Separation of Powers in the Cloud: Where Applications and Users Become Peers *

David H. Lorenz[1,2] †
[1]Open University, Raanana 43107, Israel
[2]Technion–Israel Institute of Technology, Haifa 32000, Israel
dhlorenz@cs.technion.ac.il

Boaz Rosenan[1,3]
[3]University of Haifa, Mount Carmel, Haifa 31905, Israel
brosenan@gmail.com

Abstract

We challenge the widely accepted practice that web applications must be trusted with user data. We present an alternative model based on logic programming, where users and applications are equal peers in a shared cloud environment. User data is represented as a set of facts. The application is represented as a set of rules defining how user data is to be processed, but is not given direct access to the data. This way, end users remain the owners of their own data, and are able to determine who can see it and who can modify it. For concreteness, we define a data representation and query language, named CloudLog, for a new family of deductive databases, named NoDatalog. We add access control to the language for guaranteeing that the rules provided by the application cannot change the choices made by users. We demonstrate how business logic can be expressed in CloudLog, and discuss how an efficient CloudLog-based database can be implemented.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages; K.4.1 [*Computers and Society*]: Public Policy Issues—Use/abuse of power.

General Terms Languages, Security.

Keywords Logic programming (LP), Deductive Databases, Access Control, NoDatalog, NoSQL.

* This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 1440/14.

† Work done in part while visiting the Faculty of Computer Science, Technion—Israel Institute of Technology.

Onward!'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3688-8/15/10...$15.00
http://dx.doi.org/10.1145/2814228.2814237

1. Introduction

In today's Internet, we trust the websites and applications we use. We trust them with our credit card details, with personal information, and with private communications. But do they deserve our trust? What guarantees do we have that our private profile remains private? What guarantees us that no one else writes a blog-post and signs our name? Even if the site has a user-friendly privacy policy, how can we be sure the software behind the application actually implements it? Even with best intentions at heart, our data can be compromised by a failure in the application to check a condition somewhere, or by an erroneous logging of private information to a widely-accessible log file. Nevertheless, it seems like we have no choice but to trust these websites and these applications. After all, what would we do without our social networks, online shopping sites, or webmails?

In this paper we propose a simple idea that allows us to not have to trust applications with our data: we simply do not grant the application access to it. We retain ownership over our data, thus being able to modify it or remove it at will. We may grant other users permissions to read the data or modify it. Users with access to our data can repeat it to other audiences at their discretion, such as when one user forwards an email written by another user. However, without access to our data, applications do not have the power to betray our trust or compromise the integrity or the confidentiality of our data, be it an honest mistake or be it foul play.

But how can an application work without having access to the data it needs? To rectify this matter, the way applications are implemented needs to be changed. Currently, applications collect user data, store it, and process and display it (Fig. 1). Thus the company developing and operating the application has full control over user data. We present an alternative model that enforces separation of powers between application providers, end users, and data services. In this model, the traditional *application service provider* is split into an *application provider* and a trusted third-party *service provider*. Consequently, users and applications become peers, rather than subordinates (Fig. 2). Both the user and the

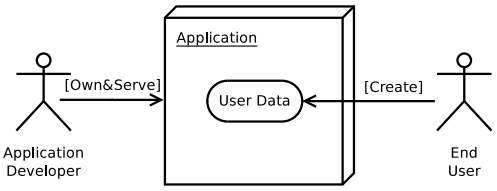

Figure 1: Traditionally, user data is owned by the application;

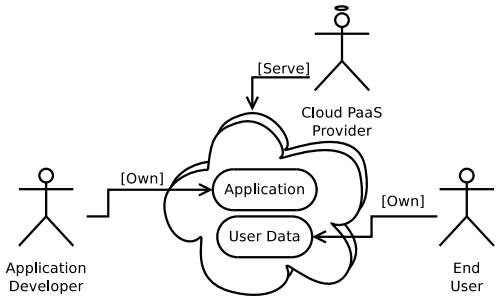

Figure 2: In CloudLog, users and developers are peers

application are merely clients of the service provider. Users publish *data* and applications publish *rules* telling the service provider what to do with the data. These rules cannot violate the user's will. They cannot make data intended for a certain group of users available to others (e.g., write sensitive information to a log file), and they cannot state anything on behalf of any user (e.g., falsify a post).

For this idea to work, several challenges must be met. First, we need to establish a *service provider* trusted by both end users and application providers. In this paper we assume that such a service can be made available by a trusted third-party cloud *Platform-as-a-Service (PaaS)* [29] provider. The service should provide a container for our application's server side, encapsulating both the storage and business logic layers of the application. This can be a shared environment where multiple applications use the same service.

Second, we need a way of implementing applications that respect separation of powers. The approach we take in this paper is based on *Logic Programming (LP)* [15, 25, 26], and involves the use of a deductive database [35] to store side by side user data and applications as facts and rules, respectively. Such a way of implementing applications is significantly different than how they are implemented today. Unfortunately, existing deductive databases fall short of supporting real life applications. In Sect. 2 we identify a new family of deductive databases, named NoDatalog, that supports the necessary expressiveness and scalability. We also introduce a Twitter-like application, named TwitLog, that is used as a motivating example throughout the paper.

Third, we need a concrete language to facilitate application development. We describe such a language, named CloudLog—a data point in the NoDatalog design space—in two steps. In Sect. 3 we define Basic CloudLog, a general data representation and query language for NoDatalog databases, allowing one to represent user data and application logic as axioms (facts and rules, respectively) in the database. Then, in Sect. 4 we move from axioms to statements. A statement is similar to an axiom (fact or rule), only that it is attributed to a set of writers, and is addressed to a set of readers. The cloud service (deductive database) needs to verify that the user adding or removing a statement from the database is a member of its writer set. When a user queries for information, the database takes into considerations only statements which that user is allowed to see. This makes it possible for a database implementation that knows nothing about the application or its users, to enforce effective access control.

Fourth, we need a sustainable software ecosystems [32]. To this end, Sect. 5 provides arguments as to why a trusted third-party cloud provider holding user data is a potential realistic solution in terms of a software business model, and sketches an efficient implementation for CloudLog.

2. NoDatalog

A deductive database [35] is a database that stores logic facts and rules and answers queries by means of logic deduction. The term *deductive database* is almost synonymous with Datalog [14, 22], just like the term *relational database* is almost synonymous with SQL [28]. Datalog can be seen as a dialect of Prolog [16] that was weakened on purpose to support efficient query of tabular data. The main two restrictions in Datalog, namely stratification and lack of compound terms (functions), render it Turing incomplete but still more powerful than (standard) SQL.[1] However, Datalog and deductive databases never became a mainstream alternative to SQL and relational databases.

During the last decade relational databases have been losing their stand as a *one size fits all* solution in the database world. A new family of databases emerged, commonly known as NoSQL [21], which are characterized by a simpler data model (typically an extension of the key-value store model), a looser consistency model, and significantly better availability and scalability [36]. NoSQL databases typically do not support sophisticated queries. Instead, an application that uses a NoSQL database is expected to store its data according to the expected query patterns. This may mean adding redundancy to the data via a process known as denormalization [4, 18, 19, 30, 33].

In a way, it seems that NoSQL databases have made nearly all design choices opposite to Datalog. Datalog

[1] In Datalog, recursive queries are supported to a certain level.

Database	Example	Logic Prog	Scalable
Relational	SQL	-	-
NoSQL	Cassandra	-	+
Deductive	Datalog	+	-
NoDatalog	CloudLog	+	+

Table 1: Design space

queries are based around conjunction of predicates, while NoSQL databases typically do not support conjunctive queries (joins). NoSQL gets away without joins by allowing compound objects (e.g., JSON documents) to be stored as a single entity, while Datalog does not support compound terms and restricts data elements to flat scalar values. This begs the question: *can a different kind of deductive databases, one that is not based on Datalog, be a better fit for the NoSQL era?*

We answer this question affirmatively, coining the term *NoDatalog* for the class of databases meeting this criterion. NoDatalog is to Datalog what NoSQL is to SQL (Table 1). NoDatalog databases are meant to serve the same purpose as traditional deductive databases, but they meet higher scale and flexibility requirements. Like NoSQL, the term NoDatalog itself does not describe a specific data model or query language.[2]

2.1 TwitLog in SQL and NoSQL

To better understand the need for NoDatalog, we describe here TwitLog, a simple Twitter-like Web application. Like Twitter, TwitLog is based on the notion of users *tweeting* (posting messages out to the world), and other users *following* them (requesting to get tweets into their *time-line*).

Formally, let $Users$ be a set of users, and let $Tweets$ be a set of all possible tweets. We denote by $\tau \subseteq Users \times Tweets$ the set of all tweets that have been performed on the system, and by $\phi \subseteq Users \times Users$ the set of all following relations in the system. The time-line of user $u_1 \in Users$ consists of every tweet t made by user u_2 who u_1 follows:

$$\mathcal{T}_{u_1} = \{(u_2, t) \mid (u_1, u_2) \in \phi \wedge (u_2, t) \in \tau\} \qquad (1)$$

To keep the example simple we ignore the *time* aspect of a time-line.[3]

Fig. 3 shows a possible entity diagram for using a relational database to store the state of TwitLog. It consists of three tables, `User`, `Tweet`, and `Following`, respectively representing $Users$, τ, and ϕ. Computing $\mathcal{T}_u$ is done using the SQL query in Fig. 4.

While this query is simple and straightforward, a system designed this way around a relational database will have

[2] CloudLog is an example of a language in the NoDatalog design space.

[3] A more complete model should include a time-stamp in each tweet in τ, which is then reflected in each element of $\mathcal{T}_u$. When presented to the user the time-line is sorted by time-stamp in descending order.

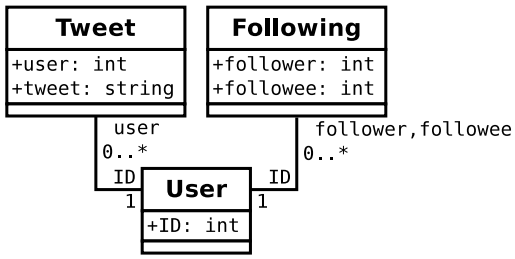

Figure 3: Entity diagram for the simple TwitLog data model

```
SELECT user, tweet
FROM Tweet, User, Following
WHERE user = ID
AND followee = ID
AND follower = u1
```

Figure 4: SQL query for the TwitLog time-line for u_1

trouble keeping up with growing demand. The main problem with this implementation is that when data is too big to fit in one machine, a *sharding* of the data is needed, i.e., τ and ϕ need to be split across different machines. The query in Fig. 4 requires first finding the followees of user u_1, a query of elements of the form $(u_1, u) \in \phi$. A reasonable implementation will store all the results of such a query on one machine. Then, for each such u, we need to find $(u, t) \in \tau$. Even if all tweets of single users reside on the same machine, looking for all tweets of several (possibly many) users require accessing multiple machines at once. This may result in unacceptable latency.

Evans [18] proposes a scalable design for a similar problem. His implementation, named Twissandra,[4] builds a Twitter-like application over the Cassandra [4, 19, 27] NoSQL database. Like many other NoSQL databases, Cassandra does not support *joins*, like the one we use in the query in Fig. 4. Instead, the application maintains an extra table, holding for each user a pre-calculated time-line (a de-normalized representation). When a user tweets, Twissandra looks up all the followers of that user, and adds the new tweet to each of their time-lines (Fig. 5). A similar loop exists in the function that removes a tweet. Unlike Twitter, however, Twissandra does not show in u_1's time-line tweets made by user u_2 before u_1 started following u_2. If we wish to change this to show all tweets made by all followed users (regardless of the time the followee was added), we need to add similar lines in the functions handling adding and removing followees.

De-normalizing the data in this way is necessary to support low latency requests on large scale data. However, from a software engineering point of view, such data redundancy is best avoided. Any bug in this mechanism will result in

[4] `https://github.com/twissandra/twissandra`

```
# Get the user's followers, and insert the tweet into all of
     their streams
futures = []
follower_usernames = [username] +
    get_follower_usernames(username)
for follower_username in follower_usernames:
 futures.append(session.execute_async(
  timeline_query, (follower_username, now, tweet_id)))
for future in futures:
 future.result()
```

Figure 5: A part of the `save_tweet()` function implementation in Twissandra

```
timeline(A, B, T) :-
  follows(A, B),
  tweet(B, T).
```

Figure 6: Prolog definition of the TwitLog time-line

inconsistent data. For example, Twissandra uses eventually-consistent [34] updates and queries over Cassandra. As a result, if a tweet has been made close to the time in which one user started following another, there is a chance the two events will miss each other. That is, it is possible that when `get_follower_usernames` in Fig. 5 is executed for adding a new tweet, the follower is not yet visible to the query. Symmetrically, when executing the corresponding code in the function adding the follower, it is possible that the tweet is not yet visible to that function. In such a case, none of them will add the tweet to the follower's time-line.[5]

2.2 TwitLog in Datalog and Prolog

As an alternative to using a relational database or NoSQL, let us consider now the use of LP languages to implement TwitLog. We will consider both Datalog and Prolog for this example, ignoring, for the sake of the example, the fact that Prolog assumes all facts and rules fit in memory on a single machine. Fig. 6 shows a simple Horn clause that implements TwitLog. This clause is valid in both Datalog and Prolog. It defines `timeline(A, B, T)`, a predicate representing T_u, in a clean, declarative manner, assuming the predicates `follows(A, B)` and `tweet(U, T)` represent ϕ and τ, respectively.

Although the same clause defines T_u correctly in both languages, in Prolog this definition is much more powerful. This is because, unlike Datalog, values in Prolog (terms) can be compound. This, together with the ability to add new features by only adding new clauses, i.e., without changing any existing code, makes the implementation much more extensible.

[5] In the existing Twissandra code, this bug does not exist, as the application does not show tweets made before the following relationship was established. However, if we decide to implement it to comply with Eq. 1 (and with the behavior of Twitter), the bug will present itself.

```
timeline(A, B, following(A)) :-
  follows(B, A).
timeline(A, C, text(T)) :-
  tweet(C, text(T)),
  replies(T, B),
  follows(A, B).
```

Figure 7: Datalog query for the TwitLog time-line

Imagine we wish to extend TwitLog to support, not only textual tweets, but also other kinds of tweets. For example, if TwitLog was made for the scientific community, it could allow its users to tweet calls for papers, or citations for interesting papers they come across. By not assuming anything on the structure of `T`, the code in Fig. 6 can support any compound term in place of a tweet. Therefore, the fact:

```
tweet(alice, text('CloudLog Rocks')).
```

contributes a textual tweet, while the fact:

```
tweet(bob, cfp('Onward!15', date(apr,2,2015))).
```

contributes a call for papers (CFPs). The facts:

```
follows(charlie, alice).
follows(charlie, bob).
```

will complete this program so that the query:

```
?- timeline(charlie, U, T).
```

will have two results, Alice's textual tweet and Bob's CFP.

While this example is very simple, it is enough to demonstrate the relative weakness of both Datalog and SQL. In both languages, `T` (or its SQL equivalent `Tweet.tweet`) must have a concrete scalar type, and therefore none of them supports this kind of polymorphism.

Now let us kick the story up a notch. Imagine we would like to add to the time-lines of users information other than tweets made by users they follow, such as replies to these tweets or a notice on who is following them.

The clauses in Fig. 7 add these features. The first clause contributes a time-line entry for each follower, and the second clause contributes a time-line entry for each reply. TwitLog, like Twitter, does not use a special header to identify replies. Any (textual) tweet can be considered a reply to any user, and potentially more than one user, if the text contains "`@`" followed by a user-name. The predicate `replies(T,U)` used in Fig. 7 succeeds for every occurrence of "`@`" in the text, e.g., `replies('@alice likes @bob',U)` will emit two results: `U=alice` and `U=bob`.

Note that our support for replies is overly inefficient. The first goal in the second clause (Fig. 7) looks for all tweets made by all users. This is because we do not have *a priori* knowledge of where to look for replies. This begs another

question: *can we define a language that will feature Prolog's extensibility, but at the same time allow efficient queries for large data-sets?* Our answer: CloudLog.

2.3 CloudLog: A Language for NoDatalog Databases

Deriving conclusions from axioms can work either top-down or bottom-up. To preserve the best of both worlds, CloudLog, a language we introduce in this paper, supports a combination of the two.

A *top-down evaluation* starts with a goal A to be proven or disproven, and identifies facts and rules that can prove it. If a rule $A \leftarrow B$ is found, the top-down evaluation will continue by trying to prove B, and so on. A *bottom-up evaluation* does not have to start with a goal in mind. Starting with an initial set of axioms P_0, it tries to find facts and rules that interact, such as $A, (A \rightarrow B) \in P_0$, and for these it adds B to the set: $P_1 = P_0 \cup \{B\}$, and so on, until reaching a steady state, i.e., $\forall A, (A \rightarrow B) \in P_n : B \in P_n$.

On the one hand, bottom-up evaluation will only terminate for sets of axioms from which a finite set of conclusions can be derived. If this set is infinite, bottom-up evaluation will go on forever, producing more and more conclusions. This makes top-down evaluation a more powerful approach and the one used in Prolog. On the other hand, top-down evaluation can only start at the presence of a query term, and therefore inhibits de-normalization of the data. This makes bottom-up evaluation scale better.

Unlike other approaches that implicitly combine bottom-up and top-down evaluation [3, Ch. 13], CloudLog leaves it to the user to explicitly decide which evaluation is done in which direction. To assure termination, bottom-up rules in CloudLog are not allowed to be recursive. Recursion is supported for top-down rules, thus rendering the language Turing complete.

3. Basic CloudLog

Next we describe CloudLog, a LP language intended both as a data representation language and as a query language for NoDatalog deductive databases. We split our description of CloudLog into two dialects. Basic CloudLog is discuss in this section. Secure CloudLog is discussed in Sect. 4.

3.1 Abstract Syntax

The abstract syntax for Basic CloudLog is shown in Fig. 8. A database supporting CloudLog stores a set of axioms. An axiom can be either one of: a *fact* (Rule (8.2)), a bottom-up *rule* (Rule (8.3)), or a top-down *clause* (Rule (8.4)).

Bottom-up rules contain a goal $\{Goal\}$, named a *guard*, to be evaluated during bottom-up propagation. In most cases, this guard will be the trivial goal $\top$. In such cases instead of writing

$$Atom\{\top\} \rightarrow Axiom$$

we omit the guard and write the rule as

$$Atom \rightarrow Axiom$$

$$\begin{array}{rcll}
Axiom & ::= & Atom & (8.2)\\
 & | & Atom\,\{Goal\} \rightarrow Axiom & (8.3)\\
 & | & Atom \leftarrow Goal & (8.4)\\
Goal & ::= & Atom, Goal & (8.5)\\
 & | & \neg Goal & (8.6)\\
 & | & \top & (8.7)\\
Atom & ::= & name\,(Term^*) & (8.8)\\
Term & ::= & number & (8.9)\\
 & | & string & (8.10)\\
 & | & Variable & (8.11)\\
 & | & name\,(Term^*) & (8.12)\\
Variable & ::= & name & (8.13)
\end{array}$$

Figure 8: Basic CloudLog: abstract syntax

$$\begin{array}{l}
\texttt{query}\,(DB, Q)\,\texttt{:-}\\
\quad \texttt{is_atom}\,(Q)\,,\\
\quad \texttt{db_find}\,(DB, Q \leftarrow G)\,,\\
\quad \texttt{query}\,(DB, G)\,.
\end{array} \tag{9.14}$$

$$\begin{array}{l}
\texttt{query}\,(DB, (G_1, G_2))\,\texttt{:-}\\
\quad \texttt{query}\,(DB, G_1)\,,\\
\quad \texttt{query}\,(DB, G_2)\,.
\end{array} \tag{9.15}$$

$$\begin{array}{l}
\texttt{query}\,(DB, \neg G)\,\texttt{:-}\\
\quad \neg\texttt{query}\,(DB, G)\,.
\end{array} \tag{9.16}$$

$$\begin{array}{l}
\texttt{query}\,(DB, \top)\,\texttt{:-}\\
\quad \texttt{true}.
\end{array} \tag{9.17}$$

Figure 9: Pseudo-Prolog implementation of `query`

Basic CloudLog can be seen as consisting of two languages: (i) facts and (bottom-up) rules that interact to create (ii) a set of (top-down) clauses, many of which will be trivial (i.e., of the form $Atom \leftarrow \top$). The set of clauses is semantically similar to Prolog, without cuts and side effects.

3.2 Semantics

A CloudLog database supports the operations *add*, *remove*, and *query*. Fig 9 shows the $\texttt{query}\,(DB, Q)$ predicate, described in Prolog-like pseudo-code. Clause (9.14) matches atomic goals, and finds clauses of the form $Q \leftarrow G$ in the database, where Q matches the query. For each such rule, this clause attempts to query the matching goal G, by making a recursive call to itself. The predicate $\texttt{db\_find}\,(DB, X)$ used in Clause (9.14) looks up all axioms in the database unifiable with X, and backtracks through all of them, unifying each of them with X. Clauses (9.15), (9.16), and (9.17)

```
add(DB, A{G} → B) :-
  db_store(DB, A{G} → B),
  db_find(DB, A),
  query(DB, G),
  add(DB, B).
```
(10.18)

```
add(DB, A) :-
  is_fact(A),
  db_store(DB, A),
  db_find(DB, A{G} → B),
  query(DB, G),
  add(DB, B).
```
(10.19)

Figure 10: Pseudo-Prolog implementation of *add*

```
remove(DB, A{G} → B) :-
  db_find(DB, A),
  query(DB, G),
  remove(DB, B).
  db_delete(DB, A{G} → B),
```
(11.20)

```
remove(DB, A) :-
  is_fact(A),
  db_find(DB, A{G} → B),
  query(DB, G),
  remove(DB, B).
  db_delete(DB, A),
```
(11.21)

Figure 11: Pseudo-Prolog implementation of `remove`

handle conjunction, negation and the trivial goal $\top$, respectively, similarly to their interpretation in Prolog.

Fig. 10 shows the implementation for the *add* operation. Again, we use the same Prolog-like language to depict the algorithm. Please note that, unlike CloudLog itself, this Prolog code is imperative. The variable DB represents a *reference* to the database, and not its content. The content is mutated by the `db_store`(DB, X) predicate under the same reference.

The predicate `add`(DB, X) is divided into two clauses: Clause (10.18) matches rules, and Clause (10.19) matches facts. In Clause (10.18) the rule is first added to the database by itself, and then the clause backtracks through all the facts in the database that may trigger this rule. For each such fact, the associated guard is evaluated (using `query`(DB, Q)), and for each result, the product of the rule is added recursively. Clause (10.19) is similar, only that it queries for rules that interact with the fact being added.

The *remove* operation is defined symmetrically (Fig. 11). It uses `db_delete`(DB, X) to remove axioms from the database.

Note that the the meaning of guard G in a bottom-up rule $A\{G\} \rightarrow B$ must not change once a rule that uses it enters the database. If it changes (e.g., if a clause providing it more solutions is added to the database) the results given by G during an invocation of `remove`(DB, X) will not match those received when `add`(DB, X) was invoked. As a result, "zombie" axioms can remain in the database. To avoid such a situation, we need a way to lock such goals, and not allow modification of clauses contributing to them once they have been used in such a way in a bottom-up rule. We leave the description of this mechanism for future work. For now, we will limit our use of this goal to the necessary minimum.

```
follows(A, B) →
  tweet(B, T) →
    timeline(A, B, T) ← ⊤
```
(12.22)

```
follows(A, B) →
  timeline(B, A, following(B)) ← ⊤
```
(12.23)

```
tweet(C, text(T)){replies(T, B)} →
  followed_by(B, A) →
    timeline(A, C, text(T)) ← ⊤
```
(12.24)

```
follows(A, B) →
  followed_by(B, A)
```
(12.25)

Figure 12: TwitLog implementation using Basic CloudLog

3.3 TwitLog in CloudLog

The implementation of TwitLog in Prolog (Sect. 2.2) can be translated to Basic CloudLog using either top-down or bottom-up rules. We choose the bottom-up rules, shown in Fig. 12, in order to shorten the query latency.

With this implementation, when Alice starts following Bob, she adds the fact `follows(alice, bob)` to the database. This fact interacts with Rules (12.22), (12.23), and (12.25) producing (26), (27), and (28), respectively:

```
tweet(bob, T) →
  timeline(alice, bob, T) ← ⊤
```
(26)

```
timeline(bob, alice, following(bob)) ← ⊤
```
(27)

```
followed_by(bob, alice)
```
(28)

Clause (27) contribute an entry to Bob's time-line, indicating that Alice is following him. When Bob tweets "Hi There," he adds the fact:

```
tweet(bob, text("Hi There"))
```

which interacts with Rule (26) to produce:

```
timeline(alice, bob, text("Hi There")) ← ⊤
```

causing Bob's tweet to appear in Alice's time-line. It also attempts to interact with Rule (12.24), but is stopped by the guard `replies` ("`Hi There`", B), since "Hi There" is not a reply. If Charlie replies:

$$\texttt{tweet}\,(\texttt{charlie}, \texttt{text}\,(\text{“}\texttt{@bob Hi}\text{”}))$$

then Rule (12.24) will yield:

$$\texttt{followed\_by}\,(B, A) \rightarrow \\ \quad \texttt{timeline}\,(A, \texttt{charlie}, \texttt{text}\,(\text{“}\texttt{@bob Hi}\text{”})) \leftarrow \top \tag{29}$$

Rule (29) then interacts with Fact (28) to produce:

$$\texttt{timeline}\,(\texttt{alice}, \texttt{charlie}, \texttt{text}\,(\text{“}\texttt{@bob Hi}\text{”})) \leftarrow \top$$

adding Charlie's reply to Alice's time-line.

The effect we get from using bottom-up rules is similar to what was achieved by Twissandra's de-normalization functions. However, here the implementation of *add* and *remove* guarantee that no matter what updates are made to the set of axioms in the database, and in which order they are made, the time-line of each user will always reflect the current state of the tweets and following relations. Moreover, by using a guarded bottom-up rule for implementing replies, we resolve the efficiency problem in the Prolog implementation (Fig. 7). Now we examine each tweet for `@`*username* patterns as it arrives, one invocation of `replies` for each tweet, and do not need to scan all tweets for each query.

4. Secure CloudLog

The rules in Fig. 12 assume that only user A can cause axioms such as $\texttt{follows}\,(A, B)$ and $\texttt{tweet}\,(A, T)$ to be added to the database. In other words, it assumes that someone outside the database decides what axioms a user may or may not add or remove.

Symmetrically, if for example we would like to keep *following* relationships private (e.g., to disallow one's wife to know her husband is following her best friend), we need to assume that only user u can query his or her own time-line, i.e., perform the query $\texttt{timeline}\,(u, B, T)$.

Access control is an important aspect of any application. Traditionally, access control for user data is defined by the application. However, in order to protect user data from the application we need a different approach. Secure CloudLog allows a third-party that knows nothing about the application to enforce access control in a way that satisfies the expectations of both end users and application providers.

4.1 Axioms vs. Statements

Basic CloudLog has one fundamental problem, making it hard to secure. It deals with axioms. An axiom is something we take to be true. Consider the sentence: "it is going to rain on Wednesday." Accepting this as an *axiom* means it would certainly going to rain on Wednesday. However, this sentence makes much more sense as an opinion or a speculation made by someone. We consider "it is going to rain on Wednesday" as a *statement*. As in its meaning in the English language, a statement is something stated by someone. This could be an opinion, a speculation, or even a lie stated by an adversary in order to gain something. We can derive an axiom from a statement without giving it more credibility than it deserves by attributing it to a speaker. For example, the sentence: "it is going to rain on Wednesday, said the weatherman" can be regarded as an axiom, regardless of the actual chance of rain.

On the flip side, axioms do not specify an audience. Statements, however, are typically addressed to a person, a group of people, or sometimes the whole world. An axiom can capture the audience along with the statement: "Danny told Sarah he did not complete the assignment."

Statements can therefore help simplify access control in a logic system. Access control in an information system serves two purposes:[6]

- *Confidentiality*: protecting writers against unauthorized readers;
- *Integrity*: protecting readers against unauthorized writers.

Through the rest of this section we describe how these properties are achieved in Secure CloudLog.

$$\texttt{s\_add}\,(U, DB, S\,[W \Rightarrow R])\,\texttt{:-} \\ \quad U \in W, \\ \quad \texttt{add}\,(DB, S\,[W \Rightarrow R])\,. \tag{13.30}$$

$$\texttt{s\_remove}\,(U, DB, S\,[W \Rightarrow R])\,\texttt{:-} \\ \quad U \in W, \\ \quad \texttt{remove}\,(DB, S\,[W \Rightarrow R])\,. \tag{13.31}$$

$$\texttt{s\_query}\,(U, DB, Q\,[W \Rightarrow R])\,\texttt{:-} \\ \quad U \in R, \\ \quad \texttt{query}\,(DB, Q\,[W \Rightarrow R])\,. \tag{13.32}$$

Figure 13: Secure database operations

4.2 Semantics

Secure CloudLog only allows users to write axioms of the form $S\,[W \Rightarrow R]$, where S is a statement, and W and R are set expressions we call the *reader* and *writer sets*, respectively. As shown in Fig. 13, Clauses (13.30) and (13.31), the secure versions of the add and remove operations require that the user adding or removing axiom $S\,[W \Rightarrow R]$ to or from the database be a member of W. This assures that a statement is always attributed to the correct person or group. For example, if the statement $\texttt{will\_rain\_on}\,(\texttt{wed})$ is made by

[6] Threats related to *availability* and *non-repudiation* are beyond the scope of this paper.

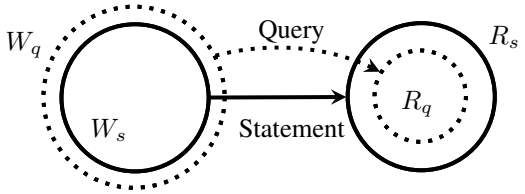

Figure 14: A statement considered by a query

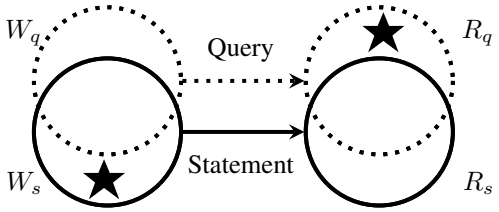

Figure 15: A statement not considered by a query

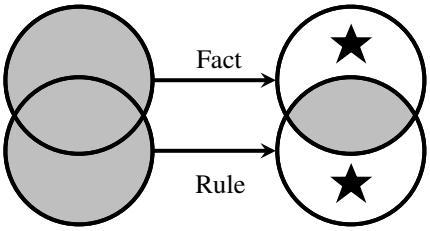

Figure 16: Unchecked rule

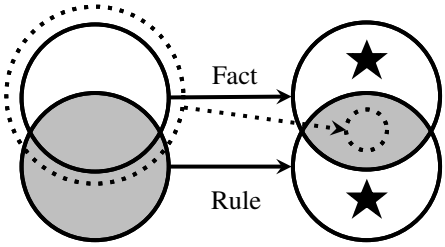

Figure 17: Checked rule

the weatherman, the axiom

$$\texttt{will\_rain\_on}\,(\texttt{wed})\,[\texttt{weathermen} \Rightarrow *]$$

would be added to the database. However, if a student wants to influence our plans by making us think it would rain on Wednesday, the student can add the axiom:

$$\texttt{will\_rain\_on}\,(\texttt{wed})\,[\texttt{weathermen} \cup \texttt{our\_students} \Rightarrow *]$$

When we read such an axiom we may infer that the speculation came from either a weatherman or one of our students. Therefore, when we query, we will limit the scope of trusted writers to only `weathermen` by making the following query:

$$\texttt{will\_rain\_on}\,(X)\,[\texttt{weathermen} \Rightarrow \texttt{us}]$$

(assuming `us` is a set of users containing ourselves). We use a set of writers (and not a single writer) to allow a statement made by one user be modified or removed by others (modification is done by removing a statement and adding a different one). If a statement is signed by a group (i.e., W contains more than one user), anyone in that group can remove that statement and replace it with a similar one, signing on behalf of the same group. Therefore, the selection of the writer set is equivalent to assigning write permissions over the statement.

Read permissions over statements are set through the readers set. For example, if Alice told Bob she likes him, and does not want Eve to know, she would add the axiom $\texttt{likes}\,(\texttt{alice}, \texttt{bob})\,[\langle\texttt{alice}\rangle \Rightarrow \langle\texttt{bob}\rangle]$ to the database ($\langle U\rangle$ represents a set of users containing only user U). This way, Bob knows the message is authentic (came from Alice), and Alice knows only Bob is able to see it. To make sure of the latter, Secure CloudLog only allows queries of the form $Q\,[W \Rightarrow R]$, where the user making the query must be a member of R (Clause (13.32)).

To complete this access control system, a query of the form $Q\,[W_q \Rightarrow R_q]$ will only consider axioms $S\,[W_s \Rightarrow R_s]$ such that $W_s \subseteq W_q$ and $R_q \subseteq R_s$, as depicted in Fig. 14. This is to protect against two possible adversaries, displayed as stars in Fig 15: an unauthorized writer, who appears in the writer set of the statement, but not in the set of trusted writers stated by the query; and an unauthorized reader, who appears in the reader set of the query, but not in that of the statement.

A statement can represent either a fact, a (bottom-up) rule or a (top-down) clause. As with axioms in Base CloudLog, a statement representing a bottom-up rule interacts with statements representing facts, to create new statements. The writer and reader sets associated with the new statement must be set in such a way that will not allow an adversary to violate the confidentiality expectations of either the fact or the rule, as well as the integrity of queries made on the derived statement.

Secure CloudLog provides two kinds of bottom-up rules: checked and unchecked. An *unchecked rule*, as depicted in Fig. 16, does not make any assumptions on the facts it interacts with. It therefore does not take responsibility for them. The writer set of a child statement will therefore be a union of the writer sets of both the fact and the rule, since both writers contributed to its content, and both of them can remove it by removing either the rule or the fact. Symmetrically, the reader set of the derived statement is an intersection of the reader sets of the rule and the fact. This is to protect against two potential adversaries, one trying to

$$
\begin{array}{rcll}
\mathit{Axiom} & ::= & \mathit{Stmt}\,[\mathit{Set} \Rightarrow \mathit{Set}] & (18.33)\\
\mathit{Stmt} & ::= & \mathit{Atom} & (18.34)\\
 & | & \mathit{Atom}\,\{\mathit{SecGoal}\} \to \mathit{Stmt} & (18.35)\\
 & | & \mathit{Atom}\,[\mathit{Set} \Rightarrow \mathit{Set}]\ \{\mathit{SecGoal}\} \to \mathit{Stmt} & (18.36)\\
 & | & \mathit{Clause} & (18.37)\\
\mathit{SecGoal} & ::= & \mathit{Goal}\,[\mathit{Set} \Rightarrow \mathit{Set}] & (18.38)\\
\mathit{Set} & ::= & \mathit{Group} & (18.39)\\
 & | & \langle \mathit{User} \rangle & (18.40)\\
 & | & \mathit{Set} \cup \mathit{Set} & (18.41)\\
 & | & \mathit{Set} \cap \mathit{Set} & (18.42)\\
 & | & * \quad | \quad \emptyset & (18.43)\\
\mathit{Group} & ::= & \mathit{name} :: \mathit{Term} & (18.44)\\
 & | & \mathit{name} & (18.45)\\
 & | & \texttt{admin}\,(\mathit{Group}) & (18.46)\\
 & | & \texttt{root} & (18.47)\\
\mathit{User} & ::= & \mathit{name} & (18.48)
\end{array}
$$

Figure 18: Secure CloudLog: abstract syntax

$$
\texttt{tweet}\,(T)\,[\langle U \rangle \Rightarrow \emptyset] \to \texttt{tweet}\,(U,T)\,[\texttt{twitlog} \Rightarrow *] \quad (19.49)
$$

$$
\texttt{follows}\,(B)\,[\langle A \rangle \Rightarrow \langle A \rangle \cup \langle B \rangle] \to \texttt{followed\_by}\,(B,A)\,[\texttt{twitlog} \Rightarrow *] \quad (19.50)
$$

$$
\texttt{follows}\,(B)\,[\langle A \rangle \Rightarrow \langle A \rangle \cup \langle B \rangle] \to \texttt{tweet}\,(B,T)\,[\texttt{twitlog} \Rightarrow \emptyset] \to \texttt{timeline}\,(A,B,T) \leftarrow \top\,[\texttt{twitlog} \Rightarrow *] \quad (19.51)
$$

$$
\texttt{follows}\,(B)\,[\langle A \rangle \Rightarrow \emptyset] \to \texttt{timeline}\,(A,B,\texttt{following}\,(A)) \leftarrow \top\,[\texttt{twitlog} \Rightarrow *] \quad (19.52)
$$

$$
\texttt{tweet}\,(\texttt{text}\,(T))\,[\langle C \rangle \Rightarrow \emptyset]\ \{\texttt{replies}\,(T,B)\,[* \Rightarrow *]\} \to \texttt{followed\_by}\,(B,A)\,[\texttt{twitlog} \Rightarrow \emptyset] \to \texttt{timeline}\,(A,C,\texttt{text}\,(T)) \leftarrow \top\,[\texttt{twitlog} \Rightarrow *] \quad (19.53)
$$

Figure 19: TwitLog implementation using Secure CloudLog

spy over the rule by creating a matching fact, and the other, trying to spy over the fact by creating a matching rule.

A *checked rule* is a bottom-up rule that makes assumptions on matching facts, and in return, takes ownership over the derived statement. Fig. 17 shows how such a rule works. The dotted lines represent the assumptions being made. It specifies a set of trusted writers, and a minimum set of mandatory readers for the fact. The latter is required to support fairness, e.g., to not allow user A to follow user B without letting user B know about it. The rule is only applied to facts that meet these requirements. The derived statement will be signed by the rule's writer set. As with unchecked rules, The derived statement's reader set will be the intersection of both the rule and the fact.

4.3 Abstract Syntax

The grammar rules in Fig. 8 supplemented by the ones in Fig. 18 define the abstract syntax for the Secure CloudLog language. Rule (18.33) defines statement axioms of the form $S\,[W \Rightarrow R]$, where S is a statement (Stmt), W is the writer set, and R is the reader set. The abstract syntax of statements is similar to that of axioms, only with two kinds of bottom-up rules: checked rules of the form $S_1\,[R_s \Rightarrow W_s]\,\{SG\} \to S_2$ (Rule (18.36)), and unchecked rules of the form $S_1\,\{SG\} \to S_2$ (Rule (18.35)). As guard, instead of using a simple goal they specify a *secure goal* ($\mathit{SecGoal}$, Rule (18.38)), which specifies its own writer and reader sets. Here too we usually omit the guard in a bottom-up rule, and refer to it as $\top\,[* \Rightarrow *]$.

User sets can be constructed from a named user group (Rule (18.39)), a singleton set of the form $\langle U \rangle$ consisting of one user U (Rule (18.40)), a union of sets (Rule (18.41)), an intersection of sets (Rule (18.42)) or one of two special sets: The set of all users $*$, and the empty set $\emptyset$ (Rule (18.43)).

A named user group can be either a term scoped under a group domain (Rule (18.44)), a group domain without a term scoped user it (Rule (18.45)), an admin group (Rule (18.46)), or the root group (Rule (18.47)). The last two kinds are used for assigning users into groups (Sect. 4.5).

4.4 Secure TwitLog

We now return to our implementation of TwitLog to demonstrate the use of Secure CloudLog for guaranteeing the privacy of our users and the integrity of the tweets they are seeing.

Fig. 19 shows an implementation of TwitLog in Secure CloudLog. Rule (19.49) is a translation of $\texttt{tweet}\,(T)$ statements made by user U into $\texttt{tweet}\,(U,T)$ statements made by the TwitLog system itself. This is a checked rule, checking that the tweet was "signed" by a user. The rule does not pose any restrictions on the audience of the tweet (the reader set), and requires the bare minimum, the empty set $\emptyset$. The purpose of this rule is to make tweets searchable according

to their author.[7] For example, if Alice tweeted "CloudLog Rocks," she would create the fact:

$$\texttt{tweet}\,(\texttt{text}\,(\text{“}\texttt{CloudLog Rocks}\text{”}))\,[\langle\texttt{alice}\rangle \Rightarrow *]$$

This axiom matches the conditions posed by Rule (19.49), and therefore the fact:

$$\texttt{tweet}\,(\texttt{alice}, \texttt{text}\,(\text{“}\texttt{CloudLog Rocks}\text{”}))\,[\texttt{twitlog} \Rightarrow *]$$

will be created. The TwitLog application signs (by its own user group) the resulting fact, to inform others (who trust the TwitLog application) that the user U indeed twitted the tweet T. If, for example, Eve wanted to forge a tweet by Alice, she would need to sign it by a group she belongs to, and `twitlog` is not such a group.

Rules (19.51) and (19.52) are similar to Rules (12.22) and (12.23) respectively, only with author and recipient annotations. When Bob starts following Alice he adds a fact of the form

$$\texttt{follows}\,(\texttt{alice})\,[\langle\texttt{bob}\rangle \Rightarrow \langle\texttt{bob}\rangle \cup \langle\texttt{alice}\rangle]$$

Rule (19.51) asserts that Bob does not hide from Alice the fact he is following her. Then it creates a rule of the form:

$$\texttt{tweet}\,(\texttt{alice}, T)\,[\texttt{twitlog} \Rightarrow \emptyset] \rightarrow \texttt{timeline}\,(\texttt{bob}, \texttt{alice}, T) \leftarrow \top\,[\texttt{twitlog} \Rightarrow \langle\texttt{bob}\rangle \cup \langle\texttt{alice}\rangle]$$

Note that since the following relation is known only to Alice and Bob, the rule that results from it is also known only to them. This rule interacts with tweets made by Alice (denormalized and signed by the application) to produce time-line entries of the form:

$$\texttt{timeline}\,(\texttt{bob}, \texttt{alice}, \texttt{text}\,(\text{“}\texttt{CloudLog Rocks}\text{”})) \leftarrow \top\,[\texttt{twitlog} \Rightarrow \langle\texttt{bob}\rangle \cup \langle\texttt{alice}\rangle]$$

Now Bob can query his time-line using the query:

$$\texttt{timeline}\,(\texttt{bob}, U, T)\,[\texttt{twitlog} \Rightarrow \langle\texttt{bob}\rangle]$$

This query will provide all entries signed by the application that are addressed at (and therefore, visible by) Bob.

4.5 Assigning Users into Groups

The secure operations described in Fig. 13 assume there is a way to determine if a given user is a member of a given user set. While the membership predicate for most kinds of sets given in Fig. 18 is straightforward, it is harder to determine which users belong to a named user group. The main challenge here is to answer this question without having to have a global system administrator assigning users to groups. So far we managed to require only simple, generic access control rules, that know nothing of our application. We therefore would like this aspect to behave the same way.

To solve this problem we use group domains. A *group domain* is a unique identifier, registered to a specific user. Registering a domain is done by the $\texttt{register\_domain}\,(U, D)$ operation, defined as:

$$\texttt{register\_domain}\,(DB, U, D) : \neg\texttt{db\_find}\,(DB, \texttt{domain}\,(D, U'))\,, \texttt{db\_store}\,(DB, \texttt{domain}\,(D, U))\,.$$

This first checks that the domain is not taken, and if not, it registers it to the user. Then, the rule:

$$\texttt{domain}\,(D, U) \rightarrow \texttt{member\_of}\,(U, \texttt{admin}\,(D :: X)) \leftarrow \top\,[\texttt{root} \Rightarrow *]$$

makes U a member of the admin group for all groups $D :: X$ in the domain registered by U, and similarly,

$$\texttt{domain}\,(D, U) \rightarrow \texttt{member\_of}\,(U, \texttt{admin}\,(D)) \leftarrow \top\,[\texttt{root} \Rightarrow *]$$

makes U an admin of a group named after the domain itself. Both membership statements are signed by the `root` group. This is the only use of this group in our model. The predicate $\texttt{member\_of}\,(U, G)$ is used to determine if a user is a member of a group. It is queried by the database to determine read and write permissions (depicted in Fig. 13 as $U \in W$ in `s_add` and `s_remove` and as $U \in R$ in `s_query`). When queried, the database uses `root` as the trusted writer set:

$$\texttt{member\_of}\,(U, G)\,[\texttt{root} \Rightarrow \langle U\rangle]$$

The use of $\langle U\rangle$ as the reader set of this query is designed to addresses a possible attack where a user tries to write statements on behalf of different groups, just to know if he or she is a member of that group.

If a user U is an administrator of group G, U can add members to G. For example, if Alice reserved the domain `foo`, she becomes the admin of all groups of the form $\texttt{foo} :: X$. Now she can add Bob to the group $\texttt{foo} :: \texttt{bar}$:

$$\texttt{group\_member}\,(\texttt{bob})\,[\texttt{admin}\,(\texttt{foo} :: \texttt{bar}) \Rightarrow *]$$

This axiom gets its meaning from the rule:

$$\texttt{group\_member}\,(U)\,[\texttt{admin}\,(G) \Rightarrow \emptyset] \rightarrow \texttt{member\_of}\,(U, G) \leftarrow \top\,[\texttt{root} \Rightarrow *]$$

5. Discussion

Lastly, we discuss the potential viability of our model.

5.1 Who Can You Trust These Days?

Our approach allows users to not have to trust the applications they use, but instead users need to trust a cloud

[7] As in Prolog, CloudLog facts are expected to be indexed by their first argument.

provider to work on their behalf. How can we assure the cloud provider is worthy of this trust?

The issue here is not trust *per se*. Already today, we can use services like Dropbox or OneCloud without fear of the vendor selling out our content. What gives us this confidence is a clear business model by which the user is a customer, and the cloud provider is a vendor. In such a business model, the service provider makes its earnings from users like us, and betraying our trust will cause them to lose their source of revenue.

The approach described in this paper does not free users from the need for trust, but rather enables separation of powers. Today's applications play two roles that used to be separated. On the one hand, they develop the software, and on the other hand, they serve it, a job which includes retention of user data. Our approach separates these two roles back into two parties, namely, the application provider and the cloud service provider. Assuming these two parties are different business entities, our approach changes the software business model, and with it, the trust model. In the new business model, both the application provider and the end user become the cloud provider's customers. In this business model the cloud provider is being payed to maintain the confidentiality and integrity of the user's data on the one hand, and the profitability of the application, on the other hand.

As customers, users may or may not have to pay for the service. We can think of models where the application providers still pay the bill for users using their application. However, in cases where application providers are the only paying customers, they may have the upper hand when the cloud service provider needs to make decisions that balance between user privacy and application profitability.

One direction that may resolve this tension all together is to hide the user data from the cloud provider as well. We believe this is possible using cyrptographic methods. The key observation that enables this is that any fact, rule or query can be represented as a set of hash values, such that any fact that matches a rule or a query shares exactly one hash value with it. This allows the cloud provider to store the statements encrypted, such that only their lawful readers can decrypt them, but still be able to apply bottom-up rules and perform queries.

5.2 Why Should Application Providers Switch Over?

So far we discussed ways to help users gain control over their data. But even if users are bought into this model, the application providers need an incentive to join in. After all, they currently make money out of user data, and relinquishing this data may harm their livelihood.

Another prominent source of motivation for application providers, second only to user data, are users themselves. Today's users are becoming more and more concerned about the lack of control they have over their data [31]. If this trend continues, users will be less and less comfortable using traditional Web applications, and an application that operates under the model we propose can get a competitive advantage.

Even without access to user data, application providers can still make a profit. The situation can be compared to the rise of *Open Source Software (OSS)*. Before OSS became popular, the most prominent source of revenue in the software world was licensing. When OSS started to emerge there were doubts regarding the ability of companies to make profit out of developing OSS. But today many software companies develop OSS as their primary product, and make money from, e.g., selling commercial support. We cannot anticipate the future business model that will emerge for such applications.

With that said, our model does not rule out the existing ways application provider make money; it just adjusts them, and puts them under tighter control. Many application providers make most of their revenues from selling ads. They leverage knowledge of their users to provide targeted ads. A typical process to do this involves two steps:

- *Analytics:* learning about relations between different things, e.g., people who were searching for plumbers tend to also search for carpet cleaners; and
- *Ad selection:* using specific knowledge about a user to provide that user with targeted ads, e.g., a user that searches for a plumber will see an ad for a carpet cleaning company.

Both steps can be implemented in our model without harming the user's privacy. Ad selection is a process that uses user data to provide information (ads) to the same user. This is the same as any private information processed by the application.

Things are a bit more delicate when considering analytics. There, data coming from all users needs to be processed together. However, the identity of the user is not important for analytics, so anonymized data may be used instead. While there is more to anonymizing user data than just removing user identity (as the real identity of a user can be inferred from seemingly unrelated facts), publishing anonymized data to be visible by all (i.e., $R = *$) can actually help detect identity leaks. Third-parties such as academics and privacy enthusiasts can analyze the data, trying to find privacy leaks and warn the community against them.

5.3 Is an Efficient Implementation Possible At All?

Finally, there are technical questions, e.g., can a CloudLog-based database be implemented so that it is efficient and scalable enough to address the needs of real-life applications?

Efficient Storage and Retrieval Our starting point involved three imperative predicates: `db_store`, `db_delete`, and `db_find`. Much of the "magic" happens in `db_find`, where axioms matching the given pattern are searched in

a large database. Unfortunately, this is not easy to do efficiently when the majority of data is located on disk.

A common way to store logic terms in a database is to sort them lexicographically, by converting each term into a sequence. For example, the fact $a\,(b\,(c)\,,d)$ can be represented by the sequence $[a/2, b/1, c/0, d/0]$ (we use name/arity pairs to refer to kinds of compound terms), and the rule $a\,(X) \rightarrow b\,(X)$ can be represented as $[\rightarrow /2, a/1, \$1, b/1, \$1]$, where variables are replaced with a Dollar sign ($) followed by their ordinal numbers in the term. These sequences can therefore be used as the key when storing terms.

If our database contains only ground terms, i.e., terms that do not have variables, looking for matches for a term such as $a\,(b\,(X)\,,Y)$ is easy. All we need to do is search for all keys that begin with $[a/2, b/1]$. For each such key we convert the sequence back into a term, and unify it with the query term. If we succeed, we have a result. However, if we allow non-ground terms, this approach will miss terms that are more general than the query term, such as the term $a\,(X, X)$.

To address this issue, we must either look for more general terms in the database (e.g., look also for terms beginning with $[a/2, \$1]$, which may result in a series of different disk accesses, or take a different approach altogether.

The different approach we take is based on range triggers. A *range trigger* is a procedure that does something in response to a change in the database within a given key-range. In our case, we would like to attach triggers to prefixes, so that changes made in the database from that point on will have a desired effect. When adding a rule such as $a\,(b\,(X)\,,Y) \rightarrow p\,(X, Y)$ to the database, it should make sure that the rule is applied to anything matching $a\,(b\,(X)\,,Y)$. To do so, it does two things: First, to handle the facts already in the database, it scans the database for keys starting with $[a/2, b/1]$, unifying each result with $a\,(b\,(X)\,,Y)$, and adding the resulting $p\,(X, Y)$ to the database. Second, to address all future facts to be added to the database, it places a range trigger on all keys starting with $[a/2, b/1]$, and whenever a key is added, it adds the corresponding $p\,(X, Y)$ to the database, and whenever a key is removed, it removes the corresponding $p\,(X, Y)$. This takes time out of the equation. It does not matter what came in before, the rule or the fact, the match will take place. When adding a fact to the database, we do the same, searching for matching rules and placing a trigger for future rules. This allows us to only scan more specific facts. If a more general fact (e.g., $a\,(X, X)$) matches our rule, the trigger placed by the fact will match the rule.

Consistency Bottom-up reasoning can take time. After adding a fact or a rule, more facts and rules can be added in response, and in turn, they can trigger more additions. By itself this is not a problem (given that it terminates). However, keeping such operations atomic becomes a challenge.

If the underlying storage supports ACID transactions, we could wrap the entire $\texttt{add}\,(DB, X)$ operation in a single transaction. Unfortunately, if transactions are very long and affect many keys in the database, scalability becomes an issue. On the other hand, if we do not wrap these changes in a transaction, an observer can see a situation where, e.g., A is true, $A \rightarrow B$ is also true, but B is not. This may yield a program incorrect and even unsafe.

To address this issue we developed Vercast [30], a software library intended to build databases based on principles from version control. Vercast allows databases to fork their entire state into a new branch, do some work on that branch, and then merge the changes atomically back to the branch we came from. A database can have any number of branches, and users are free to decide when to fork and when to merge (and where). This can be used for *add* and *remove* operations: Before performing such an operation, the database is forked to a new branch. After one or more such operations (e.g., a modification, consisting of a removal followed by an addition), the branch is merged back. We can design our database in such a way that two removals of the same axiom conflict. This way we can protect the database from conflicting modifications. If, for example, two users attempt to change the same fact by removing it and adding a replacement on two different branches, the last of them to merge the changes back to the main branch will experience a merge conflict and fail. Then they will have to look at the updated state and try again.

6. Related Work

This work makes contributions in two different fields: deductive databases and access control. We divide our discussion of related work into these two areas.

6.1 Deductive Databases

Work around deductive databases has been scarce in the last two decades. Still, Datalog-based deductive databases are still being used to solve problems otherwise hard to solve, such as the Semantic Web [13], machine learning [12] and more [2, 11]. All of these works acknowledge the declarativeness of Datalog as a key factor in the success of their work.

The BOOM Project used a Datalog-based language to implement a Hadoop-like distributed system [5], and introduced a dialect of Datalog that features some aspects of temporal logic [6]. Interestingly, their work on dialects of Datalog is not aimed at databases but rather at distributed systems, and their work on databases [7] is not related to Datalog but rather to the consistency issue we described in Sect. 5.3. There thus seems to be a reviving interest in the space we generally identified as NoDalalog.

6.2 Access Control

In this paper we use declarative programming as a way to assure the integrity and confidentiality of the data at the face of an untrusted application. Similar results have been

achieved for an imperative implementation, when the correctness and safety of the seL4 microkernel was formally proven [23, 24]. This proof includes integrity and confidentiality requirements. However, proving such features on imperative code requires tremendous effort. Declarative programming languages are typically easier for developing provable programs [10].

The notion of using statements, signed by a user in a logic system has been proposed as a solution for access control in a distributed environment [1, 8, 17]. Systems such as SecPAL [8] and Binder [17] have no single authority for setting a single set of rules, nor is there a single administrator who can assign permissions to particular users. To overcome this problem, they use the *says* modality [1]. Each atom, in both the head or the body of a clause can be prefixed with "`X says`," where `X` represents a user identity. This allows, in addition to reasoning over *what* has been stated, also referring to *who* stated it. For example, in the clause:

```
alice says canRead(X, File) :−
  trusts(alice, Y),
  Y says canRead(X, File).
```

Alice allows any user `X` to read any file `File` if some user `Y` who Alice trusts grants user `X` the same permission.

DKAL [20] adds a target audience to the mix, in the form of statements such as `X said Y to Z`. This makes `Z` *know* that `X said Y`. Although CloudLog targets slightly different settings, the problem we are trying to solve here is quite similar. In both cases there is no single authority who can grant specific permissions to particular users. The main difference between DKAL, SecPAL and Binder on the one hand, and Secure CloudLog on the other, is that the former languages are security languages, while Secure CloudLog is a data representation and query language. The *says* and *to* modalities are intended to assure the integrity and confidentiality of the conclusions that can be drawn from a program, while the program itself is intended to answer questions regarding permissions. Secure CloudLog is based on more expressive, Turing-complete semantics (Basic CloudLog), and is thus able to refer to sets of users, rather than individual users. This allows us to handle both integrity and confidentiality directly from the data language, with simple annotations.

7. Conclusion

The programming model described in this paper changes the balance of power between companies developing (and maintaining) applications and their end users. Today, user data is owned by the application and the application has full discretion over what to do with it. In our model the application is not given access to the data. It merely tells a trusted third-party what to do with it.

Our programming model is declarative, based on Logic Programming (LP). In principle, LP provides a convenient way to separate between applications (rules) and user data (facts). LP also lends itself to a concise and general definition of access control rules that apply to all end users and applications. In practice, however, we identified a gap in the field of LP that needs to be filled in order to make this possible, and coined the term NoDatalog to refer to the design space of deductive databases powerful enough to carry on this task. In particular, we presented a language for such databases, named CloudLog.

The TwitLog examples demonstrates that business logic can be expressed concisely in CloudLog, and that the resulting program is extensible, allowing new features to be added by only adding new rules and clauses to the database. It also shows that by using bottom-up rules, much of the processing is done at update time, and queries are performed by looking up results that are already waiting in the database, and does not require top-down evaluation.

The approach we propose in this paper can be considered disruptive innovation [9]. It requires a fundamental change in the way applications are developed and commercialized. These and other challenges notwithstanding, it demonstrates that separation of powers where applications and end users become peers is a possibility.

Acknowledgment

We thank Orr Dunkelman for helpful discussions and the anonymous reviewers for their constructive comments.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Sept. 1993.

[2] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'05)*, pages 358–367, Baltimore, Maryland, June 2005. ACM Press.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundation of Databases*. Addison Wesley, 1995. Available online at: `http://webdam.inria.fr/Alice/`.

[4] V. Abramova and J. Bernardino. NoSQL databases: MongoDB vs Cassandra. In *Proceedings of the 6th International C* Conference on Computer Science & Software Engineering (C³S²E'13)*, pages 14–22, Porto, Portugal, July 2013.

[5] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. BOOM analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*, pages 223–236, Paris, France, Apr. 2010. ACM Press.

[6] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. *Dedalus: Datalog in Time and Space*. Springer, 2011.

[7] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 ACM SIGMOD International Con-*

ference on Management of Data (SIGMOD/PODS'14), pages 27–38, Snowbird, Utah, June 2014. ACM Press.

[8] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security (JCS)*, 18(4):597–643, 2010.

[9] J. L. Bower and C. M. Christensen. Disruptive technologies: Catching the wave. *The Journal of Product Innovation Management*, 13(1):75–76, 1996.

[10] E. C. Brady. IDRIS: Systems programming meets full dependent types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification (PLPV'11)*, pages 43–54, Austin, Texas, Jan. 2011. ACM Press.

[11] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*, pages 243–262, Orlando, Florida, Oct. 2009. ACM Press.

[12] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling Datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[13] A. Calì, G. Gottlob, and T. Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. In *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'09)*, pages 77–86, Providence, Rhode Island, June 2009. ACM Press.

[14] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[15] K. L. Clark and S.-A. Tärnlund. *Logic Programming*. Academic Press, New York, 1982.

[16] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, fifth edition, 2003.

[17] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP'02)*, pages 105–113, Oakland, California, USA, May 2002. IEEE.

[18] E. Evans. Cassandra by example, May 2010. `http://www.rackspace.com/blog/cassandra-by-example/`.

[19] D. Featherston. Cassandra: Principles and application. CS591 Advanced Seminar, Department of Computer Science, University of Illinois at Urbana-Champaign, Aug. 2010. `http://d2fn.com/cassandra-cs591-su10-fthrstn2.pdf`.

[20] Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 149–162, Pittsburg, Pennsylvania, June 2008. IEEE.

[21] J. Han, E. Haihong, G. Le, and J. Du. Survey on NoSQL database. In *Proceedings of the 6th International Conference on Pervasive Computing and Applications (ICPCA'11)*, pages 363–366, Port Elizabeth, South Africa, Oct. 2011. IEEE.

[22] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216. ACM, 2011.

[23] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP'09)*, pages 207–220, Big Sky, Montana, USA, Oct. 2009.

[25] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–435, July 1979.

[26] R. A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, Jan. 1988.

[27] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[28] R. F. van der Lans. *The SQL Standard: A Complete Guide Reference*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[29] G. Lawton. Developing software online with Platform-as-a-Service technology. *Computer*, 41(6):13–15, June 2008.

[30] D. H. Lorenz and B. Rosenan. Versionable, branchable, and mergeable application state. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*, pages 29–42, Portland, Oregon, USA, Oct. 2014. ACM.

[31] M. Madden and L. Rainie. Americans' attitudes about privacy, security and surveillance. Technical report, Pew Research Center, 1615 L Street, NW, Suite 700 Washington, DC 20036, May 2015.

[32] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 2003.

[33] J. Patel. Cassandra data modeling best practices, part 1, July 2012. `http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/`.

[34] D. Pritchett. BASE: An ACID alternative. *Queue*, 6(3):48–55, May 2008.

[35] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23(2):125–149, 1995.

[36] P. J. Sadalage and M. Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 2012.