index, only the newest commit needs to be folded in with the existing index.

The fold is implemented as a reducing function with the database value as its initial element, and with `index` as its combining function, thus folding one commit into the index at a time.

The `index` function destructures each element of the commit vector into its sub-elements, and decides the updating function to use based on whether it is an assertion (then associate `txe` along the index path given by `e a v`) or a retraction (then dissociate). This is why building the index needs to happen in reverse order of the log. Even though the design of the system selected four common indices to cover general querying use cases, it is a trivial one-line change to the implementation of the `index` function given in listing 10 to reduce or extend the set of indices to populate. The key to note here is that assertions cause the value of the fact to be *associated in* the index (`assoc-in`), while retractions dissociate values. The entire index structure, being immutable, is *threaded* ($\rightarrow$) through successive associations to place the value inside each index.

```
(defλ index [db [e a v op txe]]
  (update db :index
    (λ [index]
      (let [index-in (case op
                       :+ #(assoc-in %1 %2 txe)
                       :- dissoc-in)]
        (→ index
          (index-in [:eavt e a v])
          (index-in [:aevt a e v])
          (index-in [:avet a v e])
          (index-in [:vaet v a e]))))))

(defλ create-index [db]
  (reduce index db
    (flatten (reverse (:log db)))))

(defλ update-index [db commit]
  (reduce index db commit))
```

Listing 10: Updating the index

Listing 11 shows the resulting index structures after applying the commit to the database. Note that the retracted fact is not part of the index anymore, it is only accessible by scanning the structured commit log.

```
{:index
  {:eavt {:patient/91 {:name {"Hye-mi" :txe/1}}
          :txe/1 {:t_x {t_x :txe/1}}}
   :aevt {:name {:patient/91 {"Hye-mi" :txe/1}}
          :t_x {:patient/91 {t_x :txe/1}}}
   :avet {:name {"Hye-mi" {:patient/91 :txe/1}}
          :t_x {t_x {:txe/1 :txe/1}}}
   :vaet {"Hye-mi" {:t_x {t_x :txe/1}}
          t_x {:t_x {:txe/1 :txe/1}}}}}
```

Listing 11: Fully populated indices after transacting

4.3 Querying

Now that the fact has been committed to the database on the server, it is time to look at the means available to get answers to questions by querying the database value. Simply looking up data directly via the indices is the preferred way to read. There is no query parsing and execution overhead, since the data is "just here" inside a sorted and easily walkable structure. A developer using this data layer should be familiar with the use cases and performance characteristics of the various indexing structures. Refer to earlier section on the design of the index structure for examples and a comparison of use cases. This subsection sketches the implementation of the edn Datalog query engine, a completely separate and optional library. The implementation is a re-write based on [Rub15] with macro-heavy code and metadata being replaced by pure functions and maps, among other simplifications. This section will look at the example query in listing 12 and trace it through a full evaluation, starting with main entry point, the `q` function: `(q db query)`.

```
[:find [?name]
 :where [[?e :name ?name]
         [?e :room :room/32]]]
```

Listing 12: "Who's in room no. 32?"

Step 1: Filter the log by time. As per design, nontrivial and performant queries are only supported for the *most recent* view of the database. There is, however, a simple way to trade *some* performance for querying the state at an arbitrary past point of transaction time t_x: By constructing a new database value from the current state with its log truncated at some point. More generally, one can also construct a new database out of an arbitrary slice of the log, doing so however may lead to unexpected results because data asserted by "older" facts might be missing as it was cut out in the filtering stage.

Step 2: Build a filtering sieve triple. The general working principle of the engine is that each of the *query clauses* is parsed into a *filtering sieve*, that is a vector containing functions which examine each part of the fact and decide whether the value being fed *matches* the expected *data pattern* of the initial clause. In the example, the first query clause: `[?e :name ?name]` is transformed into the following structure, containing the sieve and a positional mapping back to the logic variable:

```
{:sieve [true #(= % :name) true] :lvars ["?e" nil "?name"]]}
```

The first nested vector is the sieve. It matches all facts which have an attribute of `:name`, and it does not care about entity or fact values (`true` will always match any value in that position). The second nested vector keeps track of the names and positions of the lvars from the supplied query clause, because later, when matching facts are found, their values need to be *unified* with the lvar bindings.

The second query clause is transformed alike:

```
                          [?e :room :room/32]
  {:sieve [true #(= % :room) #(= % :room/32)] :lvars ["?e" nil nil]]}
```

The original macro implementation of the query engine by [Rub15] additionally allows use of any desired unary or binary predicate to be called as part of the matching process by simply wrapping them in a similar sieving function. A production application must take care, e.g. by whitelisting only certain server-side functions as predicates, that clients cannot submit malicious side-effecting code as part of a query. This work borrows the original design from [Rub15] and simplifies its implementation as in listing 13.

```
(defλ make-sieve [term]
  (cond
    ; Logic variables do not filter anything yet
    (lvar? term)
    (λ [_] true)

    ; Unary operators must hold: [valid? ?x]
    (and (vector? term)
         (= 2 (count term))
         (lvar? (second term)))
    (λ [t] ((first term) t))

    ; Binary operator, lvar is first operand: [> ?age 18]
    (and (vector? term)
         (lvar? (second term)))
    (λ [t] ((first term) t (last term)))

    ; Binary operator, lvar is last operand: [< 18 ?age]
    (and (vector? term)
         (lvar? (last term)))
    (λ [t] ((first term) (second term) t))

    ; Constants must match exactly
    :else
    (λ [t] (= term t))))
```

Listing 13: Constructing a predicate sieve

query clause	joining variable operates on	index to use
`[:e :a ?v]`	value	`:eavt`
`[:e ?a :v]`	attribute	`:vaet`
`[?e :a :v]`	entity	`:avet`

Table 3: Mapping of joining variable position to query index, slightly adapted from [Rub15]

Step 3: Decide index to use. The query clauses are now parsed into a sieve which is ready to be evaluated over an index. Descending an index means successively going from knowledge to answer, i.e. evaluation needs to start with an index which has as its top level value one of the constants provided in the query. In other words, the lvar that appears at the same position within each query clause is the *joining variable*. The index which to descend is the one that keeps this joining variable at its leaves [Rub15]. This is a design limitation, as a full query language should support arbitrary transitive joins, leveraging multiple indices. Refer to table 3 for clarification.

Step 4: Run sieve function over filtered index According to the query design goals, the order of the query clauses has no semantic meaning, yet there is an effect on query evaluation performance: when the most restricting clauses are placed at the beginning, they are processed first and immediately reduce the cardinality of the set of potential results which the next clauses have to filter through. Automatically optimizing clause order would require estimating the cardinality of the resulting sets, which is a complex undertaking and is thus omitted from this proof-of-concept [NM11, MBC07].

The `descend-index` function first needs to *reorder* the sieve functions according to the selected index. Since the sieve is always given in canonical EAV order, before it can be used to descend e.g. an `:avet` index, the sieve needs to be rearranged so that e.g. the function matching the attribute is moved from second to first position and so on. Actually descending the chosen index is a mechanical iteration, calling each sieve function on its respective index level and accumulating the results.

The result of this sieving process is a *superset* of facts matching *any* of the given query clauses, but with no attention paid to the correct binding and unification of their values with the respective logic variables of the query. The last steps of query resolution are identical to the initial implementation by [Rub15] and consequently are not further described here.

4.4 Publishing and subscribing

To keep the implementation of the publication/subscription mechanism simple, a single client only subscribes to one publication at a time. It does so by sending the following message to the server, which causes the submitted query data structure to be *installed* on the client socket:

```
[:subscribe query]
```

An installed query by itself does not do anything, as it is just a data structure. The server must ensure that whenever its global copy of the database value is *swapped* for an updated value after a successful commit, the interested clients are notified of that commit. To do so, the server walks the list of connected clients and in turn feeds each installed query to the server's *publication function*.

A developer using this data layer in an application needs to set up that custom publication function beforehand. It receives the updated database value and the client's installed query, and is expected to return a *commit* of the changes that should be sent to the client.

Note that the client's queries being passed to the publication function are not expected to be compatible with the described query engine, or be queries at all – a subscription query can be any data structure, and it is up to the developer using this data layer to supply a publication function which is able to decide, based on the query submitted by the client, what commit should be replicated.

Clients may choose to amend their subscription query at any later point in time by sending a new `[:subscribe query]` message. A simplistic mapping function from the new database value and the client's current query would not have enough information to decide what parts of the commit need to be replicated, as the client might have initially received some facts pertaining to its initial subscription but has re-subscribed with different parameters in the mean time. The publication function thus may incorrectly assume that the client already has its local database populated with all facts matching the current subscription, whereas the client actually still has all the facts of the previous subscription yet will receive facts for its current one. Consequently, the publication function is also passed the client's previous query, and the transaction entity of the last transaction that was replicated to the client:

```
(defλ publish [db new-query previous-query last-txe])
```

The return value of the publication function is either `nil` to take no action when the client's subscription is not affected by the latest commit, `[:commit commit]` to command the client to commit the commit, possibly modified, or, as a last resort if e.g. the query has changed substantially and no diff can reasonably be deduced, the publication function can return `[:reset log]` to instruct the client to start anew and swap its value to a completely new database created from the attached log. Note that the server never sends index structures over the wire because clients can trivially regenerate them.

Nontrivial publications. The publication function may also modify facts, e.g. censor some values matching sensitive attributes or choose to not publish facts matching some pattern. As a consequence of the simplistic notification mechanism, the server may send more or less, or different facts than the client asked for. While the body of the publication function can be trivial and always return the latest commit regardless of query, developers building larger systems will likely need to put a lot of effort into writing an efficient, secure, and correct publication function. The simplistic design of the publication/subscription mechanism appears to be the weakest point of the presented data layer, as it only shifts to the developer almost all of the complexity related to efficiently distributing and diffing results from changed queries.

5 Discussion

The following two subsections qualitatively review the contribution with respect to the stated goals, and expose limitations of the current implementation as well as discuss flaws inherent to the design.

5.1 Advantages

Expressive power. The minimal code size shows the expressive power of the language and its standard library of immutable data structures.

Database as a value. Being able to treat the entire database as an immutable value which can be passed around the program and queried locally is a powerful affordance which slashes incidental complexity.

Strong auditability. Accreting assertions and retractions of facts in an immutable log gives complete auditability of all changes with no programmatic overhead.

Flexible data model. Attribute-oriented data models can represent sparse data much more efficiently than traditional RDBMS, at the cost of frequent joins [Göb19]. Together with first class transactions and a notion of memory over time, EAV+t_x+t_v appears to be a promising data model for complex domains.

Simple bitemporality. Relaxing requirements of e.g. complex temporal queries and instead focusing on auditability yields a simple yet useful design. The system defers answering hard questions about e.g. the handling of future-dated facts and efficient bitemporal indexing as they are not a core concern.

Library, not a framework. Apart from enforcing data to be fully denormalized, the presented design places no boundaries on miscellaneous concerns or structure of the embedding application. A developer is free to compose this "data layer as a library" with any other libraries or frameworks as they see fit.

Easily extensible. Customizing the behavior of the database is simple, as exemplified by the implementation of the `document` function which allows atomic ingestion of multiple facts related to an entity as arbitrarily nested documents. Another example would be the client side use of a single database to store both local UI state and "global" facts from the server, the only change necessary is to filter out attributes with a `:local` namespace before transacting the change to the server.

5.2 Limitations

Query engine. No negation, no aggregations, no pull, no multi-way joins, no destructuring – the minimalist implementation of the query language presented in this thesis does not even deserve comparison to a proper Datalog.

No schema. Determining if fact was superseded needs upfront specification of cardinality. A production system would need to define a schema and at least enforce cardinalities on each attribute, i.e. the number of values that can be considered "current" for an attribute, e.g. that a person can have *many* phone numbers but only *one* birthdate.

Coupled storage. The design of the storage layer makes no provisions for offloading responsibilities to a generic storage backend or even durable persistence via local disks.

Coupled transactor. Splitting out the transactor component as in Datomic, with the goal of increasing availability in a failover configuration, is not possible in this design.

Limited subscriptions. Clients have to explicitly subscribe and manage the lifecycle of available publications, which requires duplicate, imperative, and error-prone programming effort. Another limitation of the implementation is that a client can only be subscribed to one query at a time. The mechanism to notify clients of updated facts, being critical for security and privacy, was designed ad-hoc and has not been tested thoroughly.

Time-aware queries. A complex query might want to mix different timelines in single query, e.g. join previous values with current values. This is not possible in the simplistic presented design. Datalog, however, has been extended with temporal semantics in various works [AMC+10, AtCG+15].

Stand-alone queries. The query engine provides no means for abstracting, parameterizing, and composing fragments of queries in a *hygienic* way so that expected lexical scoping semantics of logic variables are observed.

Dynamic typing. During implementation and refactoring of the prototype, a clear majority of bugs were caused by a mismatch of expected vs. actual structure of data types being passed around. While the highly dynamic nature of the language enables extremely quick experimentation, a small number of functions would have benefitted from a lightweight layer of gradual typing, as afforded by e.g. `clojure.spec` [PR19].

Incompatibility. Simultaneously changing the data model, its representation and storage formats, the interfaces and protocols, query languages and semantics requires doing away with proliferated standards, decades of established research, and proven implementations of the real world.

Switching costs. Since persistence of data is fundamental to any application, changes to the data layer of existing systems are prohibitively expensive. Yet, the same dynamic proves advantageous in green field projects unencumbered by legacy decisions.

Privacy regulations. No production system can be deployed without a way to permanently erase personal data. The concept of *excision* as in Datomic, or the use of two separate data storage components with only hashes being written to immutable logs as in Crux present clean ways around this issue.

Open vs. closed system. Incidental complexity within a closed system such as the one described in this work can be managed to stay at a minimal level because any outside components the system is interacting with is forced to adapt to the system's way. However, users of such a system eventually want to connect it to other things. Inevitably, doing so *imports* some of the incidental complexities of those systems. It remains to be seen if open systems can be built without importing accidental complexity [Mof16].

Second system effect. This work is the author's second attempt dealing with the problems of data layers for business applications. Brooks observed the tendency of successful systems to be succeeded by over-engineered, bloated systems, due to inflated expectations and overconfidence of the authors [BJ+95].

6 Future Work

Safe concurrent editing. A distributed system expects connection loss and simultaneous conflicting edits. It should be possible to define a schema that selects one of many built-in conflict resolution strategies specific to the domain requirements of each attribute. A per-field specifiable tradeoff as dictated by the theorem of C and A must propagate to the clients and dictate the possible operations on the data item in question given the current network conditions [Eme14].

A DRP approach by [MS14] focuses strongly on selectable consistency guarantees, while Conflict-free Replicated Data Type (CRDT) and Operational Transform (OT) are recently discovered concepts which appear to provide composable consistency primitives for robust replication [WKB15, WKB16].

The Braid protocol [TLWB19] is an in-progress draft of a proposed Internet Engineering Task Force (IETF) standard to add history, editing, and subscription semantics to HTTP resources. It aims to standardize the representation and synchronization of arbitrary web application state. Braid can allow simultaneous editing of the same resource by different clients and servers and can guarantee a consistent resulting state via selectable *merge types* implementing various CRDTs and OTs.

(Temporal) logic constraints. An ideal programming environment would let the programmer "use logic to express what is true, use logic to check whether something is true, [and] use logic to find out what is true [ASS96]". Research by [AMC+10, ACHM11] explores temporal extensions to Datalog and a domain-specific language for describing time-dependent behavior of distributed systems.

Full stack laziness. A fully lazy distributed data structure would allow transparent access and local caching of all facts for which the client passes access rules set up by the server. Such a design would also allow transparent querying of past facts, possibly aided by hints from the programmer as to where (on client or server) the query should be executed.

Incremental maintenance. Efficient execution of Datalog programs installed as "live" queries entails incremental updating of the result set as the source data changes. Research in the direction of incremental view maintenance in such systems includes timely dataflow [MMI+13], differential dataflow [MMII13], and 3DF providing an implementation of reactive Datalog for Datomic [Göb19].

Data as code. As the presented system's flexibility allows storing, versioning, replicating and querying arbitrary data, including functions, the question arises of how an entire application can be constructed with all its code existing as facts inside the database, replicating to the clients – thus closing the circle back to MUMPS-like systems.

7 Conclusion

Easy things are easy, hard things are hard. This thesis set out to redesign and implement the entire data layer using a combination of non-mainstream ideas. From data representation in EAV facts, to the database being an immutable value to be passed around and queried locally, to only accreting facts via assertion and retraction, yet keeping ACID guarantees, to having functions stored as values inside the database which derive new facts or act as constraints, to a replication mechanism where clients subscribe to the live-updating result set of a query, to pulling out simple values directly from the database by reaching into its index structures, or asking complex questions with a pattern matching query language.

The resulting implementation in less than 400 lines is impressively tight yet appears to map nicely to the initial design, mostly thanks to the incredibly expressive Clojure language and its built-in immutable data structures. While all of the mentioned features are implemented to a degree that is just enough to experiment with, almost all of the more complicated aspects were left simply left out of the design: There is no expectation of performance, efficiency, scale, security, safety, testing, or any implied suitability for usage in the real world or with more than just a handful of sample facts. The proof-of-concept fixates on the easy parts of that utopian data layer design which were almost trivial to implement, and barely covers any truly complex minutiae. In particular, the implementation of the query language turned out to be harder than anticipated, despite cutting out almost all but the most basic pattern matching features of a real Datalog.

Yet the formidable degree to which the presented ideas appear to mesh together, supported by considerable amount of related work, gives a sound impression of the general direction of this and similar designs for better data layers.

List of Figures

1 The author's medical information system 1
2 Two connected clients in the testbed 17

Tables

1 Comparison of Meteor with other database and stream processing systems [Win17] 7
2 Impact of the index sort order on the area of application 13
3 Mapping of joining variable position to query index, slightly adapted from [Rub15] 26

Listings

1 A sequence of EAV facts 3
2 Transaction metadata are first-class facts 5
3 Querying data at rest [Alv15] 10
4 Querying data in motion [Alv15] 11
5 "Who from Ulsan is working for whom?" 14
6 Applying a temporal filter before querying 14
7 Structure of a commit 18
8 Structure of the indices 19
9 A commit of one retraction and two assertions 21
10 Updating the index 23
11 Fully populated indices after transacting 23
12 "Who's in room no. 32?" 24
13 Constructing a predicate sieve 25

References

[ACHM11] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[AGSS11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, volume 13, pages 12–12, 2011.

[Alv15] Peter Alvaro. I see what you mean. In *Talk at StrangeLoop Conference*. University of California Santa Cruz, 2015.

[AMC+10] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In *International Datalog 2.0 Workshop*, pages 262–281. Springer, 2010.

[AR18] Raymond Aller and Richard L Rydell. The evolution of the cmio in america. pages 1–10. Productivity Press, 2018.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, 1996.

[AtCG+15] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382, 2015.

[ATK+18] Nick Alexaner, Emily Toop, Grisha Kruglov, et al. Mozilla mentat, a persistent, relational store inspired by Datomic and DataScript. `https://github.com/mozilla/mentat`, 2018. GitHub repository, accessed 2020-05-31.

[AV88] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. 1988.

[BJ+95] Frederick P Brooks Jr et al. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education, 1995.

[Bow79] Jack Bowie. Methods of implementation of the MUMPS global data-base. *Medical Informatics*, 4(3):151–164, 1979.

[CA10] Weihua Chen and Metin Akay. Developing EMRs in developing countries. *IEEE Transactions on Information Technology in Biomedicine*, 15(1):62–65, 2010.

[Cal15] Bobby Calderwood. From REST to CQRS with Clojure, Kafka, & Datomic. In *Talk at Clojure/Conj Conference*, 2015.

[CGM+16] Joshua Cole, Chris Granger, Corey Montella, et al. Eve: Programming designed for humans. `http://witheve.com`, 2016. accessed 2020-05-31.

[Che76] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1):9–36, 1976.

[Cod90] Edgar F Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[CWdlR20] Paul Copplestone, Ant Wilson, and Angelico de los Reyes. Supabase. `https://supabase.io/docs/library/subscribe`, 2020.

[DFG+97] Anne Doucet, Marie-Christine Fauvet, Stéphane Gançarski, Geneviève Jomier, and Sophie Monties. Using database versions to implement temporal integrity constraints. In *International Workshop on Constraint Database Systems*, pages 219–233. Springer, 1997.

[DMVH+00] Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of xml and rdf. *IEEE Internet computing*, 4(5):63–73, 2000.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, 1997.

[Eme14] Chas Emerick. Distributed systems and the end of the api. 2014.

[Eva04] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[Get18] Dustin Getz. A hypermedia tool for making web apps in real-time. 2018.

[Göb19] Nikolas Cintahoron Göbel. Optimising distributed dataflows in interactive environments. Master's thesis, ETH Zurich, Computer Science Department, 2019.

[HC13] Vojtech Huser and James J Cimino. Desiderata for healthcare integrated data repositories based on architectural comparison of three public repositories. In *AMIA Annual Symposium Proceedings*, volume 2013, page 648. American Medical Informatics Association, 2013.

[HH+19] Rich Hickey, Stuart Halloway, et al. Datomic documentation. Cognitect, Inc. `https://docs.datomic.com/on-prem/index.html`, 2019. accessed 2020-05-31.

[HHG19] Rich Hickey, Stuart Halloway, and Justin Gehtland. Datomic: The fully transactional, cloud-ready, distributed database, 2019.

[HHS+94] Stanley M Huff, Peter J Haug, Lane E Stevens, Robert C Dupont, and T Allan Pryor. Help the next generation: a new client-server architecture. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*, page 271. American Medical Informatics Association, 1994.

[Hic08] Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, 2008.

[Hic09] Rich Hickey. Persistent data structures and managed references, 2009.

[Hic12a] Rich Hickey. Database as a value. In *Talk at GOTO Conference, Chicago*, 2012.

[Hic12b] Rich Hickey. Values and change – clojure's approach to identity and state. Technical report, Technical report, Clojure, 2012.

[JP15] Torben Jastrow and Thomas Preuss. The entity-attribute-value data model in a multi-tenant shared data environment. In *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 494–497. IEEE, 2015.

[JS99] Christian S Jensen and Richard T Snodgrass. Temporal data management. *IEEE Transactions on knowledge and data engineering*, 11(1):36–44, 1999.

[K+18] Konrad Kühne et al. Datahike, a durable datalog implementation adaptable for distribution. `https://github.com/replikativ/datahike`, 2018. accessed 2020-05-31.

[K+19] Denis Krivosheev et al. re-posh, use your re-frame with DataScript as a data storage. `https://github.com/denistakeda/re-posh`, 2019.

[Kak19] Neil Kakkar. The human log. 2019.

[KJB12] Jaap Kabbedijk, Slinger Jansen, and Sjaak Brinkkemper. A case study of the variability consequences of the cqrs pattern in online business software. In *Proceedings of the 17th European Conference on Pattern Languages of Programs*, pages 1–10, 2012.

[Kle17] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[KM12] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL: 2011. *ACM Sigmod Record*, 41(3):34–43, 2012.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, 2006.

[MBC07] Tanu Malik, Randal C Burns, and Nitesh V Chawla. A black-box approach to query cardinality estimation. In *CIDR*, pages 56–67, 2007.

[MCBDT00] Hugh Mackay, Chris Carne, Paul Beynon-Davies, and Doug Tudhope. Reconfiguring the user: Using rapid application development. *Social studies of science*, 30(5):737–757, 2000.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[MH73] William Edward May and Leonard Holder. *A History of Marine Navigation. With a Chapter on Modern Developments by Leonard Holder*. G. T. Foulis Co. Ltd., Henley-on-Thames, Oxfordshire, 1973.

[MM06] Ben Moseley and Peter Marks. Out of the tar pit. *Software practice advancement*, 2006.

[MMI+13] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[MMII13] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[Mof16] Rob Moffat. All about Eve. `https://robmoff.at/2016/01/13/all-about-eve/`, 2016. accessed 2020-05-31.

[MS14] Alessandro Margara and Guido Salvaneschi. We have a dream: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 142–153, 2014.

[NEE95] Mario Nascimento, Margaret H Eich, and Ramez Elmasri. *IVTT: A Bitemporal Indexing Structure Based on Incremental Valid Time Trees*. Citeseer, 1995.

[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*, pages 984–994. IEEE, 2011.

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[PFH+14] Nikita Prokopov, Ben Fleis, David Thomas Hume, et al. DataScript, immutable database and Datalog query engine for Clojure, ClojureScript and JS. `https://github.com/tonsky/datascript`, 2014.

[PHD+19] Eric Petersen, Ryan Heimbuch, Timothy Dean, Ross Hendrickson, Brandon St. Jules, Tyler Wilding, et al. Eva, a distributed database-system implementing an entity-attribute-value data-model that is time-aware, accumulative, and atomically consistent. https://github.com/Workiva/eva, 2019. accessed 2020-05-31.

[PK+15] Matt Parker, Denis Krivosheev, et al. Datomic – datascript syncing/replication utilities. https://github.com/denistakeda/posh, 2015.

[PR19] Gheorghe Pinzaru and Victor Rivera. Towards static verification of clojure contract-based programs. In *International Conference on Objects, Components, Models and Patterns*, pages 73–80. Springer, 2019.

[PTM+19] Jon Pither, Jeremy Taylor, Dan Mason, Håkan Råberg, Malcolm Sparks, Johanna Antonelli, James Henderson, and Ivan Fedorov. Crux documentation, section on "comparisons". https://opencrux.com/docs#faq-comparisons, 2019. accessed 2020-05-31.

[RDP14] Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-tier functional reactive programming for the web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 55–68, 2014.

[Ree16] James Reeves. Ittyon - library to manage distributed state for games. https://github.com/weavejester/ittyon, 2016. accessed 2020-05-31.

[Rub15] Yoav Rubin. An archaeology-inspired database. In *The Architecture of Open Source Applications: 500 lines or less*, 2015.

[S+16] Christopher Small et al. Datomic – DataScript syncing/replication utilities. https://github.com/metasoarous/datsync, 2016.

[SBJS96] Richard T Snodgrass, Michael H Böhlen, Christian S Jensen, and Andreas Steiner. Adding valid time to SQL/temporal. *ANSI X3H2-96-501r2, ISO/IEC JTC*, 1, 1996.

[SDM13] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. Towards distributed reactive programming. In *International Conference on Coordination Languages and Models*, pages 226–235. Springer, 2013.

[SHS83] William W Stead, William E Hammond, and Mark J Straube. A chartless record—is it adequate? *Journal of medical systems*, 7(2):103–109, 1983.

[SMDG14] Geoffrey Robert Schmidt, Nick Martin, Matthew DeBergalis, and David L Greenspan. Live-updating web page generation, June 5 2014. US Patent App. 13/691,732.

[Sno92] Richard T Snodgrass. Temporal databases. In *Theories and methods of spatio-temporal reasoning in geographic space*, pages 22–64. Springer, 1992.

[SWO12] Sashko Stubailo, Hugh Willson, and Avital Oliver. Ddp specification. `https://github.com/meteor/meteor/blob/devel/packages/ddp/DDP.md`, 2012. accessed 2020-05-31.

[TLWB19] Michael Toomim, Greg Little, Rafie Walker, and Byrn Bellomy. Braid-HTTP: Synchronization for HTTP. `https://datatracker.ietf.org/doc/html/draft-toomim-httpbis-braid-http`, 2019.

[Vel12] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[Win17] Wolfram Wingerath. The case for change notifications. 2017.

[WKB15] Christian Weilbach, Konrad Kühne, and Annette Bieniusa. replikativ. io: Composable consistency primitives for a scalable and robust global replication system. *CoRR*, 2015.

[WKB16] Christian Weilbach, Konrad Kühne, and Annette Bieniusa. Decoupling conflict resolution with cdvcs. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–6, 2016.

[WRG19] Wolfram Wingerath, Norbert Ritter, and Felix Gessert. Real-time databases. In *Real-Time & Stream Data Management*, pages 21–41. Springer, 2019.

[Zub97] John C Zubeck. Implementing reuse with RAD tools' native objects. *Computer*, 30(10):60–65, 1997.