Oort: User-Centric Cloud Storage with Global Queries

Tej Chajed Jon Gjengset M. Frans Kaashoek
James Mickens* Robert Morris Nickolai Zeldovich
MIT CSAIL
{tchajed,jfrg,kaashoek,rtm,nickolai}@csail.mit.edu + mickens@g.harvard.edu

Abstract

In principle, the web should provide the perfect stage for user-generated content, allowing users to share their data seamlessly with other users across services and applications. In practice, the web fragments a user's data over many sites, each exposing only limited APIs for sharing.

This paper describes Oort, a new cloud storage system that organizes data primarily by user rather than by application or web site. Oort allows users to choose which web software to use with their data and which other users to share it with, while giving applications powerful tools to query that data. Users rent space from *providers* that cooperate to provide a global, federated, general-purpose storage system. To support large-scale, multi-user applications such as Twitter and e-mail, Oort provides *global queries* that find and combine data from relevant users across all providers.

Oort makes global query execution efficient by recognizing and merging similar queries issued by many users' application instances, largely eliminating the per-user factor in the global complexity of queries. Our evaluation predicts that an Oort implementation could handle traffic similar to that seen by Twitter using a hundred cooperating Oort servers, and that applications with other sharing patterns, like e-mail, can also be executed efficiently.

1 Introduction

The rise of user-generated content is a striking trend on the web. Users store and manipulate e-mail, calendars, spreadsheets, and photos in the cloud. They publish videos, blogs, product reviews, and social updates; they collaborate on documents, and communicate via comments in forums. In theory, the web provides users with the ability to share this data freely with an open-ended set of applications and users. In practice, however, web sites silo user data in private storage behind proprietary interfaces. While some sites do allow external applications to manipulate users' data, these abilities tend to be limited and site-specific.

We envision a web *without artificial application boundaries* — where users choose which applications they use to view and manipulate their data, and which users to share that data with. Once a user stores photographs in the cloud, she ought to be able to use a photo editor from one vendor, an organizer from another, and a presentation manager from a third. She ought to be able to share those photos selectively with any set of users, the general public, or nobody at all, independently of what applications those other users are using.

This paper describes Oort, a new cloud storage system based on the principle that user data should be separated from applications. This separation is intended to encourage an ecosystem of web applications that share access to user data. Each Oort user rents storage from a *provider*. Providers participate in Oort's distributed protocols to present a global name-space for all users' data objects. Applications, running in browsers or on separate servers, access users' data by talking to these providers, and Oort's access controls help users share selectively.

A key benefit of Oort's global name-space is the ability to combine many users' data in large-scale applications such as e-mail, social networks, and news aggregators. To help such applications find relevant data across users at many providers, Oort provides a *global query system* that locates data regardless of where it is stored, or which user created it. Applications issue queries that, in principle, span all data of all users of all providers, access controls permitting. The query language is a simplified form of SQL which treats each user object as a database row.

As an example, suppose Alice uses a Reddit-like discussion forum. To contribute to the forum, she creates and edits comments in her own Oort storage. All participating users run applications that issue Oort queries to gather articles and comments from users scattered over many providers. A query that fetches comments from the politics forum might look like this:

```
SELECT article_url, parent_id, content
 WHERE type = 'article-comment'
   AND forum = 'politics'
```

*Work performed as a Visiting Professor at MIT from MSR.

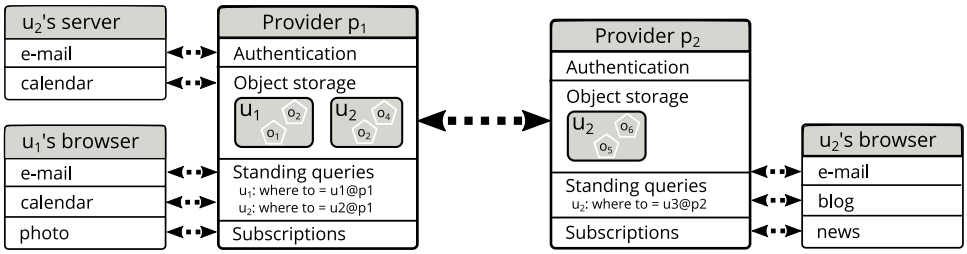

Figure 1: Overview of actors in Oort. Users run applications locally that interact with providers to access data. Providers cooperate to give each application's queries global scope.

Oort's providers cooperate to answer these queries across all data in Oort, hence the lack of a `FROM` clause.

An important benefit of Oort is that applications using common formats can share information. For example, Oort analogues of existing web sites such as Reddit and Hacker News can see each other's data in Oort storage, and, given knowledge of each other's schemas, can easily present a unified view of *all* comments for a particular article.

We expect Oort's data-centric focus to enable classes of web services that are currently difficult to build. For example, a universal messaging application can consult a user's e-mail, chat conversations, and Twitter interactions to assemble a full set of contacts, and display conversations with each contact across all these messaging systems. A calendar application can provide a unified view of events pulled from co-workers' calendars, even though each user might use different calendar software. It can also easily integrate with users' e-mail to scan for event invitations. Oort's common storage infrastructure makes these inter-application sharing scenarios possible without the need for application-specific APIs.

Oort's global queries pose performance challenges because of their power to retrieve data across many users and providers. The key insight is that, for large-scale applications, many users' applications issue queries with identical patterns, and perhaps even retrieve the same sets of objects. These overlaps enable providers to reduce the number of objects and queries they need to exchange. For example, all of the discussion forum's users will run applications that issue the above query. Oort merges these queries so that each provider sees at most one such query from each other provider, instead of one from every user. Measurements of our prototype show that these techniques will allow Oort to scale to the loads that large-scale web applications face.

The main contributions of this work are:

- A new cloud storage system that separates user data from individual applications and eases sharing of data by many applications.
- A global query primitive that helps applications find and gather content across all users in a wide-area distributed storage system.
- Techniques that reduce the cost of executing global queries by exploiting patterns in queries across users and applications.
- Prototype implementations of multi-user services that demonstrate the power of Oort's queries, and measurements that show Oort executes these applications efficiently.

2 Model

This section outlines the Oort storage and query design. The next two sections present application examples and then the design of Oort's global queries.

2.1 Providers and Principals

Each Oort provider offers storage, query execution, and authentication to its users. Some providers might operate on a commercial basis, charging their users, and others might be operated by organizations for the use of their employees. Providers are expected to implement Oort's protocols and interfaces, which include serving data and executing queries for each others' users, access controls permitting. We expect the number of providers to be significantly smaller than the number of users (*e.g.*, on the order of hundreds).

A principal in Oort is either a user or a group; users and groups are identical except for their initial authentication procedures. Each principal is associated with a "home provider" to which the principal sends most requests, and has a name of the form `aprincipal@homeprovider`.

Users authenticate with their home provider using a password (or some other provider-specific technique), and in turn receive session credentials from the provider. Applications prove that they speak for a principal by presenting these session credentials along with cryptographic proof of ownership. To obtain session credentials for a group, a

Login(user, password, session public key) → cert
GroupAuth(cred, group, session public key) → cert
CreateGroup(cred, groupname, members...)
Get(cred, handle) → object data
Create(cred, object data) → handle
SetACL(cred, handle, users...)
GetACL(cred, handle) → users
ChangeGroup(cred, groupname, members...)
RegisterStandingQuery(cred, sql) → standing query
ScanQuery(cred, sql, standing query) → set of objects

Table 1: Sketch of the Oort RPC interface a provider exposes to clients. A "cert" is a session certificate that indicates the user or group as whom the request should act; a "cred" is a session certificate along with cryptographic proof of ownership.

principal presents credentials to that group's home provider, which will issue session credentials for the group if that principal is indeed a member. When an application wishes to act as a group, it sends requests to the group's home provider, authenticated with the group's credentials.

2.2 Objects

Stored data takes the form of immutable objects, each consisting of a set of key/value fields. For example, an object might correspond to an e-mail message, with key/value fields indicating content and subject line. Principals can only create objects at their home provider, and the creator of an object is accessible through a special `.owner` field.

Each object is named by a handle that contains a cryptographic hash of the object's key/value fields, along with a hint indicating which provider holds the object. Objects are immutable to simplify caching, and to avoid the complexity of concurrent object writes; updates create new objects. Each object has a mutable access control list (ACL) indicating which users and groups can read it. This list may be modified only by the object's owner.

The contents of values within the fields of an object are chosen by the application that creates them. Oort will work best when applications use widely understandable data formats, such as plain text files, or formal and informal standards (*e.g.*, JPEG, H.264, iCalendar), as this enables interoperability with other applications. Adherence to such standards is already common, particularly among new vendors wishing to join an existing ecosystem.

2.3 Applications

Applications execute outside Oort, as shown in Figure 1, on any platform a user trusts to speak for him or her. Oort is designed primarily for client-side data access, though servers can also function as Oort clients, and can be given access to data by naming the principal they execute as in relevant ACLs. Applications might be an executable on a user's PC, JavaScript in a user's browser, or code that runs on behalf of a user on a third party's server. We use "client" to refer to an instance of an application running on such a platform.

A client interacts with Oort providers via RPC to fetch objects, create objects, and execute queries, using the API shown in Table 1. The Oort API makes provider boundaries mostly transparent to users and applications, as queries conceptually operate over the entire global set of objects.

Operations that involve other providers, such as Get() requests for remote objects, or GroupLogin() requests for other providers' groups, are sent directly to the relevant provider. This delegation is hidden from applications by the Oort client library, which determines what provider a request should be sent to based on the object handle or principal name passed to the library.

2.4 Queries

Most applications require the ability to locate objects using more flexible criteria than content-hash handles. For example, an application may wish to find all comments by a particular user, or all articles in a particular category. Furthermore, most applications need to be able to find objects created by different users, and hence need to be able to query data across multiple providers. Oort's solution to this is a global query system that conceptually covers all objects created by all users at all Oort providers, with *standing queries* to help the system execute these queries efficiently.

An Oort application expresses queries in a limited form of SQL. A query can filter, sort, group, and aggregate objects with simple expressions over the sets of keys and values for each object. The application issues queries to the user's provider in a specific way: it first installs a standing query that is expected to be useful over a long period of time using `RegisterStandingQuery()`. It then periodically examines the standing query's output with `ScanQuery()`. This arrangement allows providers to optimize over the set of long-lived standing queries, and build caches of remote objects likely to be useful in serving future client `ScanQuery()` requests.

As an example, consider the Reddit-like application mentioned in the introduction, which is replicated below. Users create and edit comments stored in their Oort provider, but applications need to collect the set of recent comments globally, across all users. The following Oort standing query suffices:

```
SELECT article_url, parent_id, content
 WHERE type = 'article-comment'
   AND forum = 'politics'
```

This standing query causes the user's provider to continuously accumulate Reddit comments created by users at all providers. Each time the user runs the application, it can send a ScanQuery() RPC to the user's provider to retrieve recent additions to the standing query's output. Oort providers register *subscriptions* with each other to satisfy users' standing queries. These subscriptions use filters derived from the standing queries, and are discussed in detail in §4.1.

The Oort prototype supports the COUNT, SUM, MIN, and MAX operators. Filters can be performed on object attributes, using equality and comparison against constants. Queries can use GROUP BY, ORDER BY, LIMIT, and a new operator ONLY MAX that is discussed in §3.4.

2.5 Group semantics

Oort applications are required to explicitly act as a particular group in order to get access to objects that group is allowed to see. This reduces the cost of Oort's access checks, which are performed often during query execution and must thus be efficient. This also simplifies certain queries, as the principal running a query is added as an implicit filter that is matched against object ACLs. This is used in §3.2 to efficiently implement e-mail lists.

2.6 Consistency and fault-tolerance

Oort provides eventual consistency across objects; since objects are immutable, individual objects are always consistent. Oort pushes new objects to interested remote providers as they are created, but while waiting for such a push, the remote provider may serve incomplete query results to its users.

Oort does not provide fault-tolerance at a protocol level. Instead, we expect storage providers to use industry-standard practices to provide high availability for their Oort services.

3 Developing Oort applications

In Oort, and on the web in general, users' data may be spread across many servers. While each application could conceivably contain code to interface with each such server and fetch relevant data, this places an undue burden on application writers. Instead, applications should have a convenient and concise interface to express interest in classes of objects, irrespective of location, and the system should take care of finding and returning the appropriate results to the application.

This section demonstrates that Oort's global queries are a good abstraction for this purpose. We implement a number of large-scale applications using global queries, and demonstrate use-cases that would be difficult to solve without them. We discuss the implementation of the query system in §4.

The examples cover the main sharing patterns for which Oort is well suited: sharing with a small, known list of individual users (e-mail), sharing with a larger group (mailing lists), and sharing public data with a large, dynamic set of users (Twitter). An example also shows how applications can use queries to organize private data such as e-mail folders, and to emulate mutable data. We evaluate implementations of the e-mail and Twitter examples given below in §6.

In all these applications, users create and edit their own content in storage supplied by their Oort provider; the full set of data needed by a multi-user application is thus generally spread across many providers. Applications collect that data using Oort's global queries, but do not need to be aware of how the data is distributed.

3.1 E-mail

As a first example, consider e-mail. In this design, a sender creates an object with `type` set to "e-mail", and with content and meta-data in other fields. The sender specifies recipients by adding them to the object's ACL. These steps constitute "sending" an e-mail, though the e-mail application does not explicitly send the object to the recipient or her provider.

To set up for receiving e-mail for Alice, her e-mail reading application installs this standing query:

```
EQ1: SELECT .owner AS from, subject, body
     WHERE type = 'e-mail'
```

Since Alice runs this query, only objects that Alice is permitted to view will be returned by the query system due to the implicit ACL matching condition. Whenever Alice's e-mail reader wants to check for new e-mail, it issues a ScanQuery() RPC to her provider to ask for new rows from the standing query's output. The e-mail reader can include a second SQL query in the call to ScanQuery() to filter the returned set of objects further, *e.g.*, by fetching only the latest message in each e-mail thread or limiting the results to recent messages.

`EQ1` causes Alice's provider to subscribe to e-mail objects Alice has access to on all other providers. When Bob creates an e-mail and adds Alice to the ACL, the subscription produced by Alice's standing query makes Bob's provider push a copy of the e-mail object to Alice's provider, where her e-mail reader can later retrieve it.

The power of global queries becomes even more apparent when multiple applications interact. Alice can use different applications for search, spam detection, mailing list maintenance, etc., all operating on her e-mail. These applications can exist completely independently from her

e-mail reader, and can all use Oort's global queries without incurring additional inter-provider traffic.

Applications can also combine e-mail with other user data. For example, a calendar application might query for a user's e-mails to scan them for event invitations, and can do so without any information about how a user's individual e-mail reader works. This is in contrast to current solutions, where such features often require either deep integration into users' e-mail clients (*e.g.*, Outlook), or for two applications to be developed together such as for the integration between GMail and Google Calendar.

3.2 Mailing lists

Consider adding support for mailing lists to e-mail as described above. Since the set of users on a mailing list changes over time, it is inconvenient to list each member separately on a new e-mail's ACL. Instead, a mailing list can be represented in Oort as a group whose members are the users on the list.

A user sends e-mail to the list by including the corresponding group in the message object's ACL. A user reads list messages by obtaining credentials for the group and issuing `EQ1` as a standing query to the group's provider. The rest of e-mail as described above remains the same, including the ability for Alice to let other applications use her e-mail in interesting ways.

For a user's e-mail reader to know the set of groups it should issue `EQ1` as, it needs a mechanism to be informed when the user has been added to a mailing list. When Bob adds Alice to the mailing list `theory@ibm.com` (*i.e.*, adds her to that group), his mailing list management application creates an object with `type` set to "list-membership" and `group` set to `theory@ibm.com`. It then adds Alice to the object's ACL. This query will tell Alice's e-mail reader which lists she is on:

```
G1: SELECT group WHERE type = 'list-membership'
```

Since the query runs as Alice, it only matches membership objects explicitly shared with her, just as for `EQ1` above. Once she learns about this new membership, she can authenticate as the new group, and then run `EQ1` as `theory@ibm.com`.

3.3 Twitter

Applications such as Twitter that share public data among many users fit well within Oort. Twitter users subscribe to each others' tweets, with some users having millions of followers. Unlike e-mail, Twitter users do not know who will read their tweets in the future; the set of followers (and ad-hoc readers) is dynamic.

In a Twitter-like application in Oort, a user "sends" a tweet by creating an object with `type` set to "tweet" and `content` set to the tweet text. The application then adjusts the object's ACL to make it public. To follow Bob (`bob@microsoft.com`), Alice issues the standing query

```
TQ1: SELECT content WHERE type = 'tweet'
       AND .owner = 'bob@microsoft.com'
```

The filter on `.owner` allows Alice's provider to send a subscription only to Bob's provider. Alice and Bob's providers can also merge Alice's query with the queries of other users interested in Bob's tweets. When Bob creates a tweet, his provider sends it to providers with interested users, and ensures that it is sent only once to each such provider. These optimizations are discussed further in §4.2.

As with e-mail, building Twitter in Oort enables applications to seamlessly use and combine Twitter data with data from other applications. For example, a user could run an application that retrieves all posts made by the celebrity Carol (`carol@celeb.com`) across a range of services by issuing queries and combining their output:

```
CQ1: SELECT content WHERE type = 'tweet'
       AND .owner = 'carol@celeb.com'
CQ2: SELECT caption, img WHERE type = 'photo'
       AND .owner = 'carol@celeb.com'
CQ3: SELECT title, content WHERE type = 'blog'
       AND .owner = 'carol@celeb.com'
```

Oort's global queries make this easy. Constructing such an application today would require interfacing with three web sites' custom APIs, each of which requires application registration, authentication, and result parsing.

3.4 Programming with mutable data

Oort provides immutable objects, but many applications need to modify existing data. For example, Alice might use the e-mail scheme described in §3.1, but also want to organize her e-mail into folders. Since the e-mail objects are immutable, the application cannot modify them to add folder names; a different mechanism is needed.

Applications can implement mutable data in Oort with objects representing successive versions. For example, Alice's e-mail reader assigns an e-mail to a folder by creating an object with fields mentioning the e-mail object, the folder name, and the time of folder assignment (`assigned_time`). It can later change the folder assignment by creating a new assignment object with a later time. The reader can use the following standing query to get the latest folder assignment for each of Alice's e-mail messages:

```
SELECT message_id, folder
 WHERE .owner = 'alice@oort.amazon.com'
 GROUP BY message_id
  ONLY MAX assigned_time
```

The special `ONLY MAX <field>` construct fetches just the object from each `GROUP BY` group with the largest

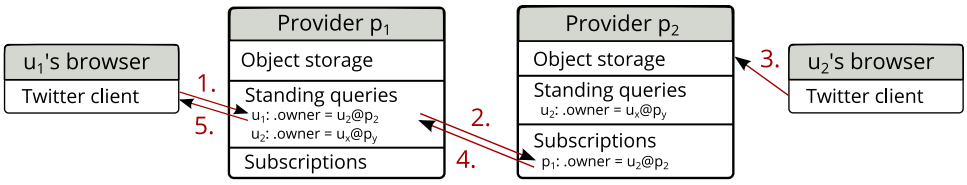

Figure 2: Example global query and response u_1 following u_2. 1) RegisterStandingQuery(): u_1's Twitter application establishes a standing query with its provider for new objects created by $u_2@p_2$; 2) Subscribe(): p_1 registers a subscription with p_2, asking for new objects created by $u_2@p_2$; 3) Create(): u_2's mail application creates a new e-mail object at p_2; 4) Push(): p_2 matches the new object with p_1's subscription, and pushes the object with its ACL to p_1; 5) ScanQuery(): the next time u_1's mail application asks for changes to the standing query output, p_1 tells it about u_2's new object.

Subscribe(sql) → (object1, acl1), . . .
Push((object1, acl1), . . .)
UpdateACL((handle1, acl1), . . .)

Table 2: Sketch of Oort's inter-provider protocol.

value for `<field>`. Alice's e-mail reader can find the messages in a particular folder, or the folder assignment of a particular message, by issuing `ScanQuery()` over the standing query above with filters on `folder` or `message_id` respectively.

4 Query system design

This section describes the design of Oort's query system, and details how it executes queries efficiently.

4.1 Inter-provider Protocol

Oort expects all providers to support the inter-provider API shown in Table 2. Figure 2 shows an example of the provider interactions that occur after a user registers a standing query.

When a provider receives a new standing query from an application, it uses the Subscribe() RPC to create an inter-provider subscription with each other provider (subject to merging as explained in §4.2). Each subscription contains only the filters from the standing query (the WHERE clause), without aggregation, grouping, or sorting operators.

When provider p_1 sends a subscription s to p_2, p_2 first records s persistently so it can match it against future object creations. It then searches for existing objects that match s, and sends these objects back to p_1. As p_2's applications create new objects, p_2 checks each one against the set of registered subscriptions; if one or more subscriptions from provider p_1 matches the object, and access controls allow any of p_1's users to see it, p_2 sends the new object to p_1 in a Push() RPC.

p_1's subscriptions yield a stream of new objects as they are created at other providers. p_1 stores these objects persistently, since standing queries are intended for long-term use. When an application uses ScanQuery() to look at a standing query's output, the provider executes the standing query against the locally stored objects to produce the query results.

Access control. Access control restricts which objects providers push to one another. p_2 will send an object o to p_1 only if o's ACL contains at least one user at p_1, as this means o's owner must trust p_1 to see o. In order that p_1 be able to decide locally which of its users can be allowed to see its copy of o, p_2 also sends a subset of o's ACL to p_1. It includes in this subset only users of p_1, to avoid sending unnecessary and potentially sensitive information.

Fault tolerance. Object pushes must be reliable in the face of temporarily slow or unreachable providers. Each provider maintains a persistent queue of objects to be pushed.

4.2 Query Optimizations

If Oort handled each user's queries separately, the total number of queries and subscriptions would be enormous. Worse, each distinct subscription might lead to a separate transfer of each matching object, causing objects to be transferred to a given provider multiple times. By exploiting overlap between subscriptions, Oort reduces the number of queries significantly, which limits both the space cost associated with keeping track of all active subscriptions, and the work a provider needs to do to match new objects against peer subscriptions when they are created.

Oort implements the following query optimizations:

- **same-filter**: A provider notices when standing queries from multiple of its clients would generate identical subscriptions to other providers, and does not re-issue the subscriptions. For example, the two queries:

```
SELECT article_url, content
 WHERE type = 'article-comment'
```

and

```
SELECT article_url, COUNT(handle)
 WHERE type = 'article-comment'
 GROUP BY article_url
```

both generate the same subscription, namely one for all objects `WHERE type = 'article-comment'`.

- **specific-owner**: If a query includes a filter `.owner = u@p2`, the provider sends a subscription only to provider `p2`. This keeps the number of subscriptions to a minimum when, for example, a user wants to know about updates to a few other users' calendars.

- **all-want**: If a provider receives the same subscription from multiple providers, it stores the subscription just once, along with a list of those providers. This allows the subscription's filters to be tested only once against a newly created object, instead of once per interested provider.

- **same-notify**: A provider sends a new object o at most once to each provider, even if o matches multiple subscriptions from that provider.

- **acl-providers**: A provider sends a new object o to a provider only if that provider is named on o's ACL, or the object is public.

The key insight for these optimizations is that each popular application causes many users to issue similar queries, and the same objects to be shared. Providers can merge these queries and object transmissions, and instead transparently share a smaller set of subscriptions and objects among many users. For common application patterns, when there are n users and p providers, this can reduce the total number of subscriptions generated by an application from $\mathcal{O}(np^2)$ to $\mathcal{O}(np)$, or even $\mathcal{O}(p^2)$. Furthermore, when a new object is created, it is sent at most once to each provider, even if many users at a particular provider have queries that match the object. Objects are never sent to providers that do not need them. We evaluate the effect of these optimizations on our example applications in §6.

4.3 Efficient subscription matching

When a provider creates a new object, it must decide which subscriptions' filters match that object's fields. Techniques exist that index subscriptions for efficient matching [3]. Since subscription filters typically compare object fields for equality with constants, the Oort prototype maintains an index for each field commonly mentioned in subscriptions, mapping each constant to the relevant set of subscriptions. Oort uses the most restrictive field index to narrow the subscriptions that must be matched. In practice, this means looking up subscriptions matching the new object's `.owner` field first, and then its `type` field.

Component	**SLOC**	
Oort provider	8414	
Go client library	483	
JS client library	279	
Go apps		
Reddit	283	Articles with votes
Mailman	419	Manage mailing lists
Dropbox	434	File and directory syncing
JavaScript apps		
Email	143	
Wikipedia	217	Create, edit, and view articles
StackOverflow	237	Questions, answers, and votes
Twitter	239	

Table 3: Lines of code for Oort core components and applications. We have implemented only the core functionality of each application that shares data between users, not user interfaces, single-user and auxiliary features, etc.

5 Implementation

The Oort provider prototype consists of about 8400 lines of Go, excluding tests (see Table 3). Prototype applications are written in Go for the command line or JavaScript for the browser. The provider implementation runs as a single multi-threaded process per provider. It stores objects in a MongoDB database. Providers communicate using Go's native RPC library over TCP. Clients communicate with providers using JSON RPC, with one connection per request. Our prototype does not currently implement persistent queues for cross-provider messages.

6 Evaluation

We use measurements of our prototype to evaluate the feasibility of a large-scale Oort deployment. We do not have access to enough servers and users to directly explore the absolute performance of a full-size Oort network. Instead, we demonstrate the viability of using Oort at scale by presenting measurements at small scale and extrapolating from those results.

First, we show that Oort's optimizations limit provider subscriptions to a reasonable number for common application patterns. Second, we demonstrate that Oort sends objects only where they are needed, and never more than once. And finally, we show that the absolute performance of our prototype implementation is high enough that it

could, in theory, support performance comparable to that of Twitter's real-life workload.

We discuss two applications in this section: the e-mail scheme from §3.1, and the Twitter-like scheme from §3.3. Twitter differs from e-mail in that the sender does not know who the recipients are, and in that the number of recipients can be very large for popular users. Together, these are representative of the communication patterns of a large number of the applications we anticipate will use Oort.

6.1 Experimental setup

All experiments run on a single 48-core Linux machine. The machine runs 20 Oort providers (one process for each), as well as one client per user, communicating over the loopback IP interface. Providers use a version of Oort that stores objects only in memory for experiments that count subscriptions and object pushes, since these do not depend on throughput, and would be slowed down unnecessarily if they had to wait for the disk. The throughput experiments in §6.4 use a single MongoDB server backed by an SSD; each provider uses a private database in MongoDB.

Each experiment involves n users, divided evenly among the 20 providers. The client load is generated by a process with a thread per user, where each thread keeps a single request outstanding. All experiments are run ten times, and the mean is plotted unless otherwise noted. Error bars are not normally shown, as the variance across runs is near zero.

6.2 Managing subscription load

This section explores the subscription burden that providers place on each other in Oort.

E-mail. For our e-mail scheme, each user's client first issues the standing query below to look for incoming e-mail. Clients do not fetch e-mail (*i.e.*, they do not run ScanQuery()), as the standing query is sufficient to ensure that providers generate subscriptions and push new e-mails.

```
EQ1: SELECT .owner AS from, subject, body
       WHERE type = 'email'
```

Figure 3 shows the number of subscriptions each provider must maintain, as a function of the total number of users. Each provider sees that all its users' mail readers issue the same standing query, and uses the **same-filter** optimization to maintain just one set of subscriptions. Thus each provider receives just one subscription from each other provider that has users querying for e-mail.

When there are fewer e-mail users than providers, this leads to roughly one subscription per provider with an e-mail user, so that the left-hand part of the graph increases. Once every provider has at least one e-mail user, the **same-filter** optimization prevents the total number of subscriptions from growing further. The net effect is a relatively low subscription burden on each provider.

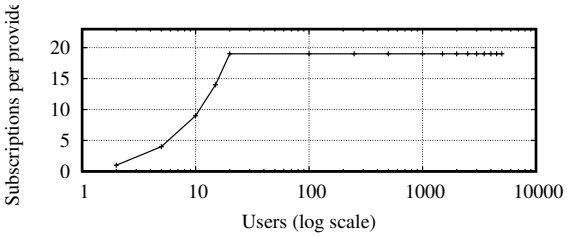

Figure 3: Subscriptions per provider as a function of the number of users for the e-mail application outlined in §3.1. The number of subscriptions does not grow beyond the number of providers because each provider merges its users' subscriptions.

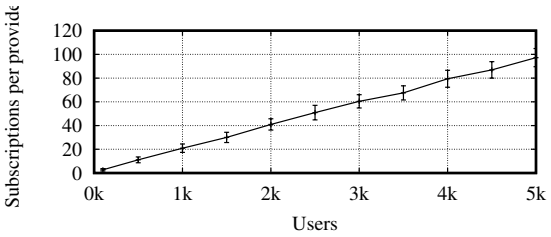

Figure 4: Subscriptions per provider as a function of the number of users for the Twitter application outlined in §3.3.

Twitter. In Twitter, a small fraction of users are very popular, and a large number of users are relatively unpopular. Since this influences the number of similar queries issued by users, we use an approximation of Twitter's social graph structure where 5% of the users have 95% of the followers. Each user follows between three and ten other users. One user follows another with this standing query (`TQ1` from §3.3)

```
TQ1: SELECT content
       WHERE .owner = 'user@provider'
         AND type = 'tweet'
```

Figure 4 shows the number of peer subscriptions each provider maintains as the number of users increases, with error bars showing standard deviation. At a high level, the graph grows linearly because each additional user follows a fixed number of users, each of which induce additional subscriptions. Particularly noteworthy is the fact that there are far fewer subscriptions than there are tweeter/follower pairs, which would be the case if no optimizations were used.

Behind the scenes, several of Oort's optimizations affect the number of observed subscriptions. Each `TQ1` generates a subscription to just one target provider due to the **specific-owner** optimization. If every follower generated a separate subscription, then 5,000 users would produce 32,500 sub-

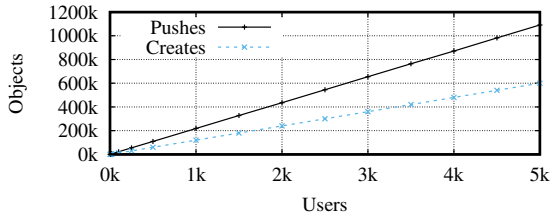

Figure 5: Object pushes and Create RPCs as a function of the number of users for e-mail.

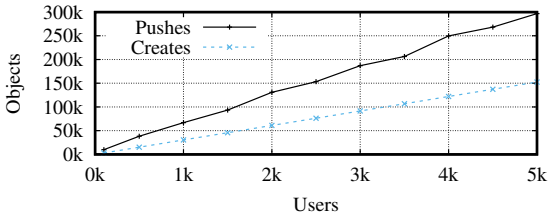

Figure 6: Cross-provider object pushes and Create RPCs as a function of the number of users for the Twitter application outlined in §3.3.

scriptions, and we would expect 1,625 subscriptions per provider for 5,000 users. The **same-filter** optimization eliminates many of these subscriptions, since most subscriptions are for the few popular users, and thus providers can merge most of their users' queries. The **all-want** optimization merges all the subscriptions for the same user at that user's provider, further reducing the number of subscriptions for the popular users who are named in many subscriptions. Because of popularity skew, there are also many users with no followers at all from other providers; this is why Figure 4 shows fewer subscriptions per provider than there are users per provider.

To estimate the effect our optimizations would have on Oort running at Twitter's scale, we determine the number of peer subscriptions that would be required for an approximation of the real Twitter graph[1]. We compute that a naïve implementation that distributes every query to every provider would observe three orders of magnitude more subscriptions than Oort. Compared to a more realistic scheme that only sends subscriptions to providers that produce relevant content (*i.e.*, **specific-owner**), Oort's remaining optimizations still reduce the number of subscriptions by a factor of five. A more accurate model will likely increase this factor, as our approximation uses uniform distributions in certain cases where we expect there to be skew (*e.g.*, number of users per provider), which is Oort's worst-case scenario. We measure the throughput of our prototype for a high number of subscriptions in §6.4.

In summary, Oort reduces the number of subscriptions from the number of tweeter/follower pairs to a fraction of the number of users, making the number of subscriptions reasonable even as the number of users grows large.

6.3 Object pushes

E-mail. After the e-mail subscriptions discussed above have been set up, every client creates a sequence of 120 e-mail objects, each addressed to between one and three users at random. Figure 5 shows the number of objects created and pushed. The number of pushes is slightly less than twice the number of created emails; this is because each e-mail has an average of two recipients, but some recipients are on the same provider as the sender, or share providers. The **acl-providers** optimization ensures that a provider pushes each newly created object only to recipients' providers. Thus, Oort only pushes exactly as many e-mails as needed to deliver every e-mail to its intended recipients.

Twitter. After all users have registered standing queries for the users they follow, each user creates 30 tweets. Figure 6 shows the resulting number of Create RPCs and cross-provider pushes.

The number of pushes is a function of how many distinct providers each user's followers are on. In the worst case, every follower is at a different provider, making the worst-case number of pushes 6.5 (the average number of followers per user) times the number of creates. However, there are three factors that bring the average down. First, followers that are co-located with the tweeter do not incur object pushes. Second, for users with many followers, the likelihood that some of those followers share a provider is high, which allows the **same-notify** optimization to be used. Third, as a result of the popularity distribution, some users have no followers, meaning their tweets need not be pushed at all. Since providers only subscribe to tweets by users their users have indicated interest in, and providers never push objects unless they match a subscription, objects are pushed at most once, and only where they are needed. Thus, Oort pushes objects as rarely and as sparsely as the follower graph allows.

6.4 Absolute performance analysis

The real Twitter has hundreds of millions of users, and averages 6000 tweets per second; record peak load is 25 times that [1, 17]. If Twitter's users and tweets were divided evenly across a hundred Oort providers, each provider's users would generate a total of 60 tweets/sec *on average*. Providers would receive tweets at several times that rate, as each tweet is sent to all providers with interested users; in the worst case, every provider may need to re-

[1]Follower counts here follow the heavy-tailed log-normal distribution, in accordance with analysis in [19].

p_s component	time	p_r component	time
Idle time	97%	MongoDB	69%
Marshalling	2%	GC	13%
MongoDB	1 %	Go runtim	13%

Table 4: Profiling results for the sending (p_s) and writing (p_r) provider with persistence.

ceive every tweet, and would thus have to absorb the full 6000 tweets/sec.

To determine whether it is feasible for our prototype to run applications at this scale, we measure the rate at which single-server providers can send and receive tweets. A sending and a receiving provider (p_s and p_r respectively) each run pinned to a dedicated core. A receiving client c_r registers 100,000 standing queries with p_r to follow 100,000 users at p_s, using query `TQ1` from §3.3. This generates 100,000 subscriptions at p_s. 100,000 subscriptions is fewer than the actual number a provider would see in Twitter. However, its main effect is on the $\mathcal{O}(\log(n))$ lookup time in an index of subscriptions on `.owner`, so more subscriptions would not affect throughput very much, only increase memory usage.

Once the subscriptions have been set up, a sending client c_s, acting as one of the users c_r followed, creates 1000 tweets on p_s, one at a time. For each tweet, p_s matches the new object against its set of subscriptions, and sends it to p_r. p_s sends each tweet object to p_r 100 times, in order to simulate the work that would be required if c_s had a follower at each of 100 providers. p_r stores all 100 copies of each received tweet (though c_r never asks for them in this experiment), which approximates all 100 providers pushing tweets to p_r.

We measure the time from the first create request to when the last tweet's push has been processed by p_r. While it involves only two providers, the experiment includes all of the processing a provider would do in a larger system, except for the work involved in clients fetching tweets. We therefore believe it should be indicative of the performance providers would see in a larger deployment.

Results with persistence. The experiment above achieves a throughput of 9.81 tweets/sec, suggesting that a hundred providers would be able to sustain a total workload of about ≈1000 tweets/sec. Table 4 gives a breakdown of where each process spends its time, and shows that the overall bottleneck is time spent waiting for MongoDB at p_r to store received tweet objects; measurements indicate that MongoDB does not saturate the disk (it writes at less than 1 MB/sec), and is CPU bound. We expect providers would use a more efficient database, and that they would divide the load over a cluster of many servers.

Results without persistence. To find the limiting factor for object creation, we modified p_r to return immediately in its Push RPC handler to simulate providers with optimized receive paths. This experiment achieves an average throughput of 115 tweets/sec, suggesting that a hundred providers should be able to generate 11,500 outbound tweets/sec in aggregate. This is nearly twice the 6000 tweets/sec required for Twitter's average workload.

p_s component	time
Idle time	38%
Marshalling	20%
MongoDB	14%
GC	14%
Go runtime	4%

Table 5: Profiling results for the sending (p_s) provider when p_r returns immediately in Push RPC handler. Idle time on p_s is time spent waiting for p_r to de-marshal incoming requests.

Table 5 gives the breakdown of time spent for p_s and p_r for this experiment, and shows that the bottleneck in this experiment is primarily marshalling of Oort objects.

Inter-provider network bandwidth. Another potential limit to the feasibility of Twitter on Oort is the wide-area network bandwidth required; in the worst case, a copy of each tweet must be sent to each provider. If we assume sending a tweet generates 200B of network traffic, a provider would need to maintain outbound and inbound data rates of 1.2 MB/sec on average. This is a reasonable amount of network traffic for an application used by millions of users. In reality, many tweets need only be pushed to a subset of the providers, further reducing the network bandwidth requirements.

This analysis suggests that it is feasible for a fully deployed Oort system to handle the load generated by Twitter-scale applications.

6.5 Summary

In summary, Oort provides applications with convenient queries, but keeps the number of subscriptions a provider must maintain low, and the number of objects it must push small, even with a large number of users. Measurements show that Oort scales well, and that it is feasible for our prototype to handle a workload similar in size to that of Twitter.

7 Discussion

In this section, we discuss some open questions about Oort's design, as well as some of the non-technical deployment questions about Oort.

7.1 Trust model

Oort requires users to entrust their provider with their data. We believe this is reasonable given that users may choose what provider to sign up with. To share data with a user residing at a different provider, the owner must also be willing to trust that user's provider with the data in question. In Oort, providers are trusted proxies for their users, so trusting a user also means trusting that they chose a trustworthy provider. Since data is only sent to providers that have at least one of their users listed on the ACL, data is never shared to a remote provider without the owner's consent.

The success of platforms such as Google Apps and Dropbox suggests that these trust requirements are reasonable; users are willing to upload their data to the cloud, and to have companies manage that data. Since users either pay or are otherwise associated with Oort providers, providers are motivated to be careful about security. At the application level, Oort gives users uniform and transparent access controls. This is an improvement over many existing web sites, where it can be difficult for users to discern to whom their data may be visible.

7.2 Provider economics

Since we expect providers to have financial relationships with their users, ether directly (*e.g.*, renting space) or indirectly (*e.g.*, through employment), we believe it is reasonable to expect providers to do work on behalf of its users. Oort often makes it easy for providers to determine exactly which users should be billed for what operations. For example, if Alice issues a query, she could be billed for each returned result.

However, there are operations where the economic model is less clear: if one of Alice's objects is pushed to satisfy queries of other providers' users, it is not clear that Alice should have to pay. If the volume of peer subscriptions and object pushes is similar in both directions between providers, the providers may be willing to do this work for free for each other. In the cases where there is a substantial imbalance in traffic volume, the solution is not so obvious. In many ways, this problem is similar to that of peering among network providers in the Internet, and experience from that setting may guide the development of inter-provider connections in Oort.

7.3 Excessive Load and Abuse

An application may innocently issue a query that generates more work than intended. In many cases, the negative feedback from the application's users who experience low performance may be enough to persuade the developers to fix the query, but in the extreme, providers will likely need to throttle expensive queries.

Some abuse is likely to be malicious. Application-level attacks, such as generation of spammy forum comments and e-mail, can be dealt with by application-specific techniques such as those used in existing web sites. This model works well within the Oort ecosystem; users could share their objects with a spam-detection application, and could read "decision objects" published by the application to determine if any have been tagged as abusive.

Attacks launched by other malicious providers are a different story. Pushing a large number of objects requires significant effort on the part of the attacker, since they must send as much traffic as they wish to impose on the other providers, and so is not a significant attack vector. On the other hand, since providers are expected to fulfill subscriptions for each other, a malicious provider *could* attempt to overload another provider by creating an excessive number of bogus subscriptions.

As a partial solution, we expect providers to limit how much time they spend matching new objects to other providers' subscriptions. If a provider registers too many subscriptions, they may be throttled or ignored. Providers could use application-level signals, such as the spam filter results, to determine that pushed objects are unwanted even though they matched a subscription.

7.4 Least privilege

The ability to query across all user data is powerful, and users may want to limit this ability on a per-application basis to prevent applications from accessing data they should have no legitimate need to access. For example, Alice might not want her Twitter client to be able to read her file backups. A privilege reduction system such as this would also allow users to place limited trust in third-party services like spam filters, giving them only access to the data they require to perform their services. We envision that this feature could be provided through the use of *sub-principals* that inherit only a subset of the parent principal's privileges, but further research is needed to establish how this is best implemented in Oort.

7.5 Non-technical questions

Migration to Oort. It seems unlikely that many existing large web sites would immediately switch to Oort; their revenue often comes not from charging users for a service, but from selling advertising based on exclusive access to their users' data. While such advertising is possible in Oort, the intent is that users' data not be tied to any one application.

Oort depends on users being willing to pay to use software and services that give them more flexible access to their data, and on vendors of new software being willing to cede that control if users are willing to pay. This trend can

be seen in the recent appearance of many federated, user-centric versions of popular web services that move away from the current application-centric, walled-garden model of the web. Diaspora*, an open-source, federated Facebook replacement; Ello, a no-advertising, paid features, social network; and pump.io/identi.ca, a framework for federated user-to-user data sharing, are a few such examples.

Oort can coexist and develop alongside the existing web. In the beginning, we imagine that primarily new applications, and applications with only small-scale sharing, will use Oort. It is attractive even for the first application to use it, for example as a convenient storage system for otherwise serverless in-browser software. As the amount of user data stored in Oort grows, more applications will be written to exploit that data, and there is a potential for network effects to drive adoption.

8 Related Work

Oort builds on ideas the authors have previously presented in [9]. The workshop paper introduces the notion of providers and global queries, but does not study the feasibility of building and using such a system. It does not provide detailed designs for how global queries work or how providers interact, and does not discuss how the cost of global queries can be reduced to a manageable level.

Oort resembles systems such as W5 [18] and BStore [10], but provides cross-provider queries that allow applications to compute across all users' data on a global scale. Similarly, Invisible Glue [6] enables joint access to heterogeneous data storage, but does not support multiple applications, nor data distributed across multiple disjoint cloud providers. User data collation systems such as openPDS [13] are similar in spirit to Oort, but focus on the problem of single-user data collection rather than multi-user data sharing. Solid [20] has a similar data model as Oort, where data is stored per user, not per application, but targets only storing and traversing social graphs, and does not provide general-purpose queries.

Oort's standing queries are similar to continuous queries [4] over the stream of all created objects. Subscriptions function similarly to content-based subscription systems such as Siena [8] and Thialfi [2], but benefit from the richer set of features provided by Oort's global queries.

Oort would benefit from existing work on query optimization to help match newly created objects against subscriptions. Prior work has produced efficient algorithms for both indexing [3, 15] and merging [5, 12, 23] such queries for content-based subscription systems. We can also draw on much existing work about distributed databases, distributed query processing, and federation [11, 16, 21, 22].

Systems that ease the construction and maintenance of distributed web applications have previously been proposed in Sapphire [24] and Orleans [7]. These systems aid in the development of traditional web applications, but do not address the tight coupling of user data and applications.

Web application platforms such as Dart and Meteor aim to unify developing the browser and server components of web applications. Oort complements these approaches by providing developers with global, application-independent storage and global queries.

Many web applications share user data through general-purpose storage backends like Dropbox with standard authentication mechanisms such as OAuth [14]. This approach requires service-specific integrations similar to that of custom application-specific APIs. That applications tolerate the significant complexity this introduces is a sign of the desire for the functionality Oort provides.

Existing multi-user ecosystems, such as Google Apps and Microsoft Windows Live, support mostly transparent sharing among their own applications and users. They usually also allow external applications to acccess and query users' data through custom APIs or explicit data exports, but these vary between different services, and between vendors. In Oort, all applications access user data through a single, unified interface, irrespective of where the data is stored. Oort requires neither individual users nor collaborators to use software from the same vendor.

9 Conclusions

Oort's design reflects a belief that data should live free of application constraints. This allows users to choose freely which applications to apply to their own data, and which other users to share that data with. Oort achieves this goal with user-centric global storage that is separate from applications. To help applications fetch and combine content from users scattered across the Internet, Oort provides flexible queries with global scope. Oort executes these queries efficiently by exploiting overlap in the queries generated by application instances executed by many users; a collection of optimizations allows extensive merging of queries and elimination of repeated work. Measurements of a prototype implementation demonstrate that Oort can scale to the load large-scale web applications face today.

References

[1] About Twitter, Inc. Accessed: 2015-03-26.

[2] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A client notification service for internet-scale applications. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, 2011.

[3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 53–61, New York, NY, USA, 1999. ACM.

[4] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, Sept. 2001.

[5] S. Bittner and A. Hinze. The arbitrary boolean publish/subscribe model: Making the case. In *Proc. 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 226–237, Toronto, Ontario, Canada, 2007. ACM.

[6] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, United States, Jan. 2015.

[7] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proc. 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14, Cascais, Portugal, 2011. ACM.

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. 19th ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 219–227, Portland, Oregon, USA, 2000. ACM.

[9] T. Chajed, J. Gjengset, J. van den Hooff, M. F. Kaashoek, J. Mickens, R. Morris, and N. Zeldovich. Amber: Decoupling user data from web applications. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[10] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with BStore. In *Proc. 2010 USENIX Conference on Web Application Development (WebApps '10)*, Boston, Massachusetts, 2010. USENIX.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Hollywood, CA, USA, 2012. USENIX Association.

[12] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE Trans. on Knowl. and Data Eng.*, 15(1):174–191, Jan. 2003.

[13] Y.-A. de Montjoye, E. Shmueli, S. S. Wang, and A. S. Pentland. openPDS: Protecting the Privacy of Metadata through SafeAnswers. *PLoS ONE*, 9(7):e98790, 07 2014.

[14] D. Hardt. The OAuth 2.0 authorization framework. RFC 6749, RFC Editor, Fremont, CA, USA, Oct. 2012.

[15] S. Kale, E. Hazan, F. Cao, and J. P. Singh. Analysis and algorithms for content-based event matching. In *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05) - Volume 04*, ICDCSW '05, pages 363–369, Washington, DC, USA, 2005. IEEE Computer Society.

[16] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, Dec. 2000.

[17] R. Krikorian. New tweets per second record, and how! `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`, 2013.

[18] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web without walls. In *Proc. 6th Workshop on Hot Topics in Networks (HotNets-VI)*, Atlanta, Georgia, Nov. 2007. ACM SIGCOMM.

[19] Luminoso. Twitter followers do not obey a power law, or Paul Krugman is wrong. `http://blog.luminoso.com/2012/02/09/twitter-followers-do-not-obey-a-power-law-or-paul-krugman-is-wrong/`, 2012.

[20] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference Companion on*

World Wide Web, WWW '16 Companion, pages 223–226, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[21] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, Sept. 1990.

[22] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[23] S. Tarkoma. Chained forests for fast subsumption matching. In *Proc. 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 97–102, Toronto, Ontario, Canada, 2007. ACM.

[24] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/-cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO, Oct. 2014. USENIX Association.