topicos

1. INTRODUCTION

- Num. DC a aumentar
- Replicar totalmente tem problemas
- Replicação parcial
- Queries sobre dados replicados parcialmente
  - Standard solution?
- Views materializadas replicadas totalmente
- Contribuições

2. SYSTEM OVERVIEW

- System model
  - Replicação parcial
- System API
  - "Create table"
  - "Create view"
    - CRDT não uniforme
    - put numa table $\Longrightarrow$ puts nas várias views
      - consistência das views face aos dados - in sync
- System description
  - CRDT não uniforme
  - Implementação de queries?

3. IMPLEMENTATION

4. EVALUATION

5. RELATED WORK

6. CONCLUSIONS

7. SYSTEM OVERVIEW

Possiveis pontos mais detalhados?

7.1 System model

- Network assumptions
- Client-server interaction (refer key-value store interface? Maybe refer this instead in System API?)
- Server-server interaction? (is it needed? We'll already touch this in Replication.)
- System guarantees
  - CRDTs
  - Consistency level
- Replication
- Async
- Op-based
- Maintains consistency, i.e., transaction level based.
- Partial (system admin defined, each server only has a subset of the data based on topics. Potencially some data can be replicated everywhere)

7.2 System API

- Basically how can we translate a problem to sql-like operations
- Create table
- Create view
- Updates (incluir problema de consistência de views/dados)
- Queries (incluir aqui problema de os CRDTs não uniformes precisarem de mais dados? Ou na zona da view?)

7.3 System description

- Structure? Maybe that's for implementation? How much detail?
  - Internal partitioning vs external partitioning? Capaz de não ser boa ideia...
- CRDTs and non-uniform CRDTs?

8. IMPLEMENTATION

- Go
- Transactions (TM/Mat?)
- Replication (RabbitMQ and other stuff?)
- Communication (protobufs. Also worth noticing the compability with existing AntidoteDB clients)
- CRDTs (version management at least)

Customer

id	int
name	string
age	int
country	string

Products

id	int
name	string
value	int

Sales

id	int
custID	string
productID	int
amount	int

Figure 1: Example objects

9. SYSTEM OVERVIEW

9.1 System model

We consider an asynchronous distributed system composed by a set of servers S connected by a network. We assume the network may delay, duplicate or re-order messages, but does not corrupt them. We also assume messages sent are eventually delivered, even if connections may temporarely drop.

Database objects are replicated in one or more servers, but not necessarely in all of them. We consider a server to be a replica of an object if it replicates such object. New updates are sent periodically to replicas asynchrnously. However, not all updates need to be delivered to every replica - only updates that modify the objects' visible state must be delivered eventually **??**.

Clients connect to one or more servers. Updates and queries are executed in a single server and return as soon as its execution ends. Only objects replicated in the server can be accessed, i.e., to access objects not present in a given server the client must connect to a replica of those objects.

9.2 [PLACEHOLDER]Example scenario

For the rest of this paper/section , we'll consider the following example scenario.

Assume we have a very large company with multiple stores across the world. Consider that there's millions of products (each store has a different stock and may sell different items), and tens of millions of clients. Also consider that, for both product warranty and statistical purposes, data related to each product unit sold is kept in the database.

Fully replicating the whole dataset would have proibitive costs. It is also unecessary, as the relevance of the data (and likehood of being accessed from) is dependent on the location. For instance, an asian customer is more likely to consult the stock of stores in his country than in an european country. Thus, it makes sence that both asian customers' data and asian stores' to be replicated mainly in asian datacenters (plus possibility a subset of others for fault tolerance purposes). That is, the servers for which data will be replicated can be choosen based on the geographic location, as its relevance depends on that.

To simplify the example, we'll consider only three types of objects: customers, products and sales. The simplified scheme of each object can be found on Figure ???.

Customer: id: int, name: string, age: int, country: string
Products: id: int, name: string, value: int
Sales: id: int, custID: int, productID: int, amount: int

Customers represent clients that at some point in time have bought at least one product from one of the stores. Products represents items that may be (or have been) for sale. Sales represents the aquisition of one or more units of a product by a client, where custID and productID refer to, respectively, the customer's and product's id field.

For the purpose of this example, we'll consider the following replication scheme for each type of object:

- *Customer:* partially replicated in the data centers present in the continent correspondent to its country;
- *Products:* replicated in all datacenters
- *Sales:* partially replicated in the same datacenters as of the customer who bought the product.

Note that each object being partially replicated does not prevent queries that refer to data replicated in different datacenters from being executed. On section ??? we'll see how can this be done efficiently.

9.3 System API

9.3.1 Key-value store API

PotionDB is a key-value store database. As such, all objects are indexed by a key and support *get* and *update* operations which, respectively, return/alter the state of the object. In PotionDB objects are CRDTs, which ensures that even if objects are modified concurrently in different replicas, their states will eventually converge. As such, both *get* and *update* operations are executed locally, with the effects of *updates* being propagated asynchronously to other replicas.

To facilitate development of applications that want to use PotionDB, both *gets* and *updates* may refer to either the full object state or part of it. A *get* has the following structure for, respectively, full/partial reads:

Get CRDT key bucket crdtType
Get CRDT key bucket crdtType{arguments}

The triple key, bucket, crdtType uniquely identify an object. Bucket is used mainly for partial replication purposes, as described in Section ???. Arguments can be supplied when only a part of the state needs to be returned, with the possible arguments depending on the type of CRDT. E.g., for a map CRDT, we could use:

Get CRDT customer1 customers MAP{name, age}

to return only the values in the customer1 map CRDT referred by the map keys name and age.

As for updates, they have the following form:

Update CRDT key bucket crdtType{arguments}

Key, bucket and crdtType have the same meaning as in gets. The type of arguments depend on the type of CRDT, and each CRDT can support one or more types of update operations. E.g., a map supports both addition and removal of key/value pairs. An insertion/update of an entry in a map can be represented as:

Update CRDT customer1 customers MAP{ADD{name: "David"}}

9.3.2 API for view CRDTs

In relational databases (and even some non-relational ones, e.g., Cassandra) it's common to have *Views.* A view can be defined as being the result of a query on one or multiple objects. Views can then be accessed as if they were normal objects, thus facilitating the definition of new queries.

Views can either be materialized or not (albeit not all databases support both). In a materialized view, the result of the query is stored as an object, which can then be reused later. This can potencially improve drastically the performance of certain queries, but incours an extra cost on object updates, as both the object and the materialized view need to be updated. On the other hand, non-materialized views don't have this extra cost, but they also don't usually improve the performance of queries, as their result is re-calculated on each query.

In PotionDB, it is particularly interesting to support materialized views. To ilustrate that, recall the example tables and scenario defined in Section 9.2, along with the following query:

Determine the 100 customers who have spent the most across all stores. For each customer, the name, age, country and total value spent must be returned.

Without a view, to implement this query it would be necessary to communicate with datacenters across the whole world (as sales and customers are partitioned). Since there are millions of customers and sales, we would not only need to download large amounts of data but also do long data joins, Only after calculating the total value spent for each customer in the service would we be able to reply with the top 100 customers. Due to the sheer amount of data involved, this would have unnaceptable performance.

A possible solution is to use a Top-K CRDT as a materialized view which keeps the list of the 100 customers with highest (global) spendings. This way, the query could be answered by executing only one *get* operation on this CRDT. This solution does pose some difficulties, namelly: (i) association of each entry in the top-k with the respective customer and sales' CRDTs; (ii) translation of updates in customers or sales to updates for the top-k; (iii) keeping top-k updated even if some entries refer to CRDTs not replicated in the replica in which the top-k is created; (iv) automatically include customer and sales CRDTs that may be introduced after the creation of the top-k.

To support materialized views, we thus introduce another construct in our API, named *link*. The intuitive idea is that, after creating the CRDT that will be acting as a view, we can issue a *link* operation in order to provide the "rules" for the automatic updating of the referred CRDT. PotionDB will then use these "rules" to keep the CRDT updated.

A link operation has the following structure:

Link CRDT key bucket crdtType ...

9.4 System description