Where in the World is My Data?

Sudarshan Kadambi[1] *, Jianjun Chen[1], Brian F. Cooper[1] *, David Lomax[1], Raghu Ramakrishnan[1], Adam Silberstein[1], Erwin Tam[1] *, Hector Garcia-Molina[2]

[1]Yahoo! Santa Clara, CA

[2]Stanford University Stanford, CA

ABSTRACT

Users of websites such as Facebook, Ebay and Yahoo! demand fast response times, and these sites replicate data across globally distributed datacenters to achieve this. However, it is not necessary to replicate all data to all locations: if a European user's record is never accessed in Asia, it does not make sense to pay the bandwidth and disk costs to maintain an Asian replica.

In this paper, we describe mechanisms for selectively replicating large-scale web databases on a record-by-record basis. We introduce a flexible constraint language to specify replication policy constraints. We then present an adaptive scheme for replicating data to where it is most frequently accessed, while respecting policy constraints and using minimal bookkeeping. Experiments using a modified version of our PNUTS system demonstrate our techniques work well.

1. INTRODUCTION

If users must wait longer than one second for pages to load, they can become distracted and may decide to switch to a different website [20]. In order to provide fast page loading for users across the globe, data must be physically located near the user. However, the cost of replicating all data everywhere can be prohibitive. If a dataset is frequently updated, those updates must be sent over a wide area network to every datacenter with a replica, incurring high bandwidth costs. Furthermore, every datacenter must be provisioned with enough servers to handle the updates. We can avoid some of this cost if we do not replicate data to locations where it is infrequently accessed. Consider for example a user record for Alice, who lives in Europe and constantly changes her profile. If Alice never travels to Asia, and her profile is rarely accessed from Asia, the network and disk bandwidth to propagate her updates to the Asian datacenter are not justified. On the other hand, she might have many admirers in the U.S. who constantly view her profile in a social networking application, and maintaining a replica in a U.S. datacenter will reduce response times for those users (and even save on bandwidth if her profile if viewed more often from the U.S. than the frequency of her updates).

A replication framework must also support a variety of policy constraints. For example, legal constraints prevent us from replicating user data to certain jurisdictions. When users create a Yahoo account, they agree to a terms-of-service contract (TOS) which details where Yahoo is allowed to store their data. So, while the namespace of the user database needs to be global (so that each ID is globally unique), all other data can only be in certain places. As another example, we may decide to always keep at least three copies of a critical data item in pairwise resilient locations to facilitate disaster recovery, even if the record is only ever accessed from one location.

In this paper, we consider how to implement a *selective replication* mechanism for a globally replicated web database. To test our techniques, we have extended our PNUTS system [9], a globally replicated, scalable web database that is used in production at Yahoo. The production version of PNUTS decides what data to replicate where at a table granularity (though updates are *p*ropagated at the per-record granularity). Our extensions to PNUTS support a finer-grained, per-record selective replication policy. The main goal of this mechanism is to minimize replication cost while respecting policy constraints (such as legal constraints or minimum replication levels). Furthermore, the mechanism can be tuned to support a latency guarantee. An example of such a guarantee is a Service Level Agreement (SLA) of the form "95% of reads must be satisfied in 50 ms or less."

We have designed a constraint language that allows web developers to specify policy constraints as a function of the content of the record (Section 4). The system has the freedom to place records to achieve system efficiency, subject to these constraints. Since it is difficult to predict future access patterns, we have designed a mechanism, called *dynamic placement*, for adaptively deciding where to make replicas. Dynamic placement uses minimal bookkeeping to dynamically migrate replicas away from locations where they are not often accessed to locations where the read rate is higher, subject to any specified constraints. Our approach is novel in considering both system efficiency and replication policies, and has a unique, record-centric adaptive design (Section 3) that balances local and global decision making: each region decides on whether adding a replica of a record will benefit the local usage of that record, but the *r*ecord master (each record has a unique master copy) decides whether to

*Kadambi now at Bloomberg. Cooper and Tam now at Google.

Proceedings of the VLDB Endowment, Vol. 4, No. 11

allow the replication, taking system-wide costs and policies into account. The local aspect allows for near-optimal performance to be achieved with minimal book-keeping, and the global aspect allows flexible policies to be enforced. The design also avoids bottle-necks and gains fault-tolerance by leveraging PNUTS' notion of record masters (the masters for records in a table are widely distributed based on access patterns [9], and the system is capable of continuing in the presence of failures by automatically transferring mastership to surviving copies). We present an experimental study, using a modified version of PNUTS installed in datacenters around the world, to demonstrate the effectiveness of our techniques in a real web database setting (Section 5).

Our approach consists of a collection of design choices that work together to address complexity, performance and policy issues encountered in managing geo-replicated data at Yahoo!. We discuss these design choices where appropriate throughout the paper; we also highlight some practical considerations in Appendix A.7. We review related work in database replication and caching in Section 2 and conclude in Section 6.

2. RELATED WORK

Cross-datacenter replication is important for web-scale databases such as PNUTS [9] and BigTable [7]. To our knowledge, few systems have focused on selective replication for optimizing bandwidth and supporting flexible policies, as we discuss here. For example, Cassandra [18] does support fine-grained control of cross-datacenter replication, but the control is static and not based on access patterns. Dynamo [12] sacrifices consistency across replicas to achieve high level of availability. We believe our selective replication algorithms are applicable to those systems also.

Caching versus replication Kossman [16] defines *caching* as storing transient copies of individual data objects, usually in memory and usually at the edge (e.g., not at a database server), and invalidating them on update. In contrast, *replication* typically operates at the table level, and stores long-lived data on disk on database server machines. Replicated data is maintained by propagating updates. Our selective replication policy is a hybrid of these two approaches: the granularity is record level and copies are transiently stored (as in caching) but storage is on disk for use by the database server and we propagate updates (as in replication).

Dynamic replica placement Computing the best placement of database replicas from a trace of accesses is NP-complete [6]. Furthermore, even if a historical trace is known, access characteristics can change. Adaptive schemes use heuristics to place replicas based on current access patterns. Our dynamic placement technique most resembles the ADR algorithm of Wolfson et al. [25], which dynamically creates and deletes replicas in response to access patterns. However, our techniques differ in fundamental ways from ADR. First, we avoid tracking read and write statistics directly. In a web-scale database, maintaining even simple statistics at a per-record granularity can be complex and expensive. Second, we integrate a richer type of constraints, which include inclusion and exclusion lists. Third, we replace some of the inter-server coordination of ADR with decision-making by the record master, which simplifies the protocol and is especially important in a geo-replicated setting. Our dynamic placement model is unique in that it combines centralized and localized strategies at the record level. Specifically, a replica makes a local decision based on its observed read/write patterns, and submits it to its record master, which makes the final decision based on application constraints. Mariposa [22] uses an economic model to provide dynamic placement of data replicas across autonomous sites. Our model is designed for networks of datacenters owned by the same entity and without a market-based resource allocation system.

Dynamic cache placement Examples of caching techniques are described in [1, 4]. Multiple techniques have been developed for deciding where to place cache replicas based on access patterns (for example, [15, 21]). This work focuses on dealing with storage constraints, rather than on minimizing bandwidth usage. Further, it assumes updates are rare and can be ignored when computing optimal placement, or that updates simply cause invalidations and thus the cost of updates is low. In our system, updates must be propagated to full replicas, and must be considered when placing replicas. Yu and Vahdat [26] consider how to place replicas based on availability; we focus on resource usage and performance.

The placement algorithm of [24] considers bandwidth optimization and update costs. However, the authors list several practical limitations, such as the assumption that all objects have the same size, and computing the placement requires complex coordination between all of the distributed nodes.

Kossmann et al. [17] discuss how to analyze a distributed query plan to determine whether cached copies should be made at specific locations. They focus on complex relational query plans; our queries are much simpler, and we can use simpler techniques without sophisticated statistics to determine our placement. Similarly, caching of query results [11] uses the semantics of the query to decide which results are useful to other queries; but such semantics are not present in the simple CRUD workloads we deal with here.

Other replication paradigms Early work on scalable, highly available distributed databases includes [19]. However, the focus is on memory-resident clusters with fast interconnects and they do not optimize for bandwidth. Peer-to-peer systems use replication to make it easier to process queries for content. Shark [5] focuses on how to cooperatively cache among mutually distrustful clients; in our setting all replicas are typically owned by the same entity. Cohen and Shenker [8] describe a replication policy that makes peer-to-peer search more efficient. Our queries are local lookups rather than distributed, peer-to-peer search. Schism [10] attempts to find a good partitioning scheme based on known workload to minimize the number of distributed transactions across multiple records, while our approach dynamically places individual records based on read/write patterns. Another broadly related work is materialized views in database system literature [14].

3. SELECTIVE REPLICATION

We start by describing an architecture to support selective, per-record replication. Given this architecture, we can formally define the optimization problem we need to solve.

3.1 Architecture

In a globally replicated database, tables containing records or other data objects are copied to geographically separated datacenters. For example, copies might be made in datacenters in Singapore, on the west coast of the U.S., on the

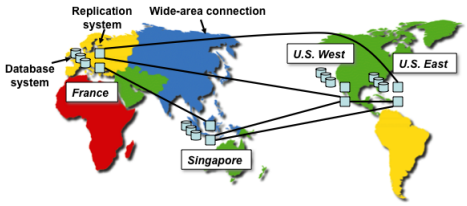

Figure 1: Globally replicated database that asynchronously propagates updates to remote datacenters.

east coast of the U.S., and in France. For clarity in our discussion, we will focus on a single table containing records, but our techniques generalize directly to multiple tables or other data models. Each *replica location* stores a full or partial copy of the table.

Because of the high latency for communicating between datacenters, replication is typically done asynchronously. Usually, writes are persisted at one or more local servers and acknowledged to the applications (e.g., made *1-safe*). Later, updates are sent to other replica locations. An example of this architecture is shown in Figure 1. As the figure shows, we can think of the system as having two distinct components: a *database system*, which manages reads and writes of data records, and a *replication system*, which manages replication of updates between replica locations. In real systems these components might be on the same server (as in MySQL replication [3]) or different servers (as in PNUTS [9]). The replication system must ensure reliable delivery of updates to remote datacenters despite failures. Individual servers might fail (and even lose data), but local or remote copies can be used for recovery.

In each location, a given record exists either as a *full replica* or as a *stub*. A full replica is a normal copy of the record, possibly enhanced with metadata to support selective replication, such as a list of other full replicas. A stub contains only the record's primary key and metadata, but no data values. Note that we do not consider selective replication at the field or column level in this paper.

3.1.1 Handling reads and writes

We assume that there is a master copy of each record where updates are applied before being propagated to replicas. In PNUTS, the master copy for different records might be in different datacenters: Alice's master copy might be in France while Bob's master copy might be in India. The result is a per-record consistency model, where replicas might lag the master by one or more versions, but will always eventually receive all updates (and apply them in the same order). No locking or commit protocol is needed since transactions are per-record; for more details see [9]. Our techniques extend to systems that do not have a master and allow updates to be applied anywhere (e.g. [12, 18], and as we discuss in Section A.7, a mode of PNUTS that supports eventual consistency).

When a record is inserted, the master copy decides where the full replicas of the record are to exist, and sends full replicas and stubs to the appropriate locations. When a record is updated, the master applies the update and then sends the updated data only to the locations that contain a full replica. This is where the resource savings of selective replication come from, since bandwidth and disk I/Os are only necessary for full replica locations. If a record is updated in a non-master region, the update has to be forwarded to the master, but this is because of the mastership scheme, not specifically selective replication. When a record is deleted, a message is sent to all replicas (full and stub) to notify them to delete the data.

A record may be read from any location. If the local database contains a full replica, it serves the request. Otherwise, the database reads the list of full replica locations from the stub and forwards the request to one of them (preferably the one with lowest network delay). This is the main penalty for selective replication: some reads that would have been served locally if all data were replicated everywhere now need to be forwarded, with an attendant increase in response latency (and some cross-datacenter bandwidth cost). As we increase the number of full replicas, there are fewer forwarded reads but also more cost to propagate updates.

It may be necessary to change the set of full replicas if, for example, the access pattern changes. In this case we might *promote* some stubs to full replicas and *demote* some full replicas to stubs. In our mechanism, each location requests promotion or demotion for records based on local access patterns, but the master decides whether to grant the request. This allows the master to enforce constraints like a minimum number of copies (see Section 4.1). If the master decides to convert a replica, it notifies all regions of the new list of full replicas for this record to ensure that reads can be properly forwarded. Additionally, if promoting a stub, the record data must be sent to the location with the new full replica.

3.2 Optimization problem

Inter-datacenter bandwidth can be extremely expensive, especially for datacenters with limited backbone connectivity. Therefore, we optimize system cost by minimizing bandwidth used. Other costs, such as server cost, are also important. However, minimizing bandwidth usage means avoiding sending traffic to some datacenters, which will also reduce the number of servers needed in that datacenter. Thus, bandwidth is a useful proxy for total system cost.

Inter-datacenter bandwidth for replication consists of:

- *Replication bandwidth*: The bandwidth required to send updates between datacenters.
- *Forwarding bandwidth*: The bandwidth required to forward read requests to remote datacenters because the local replica contains a stub.

We want to minimize the sum of *replication bandwidth* and *forwarding bandwidth*.

Additionally, two types of constraints must be enforced. First, *policy constraints* specify where data must or cannot be replicated, often for legal reasons. Policy constraints may also specify a minimum number of full replicas to ensure data availability. Second, *latency constraints* might specify that the majority of users experience good response time. It is convenient to express this constraint by specifying the fraction of total global reads (e.g., 95%) that must be served by a local, full replica. Satisfying these constraints may mean making more full replicas, or making full replicas in different locations, than would result from simply trying to minimize bandwidth cost.

Then, we can define our optimization problem as follows:

Definition 1. **Constrained selective replication problem** - Given the following constraints:

- *Policy constraints* that define the allowable and mandatory locations for full replicas of each record, and the minimum number of full replicas for each record, and
- A *latency SLA* which specifies that a specified fraction of read requests must be served by a local, full replica

choose a replication strategy to minimize the sum of *replication bandwidth* and *forwarding bandwidth* for a given workload. □

Section 4 describes policy constraints. The specification for latency SLA is described in Appendix A.1.

4. REPLICA PLACEMENT

The main decision we make is where to place full replicas of each record, since we do not consider replicating just parts of a record. In this section, we first describe our language and mechanism for enforcing policy constraints. We then describe two schemes for placing replicas: a static constraint-based scheme, and a dynamic scheme based on the datas' access pattern.

4.1 Policy constraints

Policy constraints result from legal dictates, availability needs, and other application requirements. In our approach, when creating a table, the developer specifies policy constraints as a set of rules applying to some or all records in the table. A rule consists of a predicate that defines the table and affected records, a priority, and settings for one or more of the following properties:

- `MIN_COPIES`: The minimum number of full replicas of the record that must exist.
- `INCL_LIST`: An inclusion list—the locations where a full replica of the record must exist.
- `EXCL_LIST`: An exclusion list—the locations where a full replica of the record cannot exist.

Rule priority is used when two rules apply to the same record; the rule with the highest priority is given precedence for properties defined in both rules. A full grammar for the language is given in Appendix A.2. An example rule is:

Rule 1:
```
IF
   TABLE_NAME = "Users"
THEN
   SET 'MIN_COPIES' = 2
CONSTRAINT_PRI = 0
```

This *table-level* rule specifies that all records in the "Users" table must have at least two full replicas. An example of a *record-level* rule is:

Rule 2:
```
IF
   TABLE_NAME = "Users" AND
   FIELD_STR('home_location') = 'France'
THEN
   SET 'MIN_COPIES' = 3 AND
   SET 'EXCL_LIST' = 'USWest,USEast'
CONSTRAINT_PRI = 1
```

This rule applies only to "Users" records that have value "France" for the "home_location" field. Matching records must have at least three full replicas, but full replicas cannot be placed in the USWest or USEast datacenters. Some records may match both Rules 1 and 2. For such records, we use the `MIN_COPIES` property from Rule 2, since it has higher priority than Rule 1. Consider a third example rule:

Rule 3:
```
IF
   TABLE_NAME = "Users" AND
   FIELD_STR('home_location') = 'India'
THEN
   SET 'INCL_LIST' = 'India'
CONSTRAINT_PRI = 2
```

This rule specifies that records with `home_location='India'` must be stored in the India datacenter. Such records will also be required to have two full replicas, since Rule 3 does not override the value of `MIN_COPIES` from Rule 1.

In our system, constraints are enforced on a *repair* basis—the system makes a best effort to repair a constraint that is violated. For example, if a record is inserted into a table with (`MIN_COPIES=2`), the database stores the record at one location and returns success. But because the system is not satisfying the constraint, it takes action to make a second copy elsewhere. Similarly, if there is a failure so that a copy is lost and a record no longer meets the minimum copies constraint, the system must make an extra copy of the record. If a constraint rule changes, the system may have to change the replication of a large number of records to conform to the new rule. This is potentially expensive, and thus in our prototype, constraint changes are not allowed after data is inserted into a new table.

A constraint rule is *valid* if it can be satisfied. For example, if a rule specifies the same location in both `INCL_LIST` and `EXCL_LIST`, then it is impossible to satisfy the rule. We formalize and discuss the constraint checking problem in Appendix A.3. Ultimately, a valid rule generates a `MIN_COPIES`, `INCL_LIST`, and `EXCL_LIST` that can be mutually satisfied.

4.2 Static constraint-based data placement

A static constraint-based placement policy chooses where to place records based on the values for the record's data fields. We can define a function `choose_replicas(R,C)` that takes a record R and a set of constraints $\mathbf{C}$, and chooses a set of locations for full replicas which satisfy $\mathbf{C}$. This function could choose locations randomly, or use heuristics to choose, such as preferring locations near the user's `home_location`.

The master makes an initial placement decision when the record is inserted. If the record is updated such that it matches different constraints, the master may potentially have to change the placement of the data to ensure constraints are not violated. Furthermore, on update, the heuristics in `choose_replicas()` may choose a different location.

Consider the insert of a record R. We define $stub(R)$ as the stub version of R, that is, R without any data values. Assume $\mathbf{C}$ is the set of constraints that have been defined, and L is the set of all possible replica locations. Algorithm 1 shows the steps taken on insert.

Algorithm 1 Static constraint-based policy, on insert

Set $\mathbf{P}$ = `choose_replicas(`R,$\mathbf{C}$`)`
Send R to all $p \in \mathbf{P}$.
Send $stub(R)$ to all $l \in \mathbf{L}$.

R and $stub(R)$ are published to the messaging layer in a single transaction (a special case that PNUTS' messaging layer handles). Since successfully published messages are guaranteed to be delivered, an insert either succeeds and

places a record or stub in every region, or fails and writes nothing. It is possible to achieve correctness without messaging layer transactions; we omit those details here.

If there is an update of R such that it now matches different constraint rules or so that `choose_replicas()` picks different locations, then the master may need to change the set of full replicas. In this case, the master must send the record to the new full replica regions, and stubs with the new full replica list to all of the regions. Old full replica location that are no longer listed in the full replica list of the stubs will demote their copy to a stub (discarding data values.) A full algorithm is given in Appendix A.4.

4.3 Dynamic placement

One disadvantage of the static constraint-based scheme is that it is insensitive to access patterns. Although the heuristics in `choose_replicas()` can be sophisticated enough to deal with broad access characteristics, it is difficult to tune them for fine-grained access patterns. For example, if a user in France has lots of friends in Asia, but this is not contemplated by the heuristic, the system may decide not to make full replicas in Asia even though doing so would improve response time. Furthermore, while static constraint-based adapts to changes in the data records, it does not adapt to changes in the access patterns.

We considered a replication scheme that gathered read and write statistics (as in [25]) in order to make dynamic replica placement decisions. However, access patterns vary from user to user and thus from record to record, requiring bookkeeping at the per-record level. Efficiently acquiring, tracking and collecting access statistics from around the world is a complex and expensive process. In particular, storing per-record statistics on disk requires significant disk I/O to maintain these statistics; and maintaining fine-grained statistics in memory takes up valuable cache space. Furthermore, communicating statistics to the master to make placement decisions, or making such decisions in a distributed manner while enforcing constraints, adds significant complexity to the system.

Instead, our approach is to make a full replica when we see a read at a stub location, and demote a full replica when we see a write at a different location. (There is a retention interval during which we don't demote a newly created replica, as we discuss below.) This approach, called *dynamic placement*, is based on the idea that if a record is read from a location, it is likely to be read again. For example, a user session might consist of multiple reads in a row. At the same time, if we see an update for a record that has not been read recently, then likely the user session is over and we should demote the full replica to avoid paying the cost of further propagated updates. The local replica location requests promotion or demotion, but the master decides whether to grant the request in order to ensure policy constraints are enforced. Although dynamic placement is heuristic, it takes better account of access patterns than static constraint-based. At the same time, it avoids the expensive bookkeeping of more accurate tracking of statistics.

A key aspect of our approach is a *retention interval*: after a location makes a new full replica, it retains that full replica for at least some interval I. During I, updates are applied to the replica, and it is served to local readers. Once the interval expires, crucially, we do not take immediate action. We wait for the next operation: if the next operation is a read we still have an up-to-date replica stored locally. We serve it, and then retain the full replica for another interval I. If the next operation is a write, we demote the full replica to a stub. Thus, we demote replicas only when we have not seen a read for a while (since the retention interval expired and was not renewed before the location saw the write). If no reads or writes occur for a while after the interval expires, the full replica lingers, but this does no harm since there are no updates to consume bandwidth.

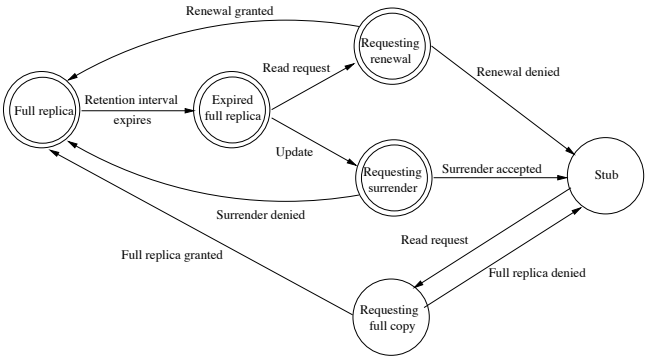

Figure 2: State diagram for dynamic placement, executed by each location. Double circles indicate location has a full replica; single circles indicate a stub.

When a record is inserted, dynamic follows the same steps as static constraint-based (Algorithm 1) to seed an initial set of full replicas. Then, dynamic placement promotes and demotes records based on the read and write pattern. Even if the initial full replicas are chosen poorly, the adaptive nature of our scheme migrates full replicas to the locations where read rates are high compared to update rates.

Figure 2 shows the states a location can be in with respect to a given record. When the retention interval of a full replica expires, the location decides whether to try to retain the full replica, or demote it to a stub, based on the next operation seen. The location makes a local decision but then confirms this decision with the master, who either grants or denies the request. Similarly, a stub record can request promotion to a full record, but must gain the master's approval to do so. Detailed algorithms for converting between stub and full replica are provided in Appendix A.5.

4.3.1 Retention interval configuration

The length of the retention interval I is a key parameter. If the interval is too short, locations will be quick to surrender full replicas, as they are likely to have an expired full replica when an update arrives. In a write-heavy workload, this could result in high cost, as even a desirable full replica may frequently be surrendered, only to be acquired again on the next read. In contrast, an interval that is too long means that a single read can cause a full replica to be retained for a long time, costing bandwidth as we apply repeated updates, even if there is not another read. We have had success tuning I both through workload simulation and adjusting it in PNUTS itself.

4.3.2 Latency constraints

Dynamic placement attempts to minimize bandwidth usage by creating full replicas where reads outnumber writes, and stubs in other locations. However, it may be necessary

to make extra full replicas to ensure the latency SLA is met. (The SLA is enforced using monitoring by operations engineers; see Appendix A.1 for more details.) There are two ways to accomplish this goal. The most direct way is to increase the `MIN_COPIES` constraint. Increasing `MIN_COPIES` is a blunt instrument, since extra full replicas will be made of all records, regardless of whether those records are read frequently in the new locations. The finer approach is to increase the retention interval I. Then, full replicas will be less likely to be demoted to stubs, meaning that more often, reads will find a full replica at a location instead of a stub.

5. EVALUATION

We evaluate our selective replication techniques using a modified version of the production PNUTS codebase installed in three datacenters in the U.S., India and Singapore. We focus on two primary metrics:

- **bandwidth** - bytes transferred between datacenters for replication and forwarded reads.
- **latency** - average read latency.

We compared five replication schemes:

- `Full`: replicate all records to all locations
- `Static constraint-based` (`SCB`) : static placement of full replicas that respects constraints (Section 4.2).
- `Dynamic constraint-based` (`DCB`) : dynamic placement of full replicas that respects constraints (Section 4.3).
- `Dynamic`: dynamic placement with no constraints.
- `Bandwidth optimal` (`BWO`) : assuming perfect future knowledge, full replicas placed statically to minimize bandwidth while satisfying policy constraints. We provide this algorithm as a "lower-bound."

We examined `Dynamic` (without constraints) to understand the behavior of that technique independently, even though a production database would likely have constraints. `BWO` is constructed using a knapsack algorithm to select full replicas based on the operations trace used in our experiments; see Appendix A.6 for details.

Our experiments show that simple dynamic schemes make good replication decisions in most cases despite keeping no statistics. However, when there is little geographic locality of access, or a very high write rate, dynamic is less effective, and the cost of adaptively promoting and demoting replicas can outweigh any savings from selective replication.

5.1 Experimental setup

Workload—Our workload models a social networking application in which the system retrieves the latest profiles for a user's friends each time the user logs in. We generated a series of synthetic data sets and read/write traces in order to directly vary different parameters of the workload. However, we validated that the distribution of logins and friend counts (both Zipfian) matched real workloads from a Yahoo! social application and the Twitter follows graph (these data sets are described in [23]). We also ran one experiment using a real trace extracted from the Yahoo application.

Each user is given a "home location" where their reads and writes originate. In some experiments, this location is fixed, and in others, it is allowed to vary. When a user logs in the application reads their friends' profile records, and with some probability, updates the user's profile record. The likelihood that a given friend has the same home location as the user is controlled by the *remote probability* parameter (default=0.1); as we increase this probability more of a user's friends are assigned a home location different than the user (necessitating a forwarded read if the friend's record is only a stub in the local datacenter). We also vary the proportion of reads and writes (default: 9 reads/1 write) and the size of reads and writes (default: 100 bytes for both).

Configuration—We set up PNUTS clusters in datacenters in the United States, India and Singapore, each a minimal setup with 1 tablet controller, 1 router, and 1 storage unit. We deployed Hedwig [2] as the pub/sub system to handle cross-data center replication. While the per-data center setup is small compared to production PNUTS, our focus here is on measuring inter-data center bandwidth and latency, rather than intra-data center. PNUTS' scalability is discussed elsewhere [9]. Our dataset consists of 100,000 1 KB user records. We ran 5 million read or write operations for each data point. Though the number of users is small for PNUTS, this lets us execute a large number of operations on each user. All policies benefit equally from the fact that data fits in memory; we observe that in practice social data is increasingly being served from RAM. For dynamic schemes, we generated a trace with 6 million read or write operations and used the first 1 million operations as a warm-up workload that generated an initial placement of full replicas; for other policies, full replicas and stubs were made on insert. *Note our methodology favors static placement in that all full replicas are created in the initial population phase, and this bandwidth cost does not count toward the results.* In contrast, dynamic schemes still create some replicas even after the warm-up phase ends. Our experimental results show that in most cases dynamic still outperforms static.

For constraint schemes, we specified two constraints: each record must have a full replica at the user's home location, and each record must have an additional full replica. For `SCB` and `DCB`, the second full replica (besides the one at the home location) was placed randomly. For `Dynamic`, a single full replica is created initially. Note that even though our constraint language is very flexible, our experiments show that even simple constraints can be very effective.

Dynamic schemes can be tuned to provide better latency by changing the retention interval. A longer interval means that locations hold full replicas longer, and hence proportionally more reads are served by a local copy. In production, if the application is experiencing unacceptable latency, retention interval can be increased to meet the latency SLA. We used a retention interval of 300 seconds. We experimented with a variety of intervals and chose this interval because it offered low bandwidth usage and reasonable latency; furthermore, it was short enough to quickly adapt to changes in access patterns.

5.2 Varying read/write ratio

We examine the impact of the mix of reads and writes on each policy. Figure 5 shows that as the proportion of writes increases, the bandwidth cost increases proportionally for all policies. Since `Full` must send updates to three copies, its bandwidth consumption increases fastest. `Dynamic`, however, can keep as few as one copy (adaptively placed), so its bandwidth cost increases most slowly. The remaining three policies all have the constraint of keeping a minimum of two copies and their bandwidth costs increase at rates be-

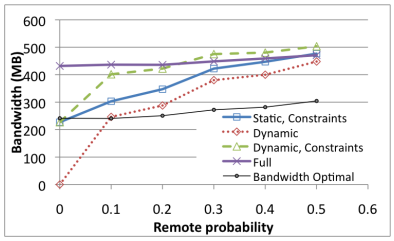

Figure 3: Bandwidth used as remote friend % increases.

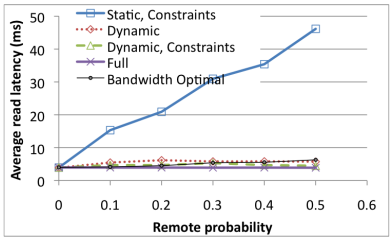

Figure 4: Request latency as remote friend % increases.

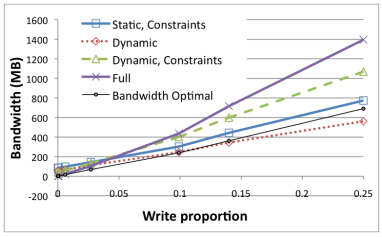

Figure 5: Bandwidth as write % increases.

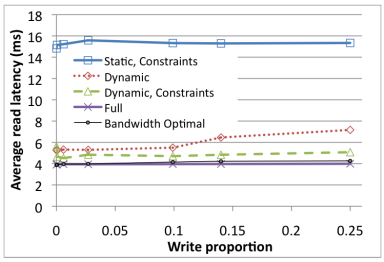

Figure 6: Latency as write % increases.

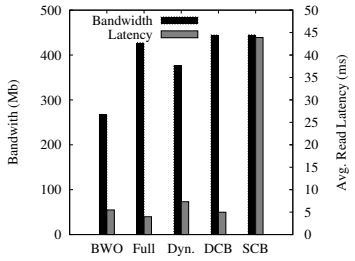

Figure 7: Impact of record access pattern shifts.

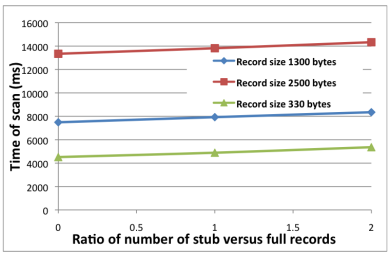

Figure 8: Impact of stubs on scan performance.

tween `Full` and `Dynamic`. The extra bandwidth overhead for leases makes `DCB` consume more bandwidth than the other two policies as the proportion of writes increases.

This result has significant implications for cloud database systems. In particular, increasing write traffic may eventually saturate the messaging layer, forcing us to scale it out with more hardware. For a fixed size messaging layer deployment, dynamic schemes send fewer messages per write, and so can support an overall higher write rate.

As shown in Figure 6, the latency of the dynamic schemes increase as write proportion increases, while the other policies experience little change in latency. With dynamic placement, as the write proportion increases, the likelihood increases that an update reaches an expired full replica and causes the demotion of that replica to a stub. The result is fewer full replicas, increasing overall latency. `DCB` must maintain two copies and hence has a better latency than `Dynamic`. In the other schemes, increased write rate leads to increased system load and a slight increase in latency, but does not change placement decisions.

We ran a similar experiment where we increased the relative size of read and write requests, while fixing the read-to-write ratio. Detailed results are omitted. Larger writes increased the bandwidth used for all policies. Latency, however, was unaffected for all policies. Since the number of write operations did not increase, dynamic schemes did not make different placement decisions.

5.3 Impact of locality

Next we examined the impact of locality on the performance of different replication schemes. In this experiment, we varied the *remote probability* from 0 to 0.5; at 0.5 on average one half of a user's friends have a remote home location. Figure 3 shows the bandwidth used for each policy. The bandwidth for `Full` is constant, since updates always propagate to all three locations, and all reads are local. For other policies, bandwidth usage increases with increasing remote probability. As remote probability increases, there is less geographic locality of access, and all schemes must do increasing numbers of forwarded reads (costing bandwidth).

Other than `BWO`, `Dynamic` provides the lowest bandwidth, as it adaptively places full replicas to achieve the best bandwidth usage. (When remote probability is zero, `Dynamic` has lower bandwidth than `BWO` because it makes only one copy, at the user's home location; while `BWO` makes two copies to meet the constraint). When constraints are enforced, `DCB` is worse than `SCB`. As an adaptive scheme, `DCB` has some cost overhead because of control messages sent when promoting or demoting replicas, and the transfer of a full record when promoting a replica.

Figure 4 shows latency measurements. `Dynamic` and `DCB` provide significantly better latency than `SCB`, and almost match `BWO`. Because `Dynamic` aligns full replicas with the locations where records tend to be read, most accesses are local, and it is easier to meet a latency SLA. `Dynamic` has slightly worse latency than `DCB`, whose constraints force the system to make extra copies, increasing the number of locally-served reads.

5.4 Changing access patterns

We examine a scenario where user access patterns shift. During this experiment we changed the home location of 20% of the users, changing locality both for users reading their own records and for users reading their friends' records. The Figure 7 dark (left side) bars show the results for bandwidth. `Dynamic` performs better than any policy other than `BWO`. The Figure 7 light (right side) bars show latency results. Though `Dynamic` has almost twice the latency of `Full`, the latency for `DCB` is comparable to `Full`. The dynamic schemes are an order of magnitude better on latency than the static constraint-based scheme. While `Full` seems a good choice in this setting, note that a 20% access pattern shift in a short amount of time is not likely; dynamic placement is superior for minimizing bandwidth over the long run.

5.5 Real data trace

We evaluated our approach on a data set drawn from the logs of a Yahoo! social application, similar to our synthetic one. A logged write adds some social content to the writing user's record. A logged read gets social content from a user's record. We sampled from 10 days of logs to produce about 170,000 unique users, and then captured all appearances of these users in the following 10 days. This results in a trace of about 32 million operations; we warm-up on the first 10 days and report results from the second. The trace is very read-heavy; only 0.06% of operations are writes. About 40% operations are remote. With the tiny percentage of writes and large percentage of remote reads, this trace is actually very favorable to `Full`. We found that `Dynamic` and `DCB` get similar excellent average read latencies (about 4ms) as `Full`. The total bandwidth for `Full` is only 8 Mb due to all reads being served locally and the small number of writes in the trace. The total bandwidth is 15 Mb and 6.8 Mb for `Dynamic` and `DCB` (MIN_COPY = 2) respectively. While we expect `Dynamic` to have a hard time beating `Full` with this trace, it is a nice surprise to see `DCB` actually consumes less bandwidth and gets similar latency to `Full`. `SCB` does not work well in this experiment, with average read latency of 14 ms and bandwidth of 183 Mb. While `Full` is effective in this experiment, we see that `Dynamic` and `DCB` are effective as well.

5.6 Scan performance

A stub record contains a record key and metadata information. The semantics we have chosen based on real customer scenarios for table scan is to skip stubs and only return full records. Though stubs are small, if a table contains small records and/or a large number of stubs, they may create a perceived slowdown in scan performance. In this experiment, we measure time to scan an entire table in a selected region by keeping a fixed number of full records (33333 rows) while increasing the number of stub records. Figure 8 shows scan time as we vary the number of stubs, and as we vary full record size. Stub size stays a constant 112 bytes. As expected, the larger the stub footprint, the more time scan spends on them. Nevertheless, in all cases the stub overhead is less than 10%, even when records are not much larger than stubs. The impact of stubs on scan performance appears to be insignificant.

6. CONCLUSIONS

We have proposed a mechanism for selectively replicating data at a record granularity while respecting policy constraints. We examined a dynamic placement scheme that achieves efficient placement of data with small bookkeeping overhead. Experimental results using PNUTS demonstrate that our techniques provide significant improvement in bandwidth usage between datacenters compared to full replication. Moreover, our adaptive dynamic placement scheme is tunable in order to meet latency constraints.

7. REFERENCES

[1] Akamai. http://www.akamai.com.

[2] Hedwig. https://cwiki.apache.org/confluence/display/ZOOKEEPER/HedWig.

[3] MySQL 5.0. http://dev.mysql.com/doc/refman/5.0/-en/replication.html.

[4] K. Amiri et al. Dbproxy: A dynamic data cache for web appplications. In *ICDE*, pages 821–831, 2003.

[5] S. Annapureddy et al. Shark: Scaling file servers via cooperative caching. In *NSDI*, 2005.

[6] P. Apers. Data allocation in distributed DBMS. *ACM TODS*, 13(3):263–304, September 1988.

[7] M. Cafarella et al. Data management projects at Google. *SIGMOD Record*, 34–38(1), March 2008.

[8] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM*, pages 177–190, 2002.

[9] B. F. Cooper et al. PNUTS: Yahoo!s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[10] C. Curino et al. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, pages 48–57, 2010.

[11] S. Dar et al. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[12] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY USA, 1979.

[14] A. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[15] M. Korupolu et al. Placement algorithms for hierarchical cooperative caching. In *Symposium on Discrete Algorithms*, pages 586–595, 1999.

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

[17] D. Kossmann, M. J. Franklin, and G. Drasch. Cache investment: Integrating query optimization and distributed data placement. *ACM TODS*, 25(4):517–558, December 2000.

[18] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A structured storage system on a P2P network. In *SIGMOD*, 2008.

[19] W. Litwin and T. J. E. Schwarz. Lh*rs: A high-availability scalable distributed data structure using reed solomon codes. In *SIGMOD*, pages 237–248, 2000.

[20] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, CA USA, 1994.

[21] L. Ramaswamy et al. Cache clouds: Cooperative caching of dynamic documents in edge networks. In *ICDCS*, pages 229–238, 2005.

[22] J. Sidell et al. Data replication in Mariposa. In *ICDE*, pages 485–495, 1996.

[23] A. Silberstein et al. Feeding Frenzy: Selectively materializing users event feeds. In *SIGMOD*, pages 831–842, 2010.

[24] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a WAN. In *PODC*, pages 134–143, 2001.

[25] O. Wolfson et al. An adaptive data replication algorithm. *ACM TODS.*, 22(2):255–314, 1997.

[26] H. Yu and A. Vahdat. Minimal cost replication for availability. In *PODC*, pages 98–107, 2002.

APPENDIX

A.1 Latency SLA

In PNUTS, the latency SLA is specified as a soft constraint: a target SLA is specified, and operations engineers monitor the performance of the system to ensure the SLA is met. If the SLA is violated on a recurring basis, then steps must be taken to reduce latency. In the production system (which does not have selective replication yet), the primary remedy is to allocate more servers to reduce overall load and hence latency. Selective replication introduces the possibility that the latency SLA might be violated not just by load, but also by the system's decision to make stubs of some records in some locations. In this case, the remedy is to either increase the retention interval (as described in Section 4.1) or to increase the `MIN_COPIES` constraint.

There are multiple possible ways to specify the latency constraint. One possibility is to specify the target latency for some percentile of users. For example:

"Read latency must be 100 ms or less for 95% of users"

However, the database sees only queries, and since one user action may generate multiple queries, it is hard to correlate queries to end users. Thus it is hard to monitor this type of SLA. A second possibility is to specify a target latency for individual operations:

"Read latency must be 50 ms or less for 95% of read operations"

This SLA is easier to monitor. One disadvantage is that a violation of the SLA could be attributable to multiple causes: bad decisions by the selective replication mechanism; overloaded servers; network congestion, and so on. For this reason, it might be useful to additionally specify a finer-grained SLA that directly targets selective replication, such as:

"95% of read operations should be served by a local, full replica"

If this SLA is violated, it is easy for the operations engineers to know that it is necessary to tune the selective replication policy, for example by increasing the retention interval.

A.2 Constraints language

We have developed a simple domain specific language for specifying constraints. The language allows the constraints to be specified by application developers and associated with new tables.

A.2.1 Example

We now present a more complex example of a constraint rule. In this example, imagine that the U.K. has laws governing data transmission, and personally identifiable information about U.K. users can only be replicated to the U.S. with the user's explicit permission. Rule 4 captures this policy: if the user's `home_location` is `'UK'` but they have not agreed to a U.S. copy, then the U.S. datacenters must be included in the exclusion list.

Rule 4:
```
IF
 TABLE_NAME = "Users" AND
 FIELD_STR('home_location') = 'UK'
 FIELD_STR('agreed_us_copy') = 'false'
THEN
 SET 'MIN_COPIES' = 2
 SET 'INCL_LIST' = 'uk'
 SET 'EXCL_LIST' = 'usw,use,usc'
CONSTRAINT_PRI = 0
```

A.2.2 Constraints language grammar

The allowable constraint rules are defined by the following grammar.

Constraints language

constraint :== "IF" **condition** "THEN" **property** **constraint_priority**
condition :== { (**table_specifier** ["AND" **predicate**]) | (**predicate** "AND" **table_specifier** [("AND" | "OR") **predicate**]) }
constraint_priority :== "CONSTRAINT_PRI" "=" **integer_literal**
table_specifier :== "TABLE_NAME" "=" **table_name**
table_name :== **string_literal**
property :== "SET" **parameter** "=" **value** ["AND" **property**]
parameter :== **string_literal**
value :== **string_literal** | **integer_literal**
string_literal :== a single quoted string
predicate :== **expression**
expression :== **term** [{ "AND" | "OR" } **term** ...]
term :== **compare_clause** | **group**
group :== "(" **expression** ")" | "NOT" **expression**
compare_clause :== **var_op_clause** | **var_null_clause** | **var_regexp_clause**
var_op_clause :== { **field** | **value** } **op** { **field** | **value** }
op :== "<" | "<=" | "=" | "==" | "!=" | ">" | ">="
var_null_clause :== **field** "IS" ["NOT"] "NULL"
var_regexp_clause :== **field_str** "REGEXP" **string_literal**
value :== **string_literal** | **integer_literal**
string_literal :== a single quoted string
field :== **field_int** | **field_str**
field_int :== "field_int(" **string_literal** ")"
field_str :== "field_str(" **string_literal** ")"

A.3 Constraint validity

Section 4.1 states that a record must be subject to a valid constraint rule. We now formally define validity. Consider the set of all possible replica locations $\mathbf{L}$. Given a constraint rule C, define $mc(C)$ as the number of `MIN_COPIES` defined by C (if any); $incl(C)$ as the set of replica locations defined in C's `INCL_LIST` clause (if any); and $excl(C)$ as the set of replica locations defined in C's `EXCL_LIST` clause (if any). Then, C is *valid* if:

- $\exists mc(C) \rightarrow 1 \leq mc(C) \leq |\mathbf{L}|$
- $\exists incl(C) \rightarrow incl(C) \subseteq \mathbf{L}$
- $\exists excl(C) \rightarrow excl(C) \subset \mathbf{L}$
- $\exists excl(C) \wedge \exists incl(C) \rightarrow excl(C) \cap incl(C) = \emptyset$
- $\exists excl(C) \wedge \exists mc(C) \rightarrow mc(C) \leq |\mathbf{L}| - |excl(C)|$

In this notation, we use $\rightarrow$ to mean "implies."

At run time a given record may match multiple constraints. Unless we know the allowable combinations of values in

records, it is impossible to determine at rule compilation time whether a particular record will match the predicates of multiple conflicting rules. To avoid invalid runtime combinations of properties, we apply at most two rules to a record: the highest priority table-level rule (if one matches), and the highest priority record-level rule (if one matches). Then, we may combine properties from these two rules to determine the constraints on a record. It is straightforward to determine at rule compile time whether a record-level rule could cause a validity conflict with a table-level rule, and raise an error if so.

A.4 Static constraint-based placement algorithms

Algorithm 2 describes how the system reacts to an update of a record.

Algorithm 2 Static constraint-based policy, on update

```
Set R=Record before the update
Set R'=Record after the update
Set C=Constraints matched by R before update
Set C'=Constraints matched by R' after update
Set P = choose_replicas(R,C)
Set P' = choose_replicas(R',C')
if P ≠ P' then
  Set Add = P' - P
  for all a ∈Add do
    Send R' to a.
  end for
  for all l ∈L do
    Send stub(R') to l.
  end for
end if
```

A.5 Dynamic placement algorithms

Algorithms 3–6 are carried out by a remote location when managing full replicas and stubs; these algorithms represent the state transitions illustrated in Figure 2.

Algorithm 3 Manage full replica

```
for Interval I do
  Retain record R, applying updates and serving reads.
end for
if Next operation == Read then
  Execute Algorithm 4 "Renew replica"
else {Operation is an update from master}
  Execute Algorithm 5 "Surrender replica"
end if
```

Algorithm 4 Renew replica

```
Send renewal request message to master
Master examines constraint rules C to determine if the
request can be granted.
if Master grants renewal then
  Execute Algorithm 3 "Manage full replica"
else {Master denies renewal}
  Convert replica to stub (discarding data values)
  Execute Algorithm 6 "Manage stub"
end if
```

Algorithm 5 Surrender replica

```
Send surrender request message to master
Master examines constraint rules C to determine if the
request can be granted.
if Master grants surrender then
  Convert replica to stub (discarding data values)
  Execute Algorithm 6 "Manage stub"
else
  Execute Algorithm 3 "Manage full replica"
end if
```

Algorithm 6 Manage stub

```
if Read request received then
  Send message to master requesting promotion from stub
  to full replica
  Master examines constraint rules C to determine if the
  request can be granted.
  if Master grants promotion then
    Convert stub to replica (storing data values)
    Execute Algorithm 3 "Manage full replica"
  else {Master denies promotion}
    Execute Algorithm 6 "Manage stub"
  end if
end if
```

A.6 Optimal static placement

If we had perfect knowledge about access patterns, where would we place full replicas? Consider a record R_i and a location L_j. We can define the following characteristics of R_i, which reflect our perfect knowledge of the access pattern:

- $UR(R_i)$: average update rate of R_i (in operations/sec)
- $US(R_i)$: average size of an update to R_i (in bytes/operation)
- $RR(R_i, L_j)$: average read rate of R_i from L_j (in operations/sec)
- $RS(R_i, L_j)$: average read size of R_i from L_j (in bytes/operation)

If we make a full replica of R_i at L_j, then we must pay bandwidth costs to propagate updates to L_j. The rate of bandwidth cost is $UR(R_i) \times US(R_i)$. If, instead, we decide to make only a stub at L_j, then every time there is a read of R_i at L_j, we must forward that read to a full replica. The rate of bandwidth costs incurred for forwarded reads is $RR(R_i, L_j) \times RS(R_i, L_j)$. Thus, the benefit $B(R_i, L_j)$ of choosing a stub over a full replica is a savings in bandwidth usage equal to $B(R_i, L_j) = UR(R_i) \times US(R_i) - RR(R_i, L_j) \times RS(R_i, L_j)$. Recall, we are constrained by the latency SLA, which says that some fraction of reads must be served locally. If B is negative, there is no bandwidth savings to making a stub, and we might as well make a full replica of R_i at L_j to ensure the reads are local. If B is positive, we might decide to make R_i a stub at L_j, as long as we do not violate the latency SLA.

Formally, we define an indicator variable F_{R_i,L_j} to be 1 if there is a full replica of R_i at L_j, and 0 if there is a stub of R_i at L_j. Define SLA to be the fraction of reads which must be served from local full replicas. Then, we can express the

latency constraint on placement as follows:

$$\sum_{i,j} RR(R_i, L_j) \times F_{R_i,L_j} \geq SLA \times \sum_{i,j} RR(R_i, L_j)$$

In other words, the total number of local reads across all locations is the sum of reads that are initiated at locations where $F_{R_i,L_j} = 1$. This total number of local reads must be at least the total number of reads times the SLA fraction. Our goal is to minimize bandwidth usage; or, equivalently, maximize bandwidth savings, where bandwidth savings equals

$$\sum_{i,j} B(R_i, L_j)$$

Imagine all replicas were full replicas, and we were allowed to demote some full replicas to stubs to attain bandwidth savings. This results in an integer knapsack problem, where the volume of the knapsack is the allowed non-local reads $(1 - SLA \times \sum_{i,j} RR(R_i, L_j))$. The items we pack into the knapsack are full replicas we make into stubs; the volume of each item is $RR(R_i, L_j)$ and the value of each item is $B(R_i, L_j)$. Integer knapsack problems are generally NP-hard [13]. However, in web databases there are a huge number of records, each with relatively small volume. Thus, we can approximate this problem as a fractional knapsack problem without too much error.

The fractional knapsack problem can be solved easily by sorting all replicas in descending order of $B(R_i, L_j)$, and converting full replicas to stubs in that order until the total number of forwarded reads reaches the constraint (e.g. the knapsack is full.) Of course, we have to respect policy constraints (such as minimum copies or `INCL_LIST`). Furthermore, we should not convert a full replica to a stub if doing so increases cost. So, as we are stepping through the descending order of replicas, we should not convert a full replica to a stub if doing so violates a policy constraint or if $B(R_i, L_j) \leq 0$. The result is the approximately optimal static placement of full replicas that respects policy and latency constraints; it is approximate because of the conversion from an integer to fractional knapsack problem.

A.7 In the Wild

This paper presents a selective replication mechanism with an adaptive dynamic placement scheme and a constraint language. In moving from our first prototype to production implementation, we encountered additional challenges:

Selective Replication and Eventual Consistency: PNUTS [9] supports timeline consistency via record-level mastership, and delivery of updates published by a record's master to the other replicas in the same order; this is the semantics we have primarily focused on when considering selective replication. PNUTS had added a second table type without record-level mastership to increase availability, giving up timeline consistency for eventual consistency. An application can apply concurrent updates to different replicas of the same record in parallel. PNUTS publishes the changes asynchronously to other replicas and resolves conflicts using the local timestamp of each write (latest wins). All replicas eventually receive all changes to the same record and resolve conflicts identically. Selective replication, however, imposes a new challenge: updates are not published to stubs, and this may cause a replica to *not* eventually receive all changes to a record. This can happen when a write and replica promotion occur concurrently: the newly promoted stub-to-replica may miss the data change. To address this issue, we require a full replica to republish its write after detecting overlapping promotions for other replicas of the same record.

Semantics of Stubs and Scans: A major concern that became apparent in practice is the impact of stubs on scan performance and semantics, especially because we avoid exposing the notion of stubs to clients as much as possible. Section 5.6 examines a semantics where stubs are skipped in scan results (i.e., only full replicas at the local site are returned), and shows that the presence of stubs (even when they outnumber full replicas) has only a slight effect on scan time. An alternate semantics is to return all records in the table, even those represented by stubs, during scans. This clearly is detrimental to scan time, taking a local operation that otherwise relies on sequential I/O and turning it into one that must regularly pause to fetch remote replicas. In practice, we do not support this semantics, and instead direct clients to a region that has a full copy of the table.

Data Co-location: It is often desirable to co-locate data items that are processed together. Selective replication provides a flexible mechanism to co-locate relevant data items without requiring fully replicating them in all locations. Our constraint language provides a flexible solution to this problem. For example, one could specify that all data records pertaining to a given group of users be in certain locations, while data for other users (even from the same tables) could be elsewhere. Even with this flexibility, developers want to be able to easily control selective replications for a particular records. Our implementation therefore supports promotion/demotion of a replica in the record insert/update interface.

In ongoing work we are also evaluating the cost and effectiveness of more complex constraint types, as well as other adaptive schemes that build on dynamic placement.