On the expressiveness and trade-offs of large scale tuple stores

Ricardo Vilaça, Francisco Cruz *, and Rui Oliveira

Computer Science and Technology Center
Universidade do Minho
Braga, Portugal
{rmvilaca,fmcruz,rco}@di.uminho.pt

Abstract. Massive-scale distributed computing is a challenge at our doorstep. The current exponential growth of data calls for massive-scale capabilities of storage and processing. This is being acknowledged by several major Internet players embracing the cloud computing model and offering first generation distributed tuple stores.
Having all started from similar requirements, these systems ended up providing a similar service: A simple tuple store interface, that allows applications to insert, query, and remove individual elements. Furthermore, while availability is commonly assumed to be sustained by the massive scale itself, data consistency and freshness is usually severely hindered. By doing so, these services focus on a specific narrow trade-off between consistency, availability, performance, scale, and migration cost, that is much less attractive to common business needs.
In this paper we introduce DataDroplets, a novel tuple store that shifts the current trade-off towards the needs of common business users, providing additional consistency guarantees and higher level data processing primitives smoothing the migration path for existing applications. We present a detailed comparison between DataDroplets and existing systems regarding their data model, architecture and trade-offs. Preliminary results of the system's performance under a realistic workload are also presented.

Keywords: Peer-to-Peer; DHT; Cloud Computing; Dependability

1 Introduction

Storage of digital data has reached unprecedented levels with the ever increasing demand for information in electronic formats by individuals and organizations, ranging from the disposal of traditional storage media for music, photos and movies, to the rise of massive applications such as social networking platforms.

Until now, relational database management systems (RDBMS) have been the key technology to store and process structured data. However, these systems

* Partially funded by project Pastramy – Persistent and highly available software transactional memory (PTDC/EIA/72405/2006).

based on highly centralized and rigid architectures are facing a conundrum: The volume of data currently quadruples every eighteen months while the available performance per processor only doubles in the same time period [22]. This is the breeding ground for a new generation of elastic data management solutions, that can scale both in the sheer volume of data that can be held but also in how required resources can be provisioned dynamically and incrementally[8,6,5,17]. Furthermore, the underlying business model supporting these efforts requires the ability to simultaneously serve and adapt to multiple tenants with diverse performance and dependability requirements, which add to the complexity of the whole system. These first generation remote storage services are built by major Internet players, like Google, Amazon, Microsoft, Facebook and Yahoo, by embracing the cloud computing model.

Cloud data management solutions rely on distributed systems designed from the beginning to be elastic and highly available. The CAP theorem [4] states that under network partitions it is impossible to achieve both strong consistency and availability. Cloud data management solutions acknowledge these difficulties and seek to establish reasonable trade-offs. Some focus on applications that have minor consistency requirements and can thus favor availability. They replace the traditional transactional serializability [2] or linearizability [14] strict criteria by eventual consistency (e.g. Basically Available Soft-state Eventual consistency [9]) or even explicit out of the system conflict resolution [8]. Furthermore, they provide only single tuple operations or at most range operations over tuples.

By doing so, these services focus on a specific narrow trade-off between consistency, availability, performance, scale, and migration cost, that fits tightly their motivating very large application scenarios. They focus on applications that have minor consistency requirements and can favor availability with an increasing complexity at the application logic. In most enterprises, in which there isn't a large in-house research development team for application customization and maintenance, it is hard to add this complex layer to the application. Moreover, these applications typically have queries that, in addition to single tuple and range operations, have the need for multi-tuple operations, and require the usual, standard, more consistent data management. As a result, it is hard to provide a smooth migration path for existing applications, even when using modern Web-based multi-tier architectures. This is a hurdle to the adoption of Cloud Computing by a wider potential market and thus a limitation to the long term profitability of businesses model.

In this paper we present DataDroplets that skews the current trade-off towards the needs of common business users. DataDroplets provides additional consistency guarantees and higher level data processing primitives that ease the migration from current RDBMS. In order to provide higher level data processing primitives, DataDroplets extends first generation remote storage services data models with tags and multi-tuple access that allow to efficiently store and retrieve large sets of related data at once. Multi-tuple operations leverage disclosed data relations to manipulate sets of comparable or arbitrarily related elements. Addi-

tionally we present a detailed comparison of existing solutions and DataDroplets regarding their data model and API, architecture, and trade-offs.

Finally, we have evaluated DataDroplets with a realistic environment and workload based on Twitter [26]. The results show the benefit of DataDroplets enhanced data model and API; the minimal cost of synchronous replication; and attest the scalability of DataDroplets.

The remainder of the paper is organized as follows. Section 2 presents emerging Cloud based tuple stores and DataDroplets. Sections 3, 4, and 5 present, respectively, a detailed comparison of existing solutions and DataDroplets regarding: data model and programming interface; architecture; and design trade-offs. Section 6 presents a evaluation of the performance of DataDroplets. Section 7 concludes the paper.

2 Tuple Stores

The need for elastic and scalable distributed data stores for managing very large volumes of structured data is leading to the emergence of several Cloud based tuple stores.

Major companies like Google, Amazon, Yahoo! and Facebook are competing for a lead in this model. In the following, we briefly present four available tuple stores and DataDroplets. The chosen tuple stores are the most representative and, although several open source projects exist, they are mostly implementations of some of the presented here.

Amazon's Dynamo [8] is a highly available key-value storage system. It has properties of both databases and distributed hash tables (DHTs). Although it isn't directly exposed externally as a web service, it is used as a building block of some of the Amazon Web Services [1], such as S3. Dynamo assembles several distributed systems concepts (data partitioning and replication, Merkle trees, load balancing, etc.) in a production system.

PNUTS [6] is a massively scalable, hosted data management service that allows multiple applications to concurrently store and query data. Its shared service model allows multiple Yahoo! applications to share the same resources and knowledge. PNUTS is a component of the Yahoo!'s Sherpa, an integrated suite of data services. Sherpa is composed of Yahoo!' Message Broker (YMB), a topic based publish-subscribe system and PNUTS.

Google's Bigtable [5] is a distributed storage system for structured data that was designed to manage massive quantities of data and run across thousands of servers. Besides being used internally at Google for web indexing, Google Earth and Google Finance, it is also used to store Google's App Engine Datastore entities. The Datastore API [13] defines an API for data management in the Google's App Engine (GAE) [12]. GAE is a toolkit that allows developers to build scalable applications in which the entire software and hardware stack is hosted at Google's own infrastructure.

Cassandra [17] is a distributed storage engine initially developed by Facebook to be used at the Facebook social network site and is now an Apache open source

project. It is a highly scalable distributed database that uses most of the ideas of the Dynamo [8] architecture to offer a data model based on Bigtable's.

DataDroplets, is a key-value store targeted at supporting very large volumes of data leveraging the individual processing and storage capabilities of a large number of well connected computers. It offers a simple application interface providing the atomic manipulation of key-value tuples and the flexible establishment of arbitrary relations among tuples.

3 Data Model and API

The emergence of Cloud computing and the demand for scalable distributed tuple stores are leading to a revolution on data models. A common approach in all recent large scale tuple stores is the replacement of the relational data model by a more flexible one. The relational data model was designed to store very highly and statically structured data. However, most Cloud applications do not meet these criteria, which results in poorer maintainability than with a more flexible data model. Additionally, the cost of maintaining its normalized data model, by the enforcement of relations integrity, and the ability to run transactions across all data in the database make it difficult to scale.

Therefore, most of existing tuple stores use a simple key value store or at most variants of the entity-attribute-value (EAV) model [19]. In the EAV data model entities have a rough correspondence to relational tables, attributes to columns, tuples to rows and values to cells. Each tuple is of a particular entity and can have its own unique set of associated attributes. This data model allows to dynamically add new attributes that only apply to certain tuples. This flexibility of the EAV data model is helpful in domains where the problem is itself amenable to expansion or change over time. Other benefit of the EAV model, that may help in the conceptual data design, is the multi-value attributes in which each attribute can have more than one value.

Furthermore, cloud based tuple stores rather than using a global data model and operations across the entire universe define disjoint partitions of data that can't be queried together, making them easier to scale. The relational data model has no abstraction for partitions and the application designers must only later considering how it might be reasonably sharded. In the following we describe the data model and API of the tuple stores presented in Section 2 and then motivate and describe the data model and API of the proposed DataDroplets.

3.1 Current tuple stores

In this subsection we provide a detailed description of the data model and programming interface for each of the tuple stores. Dynamo uses a simple key-value data model while others use some variant EAV. Another design choice in the data models is either it leads to row-based or column-based storage. Bigtable and Cassandra are column based while the others are row based. In order to ease the comparison, for each tuple store we provide a standard representation

of their data model and API. The notation has the following symbols: *a*) $A \times B$, product of A and B; *b*) $A + B$, union of A or B; *c*) A^*, sequence of A; *d*) $A \rightharpoonup B$, map of A to B; and *e*) $\mathcal{P}A$, set of A

Dynamo is modeled as:

$$K \rightharpoonup (V \times C) \ .$$

Each tuple has a key associated to it and a context, represented by C, which encodes system metadata such as the tuple's version and is opaque to the application. Dynamo treats both the key and the tuple, K and V, as opaque array of bytes.

$\mathcal{P}(V \times C)$ `get(`K `key)`
`put(`K `key,` C `context,`V `value)`

Fig. 1. Dynamo's API

Dynamo offers a simple interface, Figure 1. The `get` operation locates the tuple associated with the `key` and returns a single tuple or a list of tuples with conflicting versions. The `put` adds or updates a tuple also by `key`.

In PNUTS data is organized into tables, identified by a string, of tuples with dynamic typed attributes and tuples of the same table can have different attributes,

$$String \rightharpoonup (K \rightharpoonup \mathcal{P}(String \times V)) \ .$$

Each tuple can have more than one value for the same attribute. The type for the attributes, V, and for the key, K, can be typical data types, such as integer, string, or the "blob" data type for arbitrary data. The type for the attributes is dynamically defined per attribute.

$\mathcal{P}(String \times V)$ `get-any(`$String$ `tableName,`K `key)`
$\mathcal{P}(String \times V)$ `get-critical(`$String$ `tableName,`K `key,` $Double$ `version)`
$\mathcal{P}(String \times V)$ `get-latest(`$String$ `tableName,`K `key)`
`put(`$String$ `tableName,`K `key,`$\mathcal{P}(String \times V)$ `value)`
`delete(`$String$ `tableName,`K `key)`
`test-and-set-put(`$String$ `tableName,`K `key,`$\mathcal{P}(String \times V)$ `value,` $Double$ `version)`
$K \rightharpoonup \mathcal{P}(String \times V)$ `scan(`$String$ `tableName,`$\mathcal{P}K$ `selections,`$\mathcal{P}String$ `projections)`
$K \rightharpoonup \mathcal{P}(String \times V)$ `rangeScan(`$String$ `tableName,`$(K \times K)$ `rangeSelection,`$\mathcal{P}String$ `projections)`
$String \rightharpoonup (K \rightharpoonup \mathcal{P}(String \times V))$ `multiget(`$\mathcal{P}(String \times K)$ `keys)`

Fig. 2. PNUTS's API

In PNUTS as tables can be ordered or hashed the available operations per table differ, Figure 2. All tables support `get-*`, `put`, `delete`, and `scan` operations. However, only ordered tables support selections by range: `rangeScan` operation. While selections can be by tuple's key, `scan`, or specify a range, `rangeScan`, updates and deletes must specify the tuple's key. PNUTS supports a whole range of single tuple `get` and `put` operations with different levels of consistency guarantees, varying from a call where readers can request any version of the tuple, having highly reduced latency, to a call where writers can verify that the tuple is still at the expected version. Briefly, the `get-any` operation returns a possibly stale version of the tuple, `get-critical` returns a version of the tuple that is at least as fresh as the `version`, `get-latest` returns the most recent copy of the

tuple, `test-and-set-put` performs the tuple modification if and only if the version of the tuple is the same as the requested `version`. Additionally, a `multiget` is provided to retrieve multiple tuples from one or more tables in parallel.

Bigtable is a multi-dimensional sorted map,

$$K \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V)) .$$

The index of the map is the row key, column name, and a timestamp. Column keys are grouped into *column families* and they must be created before data can be stored under any column key in that family. Data is maintained in lexicographic order by row key where each row range is dynamically partitioned. Each cell in BigTable can have multiple versions of the same data indexed by timestamp. The timestamps are integers and can be assigned by Bigtable, or by client applications. The type of the row key, K, and the value for columns V, is a string.

```
put(K key,String ⇀ (String × Long ⇀ V) rowMutation)
String ⇀ (String × Long ⇀ V) get(K key,String ⇀ String columns)
delete(K key,String ⇀ String columns)
K ⇀ (String ⇀ (String × Long ⇀ V)) scan(K startKey,K stopKey,String ⇀ String columns)
```

Fig. 3. Bigtable's API

The Bigtable API, Figure 3, provides operations to write or delete tuples (`put` and `delete`), look up for individual tuples (`get`) or iterate over a subset of tuples, `scan`. For all operations, the string representing the column name may be a regular expression. Clients can iterate over multiple column families and limit the rows, columns, and timestamps. The results for both `get` and `scan` operations are grouped per column family.

Cassandra data model, is an extension of the Bigtable data model,

$$K \rightharpoonup (String \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V))) .$$

It exposes two types of column families: simple and super. Simple column families are the same as column families in Bigtable and super column families are families of simple column families. Cassandra sorts columns either by time or name. In Cassandra the type for rows key, K, is also a string with no size restrictions.

```
put(K key,String ⇀ (String ⇀ (String × Long ⇀ V)) rowMutation)
String ⇀ (String ⇀ (String × Long ⇀ V)) get(K key,String ⇀ (String ⇀ String) columns)
K ⇀ (String ⇀ (String ⇀ (String × Long ⇀ V))) range(K startKey,K endKey,
        String ⇀ (String ⇀ String) columns)
delete(K key,String ⇀ (String ⇀ String) columns)
```

Fig. 4. Cassandra's API

The Cassandra API, Figure 4, is almost the same of Bigtable API except for the `scan` operation. The results of `get` are grouped both per super column family and column family and ordered per column. Additionally, the current version, 0.6, of the Cassandra open source project also supports an additional `range` operation.

None of the presented tuple stores distinguish between inserts and updates. The `put` operation stores the tuple with its unique key, and previous tuple that has that key gets overwritten.

3.2 DataDroplets

In very recent proposals, contrary to RDBMS, there is no standard API to query data in tuple stores. Most of existing Cloud based tuple stores offer a simple tuple store interface, that allows applications to insert, query, and remove individual tuples or at most range queries based on the primary key of the tuple. Regardless of using a simple key value interface or flavors of the EAV model, thus disclosing more details on the structure of the tuple, previous systems require that more ad-hoc and complex multi-tuple queries are done outside of the tuple store using some implementation of the Map Reduce[7] programming model: Yahoo's PigLatin [20], Google's Sawzall [21], Microsoft's LINQ [18].

Although this opens up the possibilities of what can be done with data, it has negative implications in terms of ease of use and in the migration from current RDBMS based applications. Even worse, if the tuple store API hasn't enough operations to efficiently retrieve multiple tuples for the ad-hoc queries they will have a high cost in performance. These ad-doc queries will mostly access a set of correlated tuples. Zhonk et al. have shown that the probability of a pair of tuples being requested together in a query is not uniform but often highly skewed [29]. They also have shown that correlation is mostly stable over time for real applications. Furthermore, it is known that when involving multiple tuples in a request to a distributed tuple store, it is desirable to restrict the number of nodes who actually must participate in the request. It is therefore more beneficial to couple related tuples tightly, and unrelated tuples loosely, so that the most common tuples to be queried by a request would be those that are already tightly coupled.

Therefore, an important aspect of our proposal, DataDroplets, is the multi-tuple access that allows to efficiently store and retrieve large sets of related data at once. Multi-tuple operations leverage disclosed data relations to manipulate sets of comparable or arbitrarily related elements. Therefore, DataDroplets extend the data model of previous tuple stores with tags that allow to establish arbitrary relations among tuples,

$$String \rightharpoonup (K \rightharpoonup (V \times \mathcal{P}String)) \; . \tag{1}$$

In DataDroplets, data is organized into disjoint *collections* of tuples identified by a string. Each tuple is a triple consisting of a unique key drawn from a partially ordered set, a value that is opaque to DataDroplets and a set of free form string tags. It is worth mentioning that the establishment of arbitrary relations among tuples can be done even if they are from different collections. [1]

[1] We are working on extending it to an EAV data model, by allowing each tuple to have dynamic typed attributes but allowing tuples of the same collection to have different attributes. Additionally, each tuple may have more than one value for the same attribute. Briefly, the current opaque value V will be replaced by $\mathcal{P}(String \times V)$.

```
put(String collection,K key, V value, PString tags)
V get(String collection,K key)
V delete(String collection,K key)
multiPut( K ⇀ (V × PString) mapItems)
K ⇀ V multiGet(P(String × K) keys)
K ⇀ V getByRange(K min, K max)
K ⇀ V getByTags(PString tags)
```

Fig. 5. DataDroplets' API

The system supports common single tuple operations such as `put`, `get` and `delete`, multi-tuple put and get operations (`multiPut` and `multiGet`), and set operations to retrieve ranges (`getByRange`) and equally tagged tuples (`getByTags`), Figure 5 .

4 Architecture

Tuple stores target settings of a distributed setting with hundreds or thousands of machines in a multi-tenant scenario and must be able to store and query massive quantities of structured data. At this scale, machines' failures are frequent and therefore tuple stores must replicate data to ensure dependability. Enabling distributed processing over this kind of massive-scale storage poses several new challenges: problems of data placement, dependability and distributed processing. Given these challenges and their different design requirements, all the systems under consideration came up with different architectures.

These architectures may be categorized in three types: fully decentralized, hierarchical and hybrid. In the fully decentralized type physical nodes are kept organized on a logical ring overlay, such as Chord [24]. Each node maintains complete information about the overlay membership, being therefore able to reach every other node. Dynamo and Cassandra fall in this category, Figures 6(c)) and 6(a).

In the hierarchical type, a small set of nodes is responsible for maintaining data partitions and coordinate processing and storage nodes. Both Bigtable and PNUTS (Figures 6(b) and 6(e)) follow this type and organize tuples into tablets, horizontal partitions of tuples. Bigtable is composed of three different types of servers: master, tablets, and lock servers. Master servers coordinate the tablets servers by assigning and mapping tablets to them, and redistributing tasks as needed. The architecture of PNUTS is composed of regions, tablets controllers, routers, and storage units. The system is divided into regions where each region has a complete copy of each table. Within each region, the tablets controllers coordinate the interval mapping that maps tablets to storage units.

DataDroplets uses a hybrid architecture [27] with two collaborating layers, a soft and a persistent state, of distinct structural and functional characteristics. At the top, a soft-state layer is responsible for 1) the client interface, 2) data partitioning, 3) caching, 4) concurrency control, and 5) high level processing. Nodes in the soft-state layer are organized in a logical ring overlay as nodes of the fully decentralized type. Stable storage is provided by the persistent-state layer. Nodes

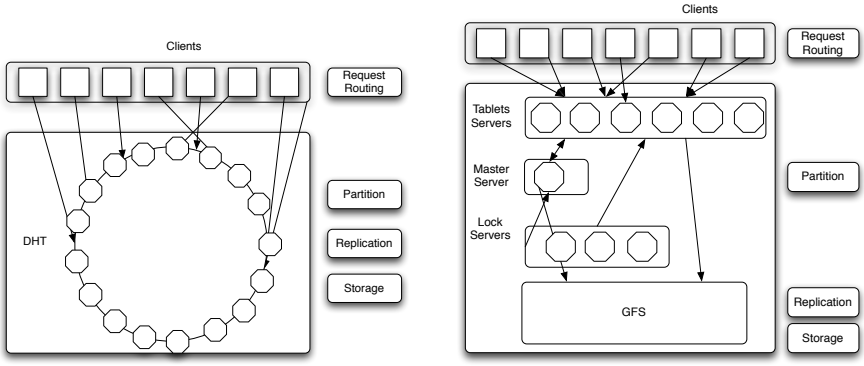
in this layer form an unstructured network overlay, in which nodes are not (a priori) structured but are probabilistically managed. Each layer tackles different aspects of the system, thus making specific assumptions over the computation model and exploiting different techniques to data management and propagation. With these two layers architecture we are able to clearly separate and address the concerns of, on one hand ensuring a strong consistent data storage and, on the other to leverage a massive and highly dynamic infrastructure.

Despite having different architectures, all of the presented tuple stores share common components like request routing and storage and use common dependable and distributed systems techniques such as partition and replication. Figure 6, shows the architecture of each system and highlights which layer is responsible for each component. In the following, we focus on those components highlighting the similarities and differences on how each component is realized in each data store.

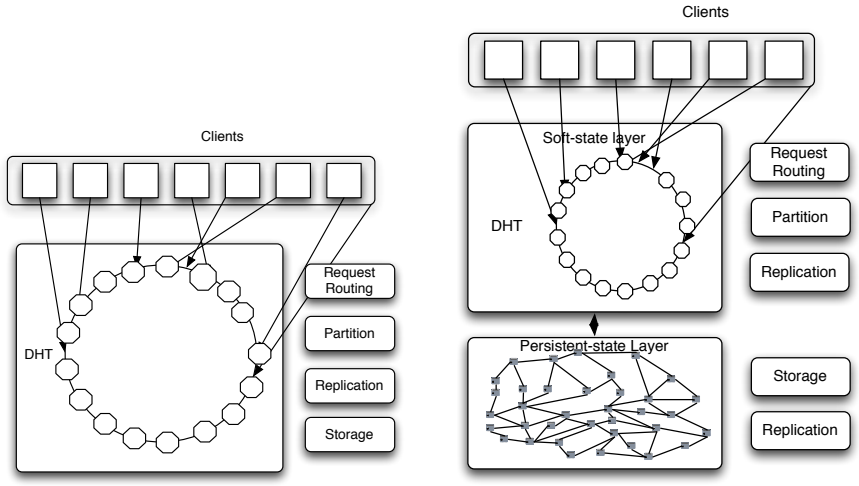
The characterization of the architecture is directly related to the way each system realizes data partitioning, an aspect of major importance. While in tuples stores using a fully decentralized or hybrid architectures the data partition is done in a fully decentralized manner through consistent hashing, in the hierarchical based architectures a small set of nodes is responsible for maintaining the data partitions. In PNUTS data tables are partitioned into tablets - by the tablet controller - by dividing the key space in intervals. For ordered tables, the division is at the key-space level while in hashed tables it is at the hash space level. Each tablet is stored into a single storage unit within a region. In Bigtable the row range is dynamically partitioned into tablets distributed over different machines.

While existing tuple stores do data partitioning taking into account only a single tuple, randomly or in an ordered manner, DataDroplets also supports a data partition strategy that takes into account tuple correlations. Currently, it supports three data partition strategies: random placement, ordered placement, and tagged placement that handle dynamic multi-dimensional relationships of arbitrarily tagged tuples. The partition strategy is defined on a per collection basis.

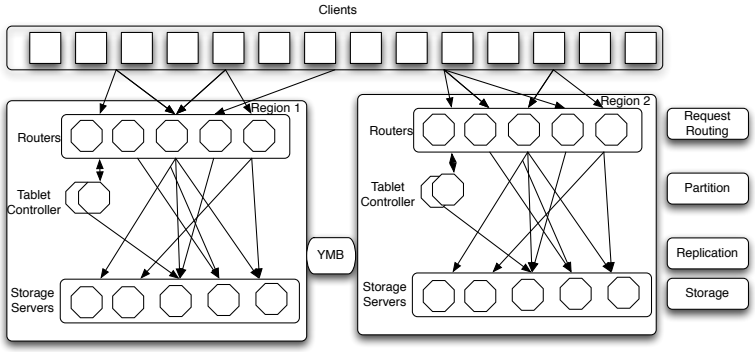
Another mandatory aspect of these tuple stores is replication, which is used not only to improve performance of read operations by means of load balancing but also to ensure dependability. In Cassandra, Dynamo and DataDroplets replication is done by the node responsible for the data, as determined by consistent hashing, by replicating it to the `R-1` successors - with a replication degree of `R`. However, while Cassandra and Dynamo use quorum replication, DataDroplets also allows the use of synchronous primary-backup offering stronger tuple consistency. In PNUTS the message broker assures inter-region replication, using asynchronous primary-backup, while in Bigtable it is done at the storage layer by GFS [11]. In DataDroplets, the replication in the synchronous primary-backup is complemented with replication at the persistent state layer. A tuple is assumed to be safely stored once it is stored in `m` nodes (which become the entry points



(a) Dynamo (b) Bigtable



(c) Cassandra (d) Data Droplets



(e) PNUTS

Fig. 6. Architectures

for the tuple at the soft-layer). For the sake of fault-tolerance more replicas of the tuple are created.

Besides replication, all studied tuple stores use a persistency component to ensure that writes are made durable. While Dynamo and Cassandra rely on local disks for persistency, PNUTS, Bigtable and DataDroplets use a storage service. However, while Bigtable uses it blindly, PNUTS and DataDroplets maintain information about the mapping from tuples to storage nodes. PNUTS maintains the tablets to storage nodes mapping. In DataDroplets tuples stored in the persistent-state layer are massively replicated through gossiping and each node at the soft-state layer maintains a mapping of its tuples into a set of nodes in the underlying persistent-state layer.

Due to the multiple nodes used and the partition of data, every time a request is issued to the tuple store, that request has to be routed to the responsible node for that piece of data. In DataDroplets and Cassandra any incoming request can be issued to any node in the system. Then, the request is properly routed to the responsible node. In PNUTS, when a request is received, the router determines which tablet contains the tuple and which storage node is responsible for that tablet. Routers contain only a cached copy of the interval mapping that is maintained by the tablet controller. In Dynamo, the routing is handled by the client (i.e. the client library is partition aware) directly sending the request to the proper node. Bigtable also needs a client library that caches tablet locations and therefore clients send the requests directly to the proper tablet server.

5 Design and Implementation Trade-offs

Cloud based tuple stores must adapt to multiple tenants with diverse performance, availability and dependability requirements. However, in face of the impossibility stated by the CAP theorem of providing at the same time: network partitions tolerance, strong consistency and availability; cloud based tuple stores must establish reasonable trade-offs.

Looking at the considered tuple stores, all provide two major trade-offs: no table joins and single tuple consistency. The reason all share this first trade-off is that making a relational join over data, which is spread across many nodes, is unbearable, because every node would have to pull data from all nodes for each tuple. Regarding the second trade-off, offering full database consistency through global transactions would restrict scalability, because nodes would have to achieve global agreement.

Although all tuple stores have in common those trade-offs, depending on the internal design requirements each tuple store also establishes specific trade-offs. As a result, in the following we present them and show how they focus on a specific narrow regarding consistency, availability and migration cost. Then, we describe how DataDroplets aims at shifting them towards the needs of common business users. At the end, Table 1 presents a brief comparison of current tuples stores and DataDroplets.

5.1 Trade-offs of current tuple stores

The data model type of a tuple store, in addition to determine its expressiveness, also impacts the way of storing data. In a row based (Dynamo and PNUTS) the tuples are stored contiguously on disk. While in a column oriented storage (Bigtable and Cassandra) columns may not be stored in a contiguously fashion. For that reason, column oriented storage is only advantageous if applications only access a subset of columns per request.

The API of tuple stores is not only highly coupled with their data model, but also with the supported data partition strategies. Single tuple operations are highly related to the data model, but the availability of a range operation is dependent on the existence of an ordered data partition strategy. Therefore, Dynamo doesn't offer a range operation like the other approaches.

Another important trade-off is the optimization either for read or write operations. A key aspect for this is how data is persistently stored. In write optimized storage, Bigtable and Cassandra, records on disk are never overwritten and multiple updates to the same tuple may be stored in different parts of the disk. Therefore, writes are sequential and thus, fast, while a read is slower because it may need multiple I/Os, to retrieve and combine several updates. Dynamo is also optimized for writes because the conflict resolution is done in reads ensuring that writes are never rejected.

As previously stated, all tuple stores offer only single tuple consistency. However, they differ from each other in the consistency given per tuple and how they achieve it through replication. Dynamo exposes data consistency and reconciliation logic issues to the application developers, which leads to a more complex application logic. Moreover, the application must tune the number of tuple replicas `N`, read quorum `R` and write quorum `W`. Therefore, stale data can be read and conflicts may occur, which must be tackled by the application. The conflict resolution can be syntactic or semantic based on the business logic. As multiple versions of the same data can coexist, the update of some tuple in Dynamo explicitly specifies which version of that tuple is being updated. PNUTS also chooses to sacrifice consistency. Its consistency model lays between single tuple atomicity and eventual consistency. Although every reader will always see some consistent version of a tuple, it may be outdated. The proper consistency guarantees depend on the specific calls made to the system. Therefore, the burden of strong consistency is left to the application that must reason about updates and cope with asynchrony. In Bigtable every read or write under a single tuple is atomic. However, every update on a given column of the tuple specifies a timestamp and therefore, creates a new version. Cassandra's consistency is similar to Dynamo with the value's timestamp defining its version.

More on replication, in order to tolerate data center outages, tuple stores must replicate tuples across data centers. Bigtable isn't data center aware. Dynamo is configured such that each tuple is replicated across multiple data centers. PNUTS's architecture was clearly designed as a geographically distributed service where each data center forms a region and a message broker provides

replication across regions. Cassandra supports a replication strategy that is data center aware, using Zookeeper.

5.2 DataDroplets

As previously stated, current tuple stores focus on a specific narrow trade-off regarding consistency, availability and migration cost that fits tightly their internal very large application scenarios. Particularly, all current tuple store's API provide only single tuple operations or at most range operations over tuples of a particular collection. Moreover, while availability is commonly assumed, data consistency and freshness is usually severely hindered.

As explained in Section 3.2, for some applications single tuple and range operations are not enough. These applications have multi-tuple operations that access correlated tuples. Therefore, DataDroplets extends the data model of current tuple stores with tags, allowing to establish arbitrary relations between tuples, which allows to efficiently retrieve them through a tag based data partition strategy.

As previously shown, current tuple stores offer varying levels of tuple consistency but only PNUTS and Bigtable can offer tuple atomicity. However, in both the burden is left to the application that must deal with multiple tuple's versions. In DataDroplets if an application needs atomic guarantees per tuple, it simply configures synchronous replication and it will obtain it transparently without having to maintain and deal with multiple tuple' versions.

Table 1. Comparison of tuple spaces

	Dynamo	PNUTS	Bigtable	Cassandra	DataDroplets
Data Model	key value, row store	EAV, row store	column store	column store	key value + tags, row store
API	single tuple	single tuple and range	single tuple and range	single tuple and range	single tuple, range and correlated tuples
Data Partition	random	random and ordered	ordered	ordered	random and ordered; tuples correlation
Optimized for	writes	reads	writes	writes	reads
Consistency	eventual	atomic or stale reads	atomic	eventual	atomic or stale reads
Multiple Versions	version	version	timestamp	timestamp	none
Replication	quorum	async message broker	file system	quorum	sync or async
Data Center Aware	yes	no	yes	yes	no
Persistency	local and pluggable	storage service and custom/MySQL	replicated and distributed file system	local and custom	storage service and pluggable
Architecture	decentralized	hierarchical	hierarchical	decentralized	hybrid
Client Library	yes	no	yes	no	no

6 Experimental Results

We ran a series of experiments to evaluate the performance of DataDroplets, under a workload representative of applications currently exploiting the scalability of emerging tuple stores.

As there is neither an available version of most of considered tuple stores nor enough available machines to run them, in the following we present performance results for DataDroplets, in particular the enhanced data model and additional consistency guarantees. Moreover we present performance results for the effects of scale by substantially increasing the number of nodes.

6.1 Test workload

For the evaluation of DataDroplets we have defined a workload that mimics the usage of the Twitter social network.

Twitter is an online social network application offering a simple micro-blogging service consisting of small user posts, the *tweets*. A user gets access to other user tweets by explicitly stating a *follow* relationship, building a social graph.

Our workload definition has been shaped by the results of recent studies on Twitter [15,16,3] and biased towards a read intensive workload based on discussions that took place during Twitter's Chirp conference (the Twitter official developers conference). In particular, we consider just the subset of the seven most used operations from the Twitter API [25] (Search and REST API as of March 2010): `statuses_user_timeline`, `statuses_friends_timeline`, `statuses_mentions`, `search_contains_hashtag`, `statuses_update`, `friendships_create` and `friendships_destroy`. Each run of the workload consists of a specified number of operations. The next operation is randomly chosen and, after it had finished, the system waits some pre configured time, think-time, and only afterwards sends the next operation. The probabilities of occurrence of each operation and a more detailed description of the workload can be found in [26].

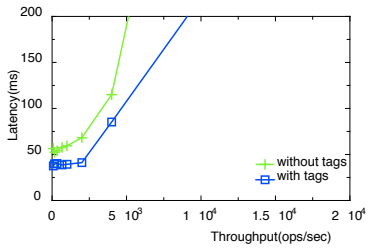
6.2 Experimental Setting

We evaluate our implementation of DataDroplets using the ProtoPeer toolkit [10] to simulate 100 and 200 nodes networks. ProtoPeer is a toolkit for rapid distributed systems prototyping that allows switching between event-driven simulation and live network deployment without changing any of the application code.

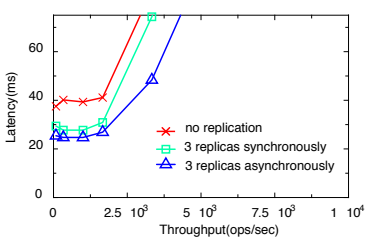
From ProtoPeer we have used the network simulation model and extended it with simulation models for CPU as per [28]. The network model was configured to simulate a LAN with latency uniformly distributed between 1 ms and 2 ms. For the CPU simulation we have used a hybrid simulation approach as described in [23]. All data has been stored in memory, persistent storage was not considered. Briefly, the execution of an event is timed with a profiling timer and the result is used to mark the simulated CPU busy during the corresponding

period, thus preventing other event to be attributed simultaneously to the same CPU. A simulation event is then scheduled with the execution delay to free the CPU. Further pending events are then considered. Each node was configured and calibrated to simulate one dual-core AMD Opteron processor running at 2.53GHz.

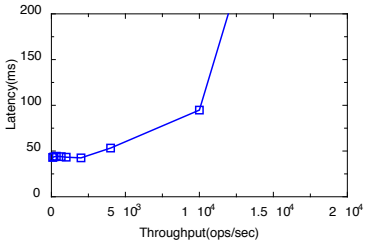
For all experiments presented bellow, the performance metric has been the average request latency as perceived by the clients. A total of 10000 concurrent users were simulated (uniformly distributed by the number of configured nodes) and 500000 operations were executed per run. Different request loads have been achieved by varying the clients think-time between operations. Throughout the experiments no failures were injected.



(a) 100 nodes configuration without replication



(b) 100 nodes with replication



(c) 200 nodes configuration without replication

Fig. 7. System's response time.

6.3 Evaluation of DataDroplets

Figure 7(a) depicts the response time for the combined workload. Overall, the use of tags in DataDroplets to establish arbitrary relations among tuples consistently outperforms the system without tags with responses 40% faster.

When using tags, DataDroplets may use the data partition strategy that takes into account tuple correlations and therefore, stores correlated tuples together. As the workload is composed of several operations that access correlated tuples, the access latency when using tags is lower than without tags, as only other data partition strategies that only take into account a single tuple may be used.

6.4 Evaluation of node replication

Data replication in DataDroplets is meant to provide fault tolerance to node crashes and improve read performance through load balancing. Figure 7(b) shows the results of the combined workload when data is replicated over three nodes.

Despite the impact replication inevitably has on write operations, the overall response time is improved by 27%. Moreover, we can see that despite the additional impact synchronous replication inevitably has on these operations, the overall gain of asynchronous replication is up to 14% which would not, per se, justify the increased complexity of the system. It is actually the dependability facet that matters most, allowing to provide seamless fail over of crashed nodes.

6.5 Evaluation of the system elasticity

To assess the system's response to a significant scale change we carried the previous experiments over the double of the nodes, 200. Figure 7(c) depicts the results.

Here, it should be observed that while the system appears to scale up very well providing almost the double of throughput before getting into saturation, for a small workload, up to 2000 ops/sec with 200 nodes there is a slightly higher latency. This result motivates for a judicious elastic management of the system to maximize performance, let alone economical and environmental reasons.

7 Conclusion

Cloud computing and unprecedented large scale applications, most strikingly social networks such as Twitter, challenge tried and tested data management solutions. Their unfitness to cope with the demands of modern applications have led to the emergence of a novel approach: distributed tuple stores.

In this paper, we presented a detailed comparison of the most representative distributed tuple stores regarding their data model and API, architecture and design and implementation trade-offs. This comparison shows that despite having similar requirements each system offers different data modeling and operations' expressiveness and establish specific trade-offs regarding consistency, availability and migration cost.

Moreover, we introduce DataDroplets, a distributed tuple store, that aims at shifting the trade-offs established by current tuple stores towards the needs of common business users. It provides additional consistency guarantees and

higher level data processing primitives smoothing the migration path for existing applications. Specifically, DataDroplets fits the access patterns required by most current applications, which arbitrarily relate and search data by means of free-form tags.

The results show the benefit, in request latency, of DataDroplets enhanced data model and API; the minimal cost of synchronous replication; and attest the scalability of DataDroplets. Our results are grounded on a simple but realistic benchmark for elastic tuple stores based on Twitter and currently known statistical data about its usage.

References

1. Amazon: Amazon WebServices. http://aws.amazon.com/ (June 2010)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
3. Boyd, D., Golder, S., Lotan, G.: Tweet tweet retweet: Conversational aspects of retweeting on twitter. In: Society, I.C. (ed.) Proceedings of HICSS-43 (January 2010)
4. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. p. 7. ACM, New York, NY, USA (2000)
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 205–218. USENIX Association, Berkeley, CA, USA (2006)
6. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1(2), 1277–1288 (2008)
7. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA (December 2004)
8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 205–220. ACM, New York, NY, USA (2007)
9. Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A., Gauthier, P.: Cluster-based scalable network services. SIGOPS Oper. Syst. Rev. 31(5), 78–91 (1997)
10. Galuba, W., Aberer, K., Despotovic, Z., Kellerer, W.: Protopeer: From simulation to live deployment in one step. In: Peer-to-Peer Computing , 2008. P2P '08. Eighth International Conference on. pp. 191–192 (Sept 2008)
11. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. SIGOPS Oper. Syst. Rev. 37(5), 29–43 (2003)
12. Google: Google App Engine. http://code.google.com/appengine/ (June 2010)
13. Google: Google App Engine Datastore. http://code.google.com/appengine/docs/datastore/ (June 2010)
14. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)

15. Java, A., Song, X., Finin, T., Tseng, B.: Why we twitter: understanding microblogging usage and communities. In: WebKDD/SNA-KDD '07: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis. pp. 56–65. ACM, New York, NY, USA (2007)
16. Krishnamurthy, B., Gill, P., Arlitt, M.: A few chirps about twitter. In: WOSP '08: Proceedings of the first workshop on Online social networks. pp. 19–24. ACM, New York, NY, USA (2008)
17. Lakshman, A., Malik, P.: Cassandra - A Decentralized Structured Storage System. In: SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS) 2009. Big Sky, MT (Ocotber 2009)
18. Meijer, E., Beckman, B., Bierman, G.: Linq: reconciling object, relations and xml in the .net framework. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. pp. 706–706. ACM, New York, NY, USA (2006)
19. Nadkarni, P., Brandt, C.: Data extraction and ad hoc query of an entity-attribute-value database. Journal of the American Medical Informatics Association 5(6), 511–527 (1998)
20. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 1099–1110. ACM, New York, NY, USA (2008)
21. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: Parallel analysis with sawzall. Sci. Program. 13(4), 277–298 (2005)
22. Skillicorn, D.: The case for datacentric grids. Tech. Rep. ISSN-0836-0227-2001-451, Department of Computing and Information Science, Queen's University (November 2001)
23. Sousa, A., Pereira, J., Soares, L., Jr., A.C., Rocha, L., Oliveira, R., Moura, F.: Testing the Dependability and Performance of Group Communication Based Database Replication Protocols. In: International Conference on Dependable Systems and Networks (DSN'05) (june 2005)
24. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable Peer-To-Peer lookup service for internet applications. In: Proceedings of the 2001 ACM SIGCOMM Conference. pp. 149–160 (2001)
25. Twitter: Twitter API documentation. http://apiwiki.twitter.com/Twitter-API-Documentation (March 2010)
26. Vilaca, R., Oliveira, R., Pereira, J.: A correlation-aware data placement strategy for key-value stores. Tech. Rep. DI-CCTC-10-08, CCTC Research Centre, Universidade do Minho (2010), `http://gsd.di.uminho.pt/members/rmvilaca/papers/main.pdf`
27. Vilaça, R., Oliveira, R.: Clouder: a flexible large scale decentralized object store: architecture overview. In: WDDDM '09: Proceedings of the Third Workshop on Dependable Distributed Data Management. pp. 25–28. ACM, New York, NY, USA (2009)
28. Xiongpai, Q., Wei, C., Shan, W.: Simulation of main memory database parallel recovery. In: SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference. pp. 1–8. Society for Computer Simulation International, San Diego, CA, USA (2009)
29. Zhong, M., Shen, K., Seiferas, J.: Correlation-aware object placement for multi-object operations. In: ICDCS '08: The 28th International Conference on Distributed Computing Systems. pp. 512–521. IEEE Computer Society, Washington, DC, USA (2008)