Global Views on Partially Geo-Replicated Data

André Rijo
NOVA LINCS, FCT,
Universidade NOVA de Lisboa

Carla Ferreira
NOVA LINCS, FCT,
Universidade NOVA de Lisboa

Nuno Preguiça
NOVA LINCS, FCT,
Universidade NOVA de Lisboa

ABSTRACT

andre: Please review this

Many existing web services of global scale have strict latency, availability and fault tolerance requirements. To handle this, some services distribute their datacenters across the globe, with users accessing the closest datacenter. As the number of servers increases, so does the replication cost, thus partial replication can be desirable. However, some queries may require global data not replicated in every server (e.g. in an online store scenario, the top 10 most sold products globally).

In this paper, we present PotionDB, a novel geo-distributed key-value store which provides partial replication. We present a replication scheme which allows data to be partially replicated, yet still answer queries relative to global data efficiently. We leverage on existing partial and non-uniform replication algorithms to provide materialized views of data that may be split across multiple servers. With these views, a client can obtain global information by querying a single server. Our evaluation shows that queries concerning global data can be completed in less than one millisecond, while maintaining high throughput despite the extra updates required to keep the views up-to-date.

PVLDB Reference Format:
. . *PVLDB*, 12(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/TBD

1. INTRODUCTION

The increasing reliance on web service in many domains of activity, from e-commerce to business applications and entertainment, leads to stringent requirements regarding latency, availability and fault tolerance [15, 9]. To address these requirements, cloud platforms have been adding new data centers at different geographic locations. By allowing users to access a service by contacting the closest data center, a global service can provide low latency to users spread across the globe. The increasing number of data centers also contributes for proving high availability and fault tolerance, by allowing a user to access the service by accessing any available data center.

The database is a key component of any web service, storing the service's data. For supporting global services running at multiple geographic locations, it is necessary to rely on a geo-replicated database [7], which maintains replicas of the data at the data centers where the service is running. A number of geo-replicated databases have been proposed, providing different consistency semantics. Databases that provide strong consistency [4, 17, 11] intend to give the illusion that a single replica exists, requiring coordination among multiple replicas for executing (update) operations. This leads to high latency and may compromise availability in the presence of network partitions. Databases that provide weak consistency [18, 7, 12] allow any replica to process a client request, leading to lower latency and high availability. As a consequence, these databases expose temporary state divergence to clients, making it more difficult to program a system.

In either case, geo-replicated databases typically rely on a full replication model, where each data center replicates the full database, with data being sharded across multiple partitions in each data center. As both the data managed by these systems increases in size and the number of data centers increases, this approach leads to a number of problems. First, storing all data in all data centers imposes a large overhead in terms of storage. Furthermore, storing some data in all data centers may be unnecessary, as data is only needed at some geographic locations. Second, increasing the number of data centers makes the replication process more complex and costly, as each update needs to be propagated to all other data centers.

For addressing these problems, partial replication is an attractive approach, with each data center replicating only a subset of the data. A number of works have been addressing the challenges of partial replication, for example by proposing algorithms to manage partially replicated data [**?**, **?**, **?**] and to decide which data is replicated in which replica [].

In this paper we address the problem of querying data in a weakly consistent partially geo-replicated database, focusing on recurrent queries for which a programmer would want to generate a (materialized) view. For example, consider an e-commerce system with users from multiple geographic locations. In this case, the data pertaining users of a given location does not need to be replicated in all data centers (but only in a few for fault tolerance). The same applies to other information, such as data on orders and warehouses. Other data, such as information on products would be replicated in the regions where the product is available. Under this data placement, obtaining the list of best seller products is challenging, as it requires accessing data that is located at multiple data centers.

Several possible solutions exist for this problem. First, it is possible to have a data center that replicates all data, and forward these queries to such data center. Doing this imposes a latency penalty

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/TBD

and requires a data center to host all data and execute all queries of this type. Second, it is possible to execute the query by accessing multiple locations, by using, for example, a distributed processing system with support for geo-partitioned data [10, **?**]. This approach requires running an additional external service and poses challenges for the consistency of the results returned and the data observed by users.

We propose a different approach: to maintain materialized views, as commonly available in relational databases. Implementing such feature efficiently in a partially geo-replicated database requires addressing two main challenges. First, it is necessary to guarantee consistency between the base data available in a replica and the relevant materialized views. To achieve this, we designed a replication mechanism where updates to the base data and views are made visible atomically in each replica.

Second, it is necessary to efficiently support views with limits, used for example to support *top-k* queries. To achieve this, we build on the concept of non-uniform replication [3], in which the state of different replicas may be different, given that the observable state is (eventually) the same. This allows each replica to propagate only the updates that might be relevant to the observable state. Providing support for views required us to extend non-uniform replication from simple data types to more complex structures that could support a view with multiple columns. **nuno:** can we support updates to the views? why not? **andre:** short-answer: no. Since views don't have all data (e.g: only the name of a customer), we can't translate an update in a view to updates in other CRDTs, as such update would be incomplete

We present the design and implementation of PotionDB, a geo-replicated key-value store with support for partial replication and materialized views. PotionDB provides weak consistency, for improved latency and availability, and support for highly available transactions [**?**]. To our knowledge, our work is the first to address the problem of maintaining materialized views in such setting.

We have evaluated our system using micro-benchmarks and TPC-H queries [5]. The results show that our algorithms for maintaining materialized views impose low overhead when executing and asynchronously replicating transactions, particularly for views with limits. **nuno:** deviamos ter uns micro-benchmarks que comparassem o overhead com limites e sem limites Additionally, the results show that executing queries by relying on the materialized views is much more efficient than using alternative mechanisms. Furthermore, our algorithms for maintaining materialized views in a decentralized way perform better that alternative approaches where the view is computed in a single data center, while being able to keep consistency between the base and view data in every replica.

In this paper we make the following contributions:

- the design of a geo-replicated key-value store with support for partial replication and views over partially replicated data;
- replication algorithms for efficiently maintaining consistent materialized views over partially replicated data;
- an implementation and evaluation of the proposed approach with micro-benchmarks and TPC-H.

The remainder of the paper is organized as follows. Section ...

2. SYSTEM OVERVIEW

In this section we provide an overview of the functionality of PotionDB, focusing on its support for repetitive queries based on views.

PotionDB is designed for supporting global services deployed at multiple data centers. In these settings, it is common that (at least some) data items are only needed at some geographic locations. As a running example, we will consider a large e-commerce company with online stores for different countries and clients spread across the world. The information system includes data of millions of products, with some products available only at some locations. The system also includes information about customers and their purchases, with clients being associated a given online store. We note that similar patterns occurs in multiple other systems: online games, where players are partitioned depending on their locations; **we need more examples**.

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

Figure 1: Example objects

In this context, fully replicating the whole dataset might become too expensive. Additionally, as data is tied to some geographic location, one can expect that the large majority of accesses occurs on that location. **need better examples**. For instance, an asian customer is more likely to consult the stock of stores in his country rather than in an european country. Thus, it makes sense that both asian customers' and asian stores' data 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 such factor. To support this

As a consequence,

3. MOTIVATION EXAMPLE

In order to both facilitate the understanding of the rest of the paper, as well as show the usefulness of queries supported by materialized views, we present the following example scenario.

Consider a large scale commerce company with stores and clients spread across the world. Assume the company keeps data of millions of products, with each store having its own stock of products available. Also consider that there are millions of both current and past customers and, among other data, all sales ever done are kept in the database, for both product warranty and statistical purposes.

Fully replicating the whole dataset would have proibitive costs. It is also unecessary, as both the relevance of the data and likehood of being accessed from are dependent on the location. For instance, an asian customer is more likely to consult the stock of stores in his country rather than in an european country. Thus, it makes sence that both asian customers' and asian stores' data 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 such factor.

To keep the example simple, we will only consider three types of objects: customers, products and sales. The simplified scheme of each object can be found on Figure 1.

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, respectivelly, 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:* replicated in the data centers present in the continent correspondent to his country (plus a few other data centers for fault-tolerance purposes);
- *Products:* replicated in all datacenters;
- *Sales:* replicated in the same data centers as of the customer who bought the product.

Despite the fact that some objects aren't replicated everywhere, PotionDB will still support queries that refer to a global view of the database. E.g., queries such as "top 100 customers who have spent the most across all stores" must be efficiently handled by PotionDB, even though no server contains all customer and sales data. This kind of queries allows to gather important real-time statistics which allow businesspersons to make decisions and changes on the business strategy to improve its rentability. **andre:** Should I refer tpc-h here?

The following is an example of a query requiring a global view of the database:

> *Get the top 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 views and by using just the three kinds of objects in Figure 1, executing this query would require communication with multiple datacenters across the world, as both sales and customers' data is partitioned. Since there are millions of customers and sales records, this would imply downloading large amounts of data and also spending a considerable amount of CPU time for data joining. After the joins are complete, it is still needed to calculate the total value spent for each customer and sort descendingly based on the total value spent by each customer. Only after this whole process is complete can the query be answered correctly. Due to the sheer amount of data involved, this query would have unnaceptable performance if executed in this way.

Defining the right view(s) avoids having to do for each query the process referred above. For the example query above, a view that keeps, for each customer, his name, age, country and total value, ordered by total value, can answer efficiently the query with a single get operation. With this we no longer need to contact multiple datacenters and execute millions of gets and result processing.

On Section 8 we address multiple difficulties related to supporting and keeping views updated, as well as our solutions for such difficulties in PotionDB.

4. SYSTEM DESIGN

PotionDB is a geo-distributed storage system which provides weak consistency or, more precisely, causal consistency. A key feature in PotionDB is the ability to efficiently provide global views on partially replicated data under the conditions mentioned previously. In this section we discuss design decisions which made it possible to provide such feature.

4.1 Database Model and API

Explicar qual o modelo da BD: key-value store

Objetos são CRDT.

Objetos organizados em buckets. Falar das partições.

Clientes executam transações com modelo de consistência: Parallel Snapshot Isolation.

Unlike typical key-value stores, suporta a definição de views sobre os dados.

Modelo de consistência estende-se para as views (SEMPRE???).

4.2 Data Definition API

(será aqui??) apresentar como se define a que partição pertence cada objeto.

4.3 Data Access API

Apresentar as operações

```
begin_tx()
commit_tx()
rollback_tx()

get( key) -> object
readOp( key, op, params) -> data
updateOp( key, op, params)
```

Explicar o que são - isto ém qual o resultado esperado.

4.4 View Definition API

Apresentar as operações - que tipo de views são suportadas.

Explicar o que são e o resultado esperado.

Dizer que do ponto vista do acesso uma view é como se fosse um objeto normal, sendo acedid através da API normal.

4.5 Database Architecture

apresentar a arquitetura a nível geral.

Explicar servidores, clientes, sincronização.

5. TRANSACTIONS OVER PARTIALLY REPLICATED DATA

6. SUPPORTING VIEWS

6.1 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. We don't require for all updates to be delivered to all replicas, as will be detailed in Sections **??** and 7.2.2

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

6.2 System API

6.2.1 Key-value store API

PotionDB provides a key-value store interface [**?**]. As such, all objects are indexed by a key and support *get* and *update* operations which, respectively, query/modify the state of an object. In PotionDB objects are CRDTs [**?**], which ensures that even if objects are modified concurrently in different replicas, their states will eventually converge. This allows for both *get* and *update* operations to be executed locally, with the effects of *updates* being propagated asynchronously to other replicas.

To ease development of applications that use PotionDB, both *gets* and *updates* may refer to either the full object state or part of it. A *get* has the following interface:

Get CRDT key bucket crdtType{*arguments*}

The triple key, bucket, crdtType are mandantory arguments used to uniquely identify an object. Bucket is used mainly for partial replication purposes, as will be described in Section **??**. Arguments are optional as they're used only if reading part of the state, 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 name and age of a customer stored as a map CRDT.

Updates have a similar interface:

Update CRDT key bucket crdtType{*arguments*}

Key, bucket and crdtType have the same meaning as in *get*. The type of arguments depend on the type of CRDT, and each CRDT may support multiple 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"*}}

6.2.2 API for view CRDTs

In relational databases (and even some non-relational ones, e.g., Cassandra [**?**]) it is common to have *views*. A view can be defined as being the result of a query on one or multiple objects [**?**]. Views can be acessed as if they were normal objects, thus it eases 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 may potencially improve drastically the performance of certain queries, but incours an extra cost on 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.

PotionDB supports materialized views. Similarly to relational databases, defining the right materialized views allows for efficient execution of complex queries. In the case of PotionDB, views may even refer to data that isn't replicated locally.

We extend our key-value API with SQL-like sintax in order to support views. This can be seen as a layer on top of the key-value API which offers two operations: *create table* and *create view*. This layer then converts those operations to one or more *gets* and *updates*.

The *create table* operation has the following structure:

Create Table name As
From (key1, bucket1*), (key2*, bucket2), ..., (keyN*, bucketN) crdtType*

andre: Should the create table operation also have where? E.g., group only customers of a certain nation.

The goal of *create table* is to group together multiple CRDTs under the same name, which can then be referred to by *create view*. The *from* clause specifies the objects to be grouped. Note the use of the "*" symbol - it means that every value which starts with the string before "*" will match. E.g., for "customer*", both "customer" and "customer1" match. This serves two goals - first, to more easily include existing objects and, second, to also allow to include future objects that may be latter added which should also belong to the group.

One could argue that *bucket* could be used for the purpose of grouping objects for views , but doing such would mix the management of partial replication with views. If that were done, a system admin would have to consider both how data should be replicated and which views might be created when defining buckets. We believe that splitting view and partial replication management is easier to be deal with.

For creating the view itself, the *create view* operation is used, which has the following sintax:

Create View key bucket As
Select table1.a, table1.b, ..., tableN.a, tableN.b, ...
From table1, ..., tableN
Where conditions
Group by table1.a, ...
Order by table1.a, ...

All the clauses in create view have the same meanings as in SQL. *From* specifies which tables to take information from, while *select* chooses which fields should populate the view. Note that both fields and read operations can be issued in select, which allows to use as base others objects besides maps. *Where* restricts which objects should populate the view, while *group* and *orderBy* are optional and serve to organize the objects in the view.

Depending on *select* and the presence of *group by* and *order by*, different types of CRDTs may be generated by the *create view* operation. Section 8 covers in detail which kinds of views are supported in PotionDB, along with how views are kept correct and in sync with their base data.

andre: Should I still give here an example? Before I had here an example of a view operation that generated a TopK which answered the query in Section 3 with just a simple get. But now it would seem odd to include such example here imo.

6.2.3 [OLD]API for view CRDTs

In relational databases (and even some non-relational ones, e.g., Cassandra [**?**]) it is common to have *views*. A view can be defined as being the result of a query on one or multiple objects [**?**]. Views can be acessed as if they were normal objects, thus it eases 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 may potencially improve drastically the performance of certain queries, but incours an extra cost on 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.

PotionDB supports materialized views. Similarly to relational databases, defining the right materialized views allows for efficient execution of complex queries. In the case of PotionDB, views may even refer to data that isn't replicated locally.

We extend our key-value API with the *link* operation. The idea of *link* is to specify how can an object (view) be updated based on updates executed on other objects (base data). Whenever an update is issued on the base data, PotionDB will also simultaneously update the view.

The link construct has the following structure:

Link CRDT key bucket crdtType
From (key1, bucket1*, crdtType1), ..., (keyN*, bucketN*, crdtTypeN)*
Where conditions
Update updateArgs

Similarly to *get* and *update*, each triple key, bucket, crdtType uniquely identifies an object. The link operation can be split in four parts:

- *Link*: specifies which CRDT will be acting as a view, i.e., the target of *update*;
- *From*: Similarly to the *from* construct in SQL, it specifies which CRDT(s) will be used to build the view; In other words, whenever an update happens in one of these CRDTs, an update may also be triggered in the link CRDT. Note that in this construct exclusively, both key and bucket may be a prefix of the keys/buckets of the CRDTs (this is represented by the '*' character). This allows to include objects that may not yet exist in the database but that should also be included when they get added (e.g., new customers);
- *Where*: Similary to the *where* construct in SQL, it specifies using conditions how can the CRDTs returned in *from* be joined/filtered.
- *Update*: Specifies the operation that, for each entry returned by *from* after applying *where*, will be used to update the CRDT specified in *link*.

The intuition is that when *link* is first executed for a view CRDT, the view is updated based on all existing objects that match the *from* and *where* clauses. Afterwards, whenever a CRDT that matches *from* is updated, the *where* clause is executed for that CRDT only and the *update* part is executed on the view. In Section 8 we discuss in detail different possibilities for implementing this, as well as how it is done in PotionDB.

The query presented in Section 3 can be answered with a single *get* on the Top-K CRDT generated by the following *link*:

Link CRDT topsales statistics TOPK
From (customers, customers, MAP), (products*, products, MAP), (sales*, sales, MAP)*
Where sales.MAP{*customerID*} *= customers.MAP*{*id*} and *sales.MAP*{*productID*} *= products.MAP*{*id*}
Update MAP{*name: customers.MAP*{*name*}*, age: customers.MAP*{*age*}*, country: customers.MAP*{*country*}*, spent: products.MAP*{*value*} ** sales.MAP*{*value*}}

7. PARTITIONING, REPLICATION AND CONSISTENCY

In this section we describe important mechanisms in PotionDB: partitioning, replication and transactions. The referred mechanisms are the basis for efficiently supporting queries on global data, while still maintaing reasonable update performance and space overhead.

7.1 Partitions

In PotionDB objects are partitioned both internally in a server and externally between different servers. Both partitioning mechanisms work differently and have different goals, which we explain in this subsection.

7.1.1 Internal partitions

andre: This likely needs to be better motivated/explained...

Inside a server objects are split across multiple partitions. Each object is assigned to one partition automatically. The partition to which an object is assigned to is determined by computing an hash based on the object's key, bucket and type.

The idea of partitioning objects internally in a server comes from the observation that, by having data grouped in multiple "slot" (i.e., partitions), if different clients access objects in different partitions, then both requests can be processed concurrently by a replica without any conflict. And if both requests are read only, then they can be processed concurrently even if some objects are present in both requests, as reads don't conflict [**?**]. Each partition has one thread associated, which allows PotionDB to efficiently use multi-core CPUs.

At a first glance it may seem odd to partition based on a hash with the goal of improving performance. However, this is justified by the expected use of PotionDB - simirlarly to relational databases, the most frequent and complicated queries are expected to have, for each one, a single view that answers them directly, thus requiring only a single operation. Due to the nature of hashing, these views are very likelly be spread across multiple partitions. This implies that multiple queries can be executed concurrently without conflicts even in the presence of concurrent updates that only affect unrelated objects. Our practical evaluation shows that this partitioning improves PotionDB's performance quite considerably.

Other possible solutions to make use of multi-core CPUs are possible, for instance:

1. using locks to protect the same object from being concurrently modified [**?**];
2. running multiple PotionDB servers in the same computer.

Alternative 1 is difficult to implement, as it is needed to find the right level of lock granularity and when should locks be obtained/released [**?**]. Special care is needed to avoid deadlocking concurrent transactions. There's also concerns with both fairness and overhead of obtaining locks. Our solution does not need to lock data, thus it avoids the overhead from using locks, is easier to implement and does not have deadlocks.

As for 2, this would imply more replicas running in the system, which increases both replication and storage costs. This would also imply concurrency conflicts even in the same computer. Finally, while this can allow more clients to be processed in the same period of time, and is quite easy to implement, it does not improve the performance of each client, as each transaction must be done in a single replica (and thread) to avoid breaking consistency.

7.1.2 External partitions

Since PotionDB is a partially replicated [**?**] database, it is necessary to have a mechanism which allows to precisely define in which servers should an object be replicated in.

We achieve this by requiring each object to be identified not only by its key, but also by a "bucket". Buckets define groups of objects that are replicated in the same group of replicas, that is, two objects with the same bucket value will be replicated in the same replicas. For each replica, the system admin can define which buckets will be replicated in it.

To exemplify, recall the commerce example presented in Section 3. The customers can be partitioned based on their country's continent by taking the following steps:

1. Define one bucket per continent;

2. Configure the servers so that each replica replicates the bucket of its continent + another one for fault tolerance purposes;

3. When adding a customer to the database, specify the bucket correspondent to the customer's continent.

It's worth noting that clients must ensure they are communicating with a server which replicates the buckets they want to operate on, since each server only replicates a subset of the buckets and doesn't forward operations to other servers.

7.2 Replication

Replication in PotionDB is asynchronous and partial. Operations are executed localy, without needing to contact other replicas. Periodically, new updates are propagated asynchronously to other replicas. All objects in PotionDB are operation-based CRDTs, thus the information required to propagate an update consists in the operation type and its arguments.

7.2.1 RabbitMQ

PotionDB uses RabbitMQ [**?**] for handling communication between replicas. RabbitMQ is a message brooker which allows consumers to register the topics of messages in which they're interested. We leverage on topics to ensure each replica only fetches the messages containing updates for objects in buckets they are replicating.

We use RabbitMQ as follows. Each PotionDB servers runs alongside it a RabbitMQ instance. When a replica starts, it contacts other server's RabbitMQs and subscribes to all messages whose topics match the buckets it replicates. When a replica wants to publish new updates, it splits those updates by buckets (while keeping causality), ensuring each message only contains updates for one bucket and whose topic value is that bucket. Those updates are then sent to the local RabbitMQ instance, which then forwards them to interested PotionDB instances.

We also use RabbitMQ to support the addition of new replicas without stopping the system. However, such mechanism is outside of the scope of this paper and thus we don't describe it here.

7.2.2 NuCRDTs replication

Non-uniform CRDTs [**?**] leverage on the fact that, for certain objects, not all of the object's data is necessary to answer queries. E.g., in a Top-K CRDT that only maintains the K elements with highest value, only those K elements must be replicated in every replica.

The key difference between eventual consistency and non-uniform eventual consistency is that, instead of requiring for the states of each object to be eventually equivalent, it requires for the *observable* stables to be eventually equivalent [**?**]. Two states are defined as observable equivalent iff, for each possible query, the result is equivalent when executed on either states. This allows to save both storage space and communication overhead, as not all updates in a non-uniform CRDT must be replicated to all replicas [**?**].

Non-uniform CRDTs are specially useful for using as materialized views, as they allow to keep summaries of data easily while also being space-efficient. E.g., in the scenario described in Section **??**, even if there's millions of customers globally, replicas don't need to keep data for all customers in order to correctly apply reads in the Top-K CRDT.

andre: What's above, technically, isn't 100% true: all replicas together need to keep ALL entries of the top-k due to removes being supported. Thus, an entry for all customers in the system. This WILL BE reflected in the experimental evaluation's graphics. However, no single replica needs to keep all entries (albeit, as of now, one does).

PotionDB fully supports NuCRDTs. Thus, when an operation is applied on a NuCRDT, the operation is only replicated to all of that CRDT's replicas iff that operation changed the visible state. Some operations however may only have an effect later (e.g: in a top-k, an element may rise to the top after another one gets removed) [**?**]. Operations in that category are only replicated to all replicas when it affects the visible state.

andre: Should I refer that those operations in the last category are/should (as they aren't as of now) be sent to a few replicas for fault-tolerance purposes?

7.3 Transactions

andre: This section quite likely still needs to be heavily worked on, but I don't know how to proceed with it...

PotionDB offers transactions with casual consistency. That is, clients see operations' effects by an order consistent with causality [**?**]. Since casual consistency is a form of weak consistency, concurrency conflicts will still occour when operations are applied on different replicas. We leverage on CRDTs in order to deal with those.

Transactions allows to group multiple operations together and have them executed with certain guarantees. They facilitate the usage of database systems [**?**]. For example, when registering a sale of a product, it is useful to have the product stock and customer info to be updated simultaneously.

Relational databases typically provide ACID (atomicity, strong consistency, isolation, durability) guarantees for their transactions [**?**]. These properties ensure that, respectivelly: (i) either all operations execute successfully or none does; (ii) the database is left in a consistent state; (iii) there is no interferance from other transactions; (iv) the effects of the operations won't be lost even if replicas fail. Having all of these properties facilitates the development of applications, as it reduces the possible anomalities that may be observed. E.g., in the scenario of a cash transfer between two entities, if we forego atomacity, it's possible the cash is withdraw from one account without being deposited in the other due to, e.g., a server crash.

Unfortunately providing ACID in a database requires providing strong consistency [**?**] , which implies reduced fault tolerance and may limit performance. In a geo-distributed scenario it also implies a quite higher latency overhead due to the required synchronization between DCs [**?**]. Transactions are still useful even without all ACID guarantees, as is evidenced by multiple weakly consistent databases providing them with different guarantees [**?**].

PotionDB supports non-ACID transactions. Transactions in PotionDB can span any number of partitions (both internal and external) existent in a replica, but it cannot refer to buckets that aren't replicated locally. That is, transactions are local to a replica. We support both read-only, write-only and mixed transactions, with all having the same guarantees.

PotionDB's transactions are atomic and provide causal consistency. Each transaction runs isolated from other transactions in the same server, but the same objects may be concurrently modified by different transactions in different servers. Durability isn't ensured, as replication is asynchronous and operations don't need to be written to disk for a transaction to commit. In order to understand in detail the guarantees of PotionDB's transactions, some insight on how they're handled needs to be given.

andre: We don't exactly guarantee atomicity - if the replica executing the transaction fails while applying updates, the DB will be left in an inconsistent state. Which isn't exactly relevant, since data would be lost in that case anyway... We do guarantee that, if the server doesn't fail, the client either sees the effect of the whole transaction or of none of its operations.

andre: Also... how much detail of the transaction execution mechanism should I provide here? Or just talking about the guarantees is enough?

Each replica maintains a vector clock which summarizes the current DB state. Each entry in the vector clock corresponds to the latest commit of each replica that is known locally. When a transaction starts, a copy of this vector clock is associated to the transaction. The entry correspondent to the local replica is processed differently - for that entry, a unique, monotonically increasing timestamp, is assigned to each transaction, which ensures each transaction will commit with a different vector clock.

andre: Where (and should I?) do I explain how the version management works? Or should we leave that for a short paper in another conference? Also I'm hiding the 2 phase-commit part, and that commitTS != initialTS.

Both updates and reads take the transactions' vector clock into account. More precisely, reads execute on the state of the object correspondent to the transaction's vector clock (i.e., ignoring all update operations that may have happened afterwards, but considering updates in the same transaction that happened-before). The effects of updates are only applied to the latest-version of an object when the transaction is commiting. Transactions commit according to the order defined by the vector clocks. A transaction is considered commited after all operations have been applied. At this moment, if it isn't a read-only transaction, the replica's entry on the local vector clock is updated and the client is notified.

We can now leverage on the workings of PotionDB's transactions to detail the exact guarantees they provide.

7.3.0.1 *Atomicity.*

A transaction is atomic because the replica's vector clock is only updated after all operations are successfully applied. If at least one fails, all updates are rendered innefective as we return the CRDTs to their previous state. Previous reads on that transaction are irrelevant due to the transaction being considered as aborted. Reads on other transactions don't see the effects of a transaction until it is commited, thus aborts don't pose a problem to them. Other replicas only start applying a transaction after all updates for it are received.

7.3.0.2 *Consistency.*

As every transaction has a vector clock associated, we can thus totally order all transactions. A transaction is only executed if all transactions with a smaller vector clock have already been executed. We ensure locally generated transactions are executed by the order specified by the monotonicaly increasing local timestamp. For concurrent transactions any order is possible and consistent with casual consistency. CRDTs ensure concurrent operations don't pose a problem as they're commutative [**?**]. Whenever a CRDT with views associated is updated, the views are also updated in the same transaction with the same clock. Thus, PotionDB respects causality when executing transactions and, as such, provides causal consistency.

7.3.0.3 *Isolation.*

There is no concurrency inside each partition. Reads are executed on top of the version specified by the transaction's vector clock. Updates are only applied when commiting and are executed sequentially inside the partition. Updates from other transactions are either executed before the first operation of the current transaction, or after the last. Thus, the effects of each local transaction can be ordered sequentially, which implies isolation. We do not guarantee isolation between transactions executing in different replicas however.

It is important to note that even though we provide atomicity, consistency and isolation guarantees, the latter two are equivalent to the ones provided in ACID. We provide causal consistency (i.e., weak consistency) instead of strong consistency; we don't guarantee isolation between servers (which is required in ACID). As evidenced in both academy and industry, while ACID properties are quite useful, they are also severlerly limiting in terms of fault tolerance and performance [**?**].

8. VIEWS

Views are useful as they make it easier to specify queries and, in the case of materialized views, it may improve their performance as well. An essencial question in supporting materialized views in a database is on keeping them coeherent with the base data they refer to. When an object is updated the views must also be updated, otherwise the database becomes inconsistent which may then lead to errors in applications [**?**].

Usually in relational databases, views are updated automatically by using triggers associated to the base data tables [**?**]. Due to data being fully replicated and strongly consistent, it is guaranteed that at all times the view reflects the latest correct state [**?**].

PotionDB is a weakly consistent, geo-distributed and partially replicated database. This poses considerable challenges for maintaining materialized views, namelly:

1. Association between views and their base data. PotionDB is a key-value store, thus multiple objects that would usually belong to the same table in a relational database (e.g., customers) have different keys associated.

2. Automatic inclusion of objects created after the view that represent entities of the same kind (e.g., new customers) but necessarely have different keys.

3. Keeping the views and base data synchronized at all time. I.e., from the point of view of a client, updates on base data should lead to the view being updated simultaneously.

4. Keeping views updated even if all necessary data for it isn't present locally due to partial replication.

5. Translation of updates in base data to views. Unlike in relational databases in which all data can be seen as tables with multiple columns, here there are multiple types of objects which have different kinds of properties and operations.

In this section we detail the inner workings of views. First, we show which kind of objects are supported as views and how they're built. Secondly, we explain how are the referred problems dealt with in PotionDB. We start by assuming all required data is present localy and thus focus on the first three challenges. Afterwards, we drop such assumption and discuss how view updates for non-local data are handled. Finally, we explain how are updates for each kind of view generated based on updates for the base data.

8.1 View CRDTs

andre: Suggestions for better names for these "kinds" of views?

andre: TODO: For each one, include how a SQL update in a table would be converted to updates to both the base and view data.

We support four kinds of views in PotionDB: (i) Top-K; (ii) Max/min; (iii) Average; (iv) Sum. The idea is that, depending on the arguments given in the *create view* construct, a different type of CRDT will be generated.

Top-k should be used when the intention is to maintain a sorted set of data, e.g., sort customers by the total value they have spent. Depending on the use case, this can be used to maintain only the top entries (e.g., top 100 customers) or all. This kind of view can be generated with the following structure:

Create View name As
Select Limit number key(table1.a), table1.b, table1.c, ..., tableN.a, ...
From table1, ..., tableN
Where conditions
Order by table1.b, ...

The mencioned structure generates a Top-K CRDT [**?**]. *Limit* specifies how many entries should be present in the view (optional). *Key* takes one or more arguments and specifies which field(s) will be used to uniquely identify each entry in the top-k. *Order* specifies how are elements sorted in the top-k. Using *limit* is preferred, as in that case it's possible to leverage on non-uniform replication to reduce the amount of data required to be replicated and stored in each replica [**?**].

Max and min views are implemented by, respectivelly, max and min NuCRDTs. Since only the highest/lowest value needs to be stored, both these CRDTs leverage on non-uniform replication to avoid sending/storing updates which don't affect the visible state (i.e., the max/min value). Both kinds of views can be generated with the following structure:

Create View name As
Select max(table1.a)
From table1, ..., tableN
Where conditions

In *select* we specify which field is used to determinate the max/min entry using, respectivelly, the max or min functions. *From* and *where* have the same meanings as before.

andre: Max could also have the same structure as avg and max/min...?

Average is also implemented by a NuCRDT which keeps only the sum of all values and the number of values inserted, which is enough to calculate the average [**?**]. On the other hand, sum is implemented by a counter CRDT [**?**], which keeps the total added value by each replica. Both average and sum are generated by *create view* with similar structure:

Create View name As
Select table1.a, avg(table1.b)
From table1, ..., tableN
Where conditions
Group by table1.a, ...

The only change required to generate a sum view instead of an average view is to use *sum*() instead of *avg*() in the *select* clause. Group by is optional - if it isn't present, the avg/sum is calculated for all entries. Otherwise, it is calculated separately for each value of the tuples specified in the group by clause and stored in a map CRDT of either avg or counter CRDTs.

8.2 Updating views with local data

Building on the API presented in Section 6.2.2, we now describe how we solve the challenges presented at the start of this section.

The association between views and base data, as well as inclusion of objects created after the view, are both handled with the help of the *create table* construct. In practice this construct groups multiple CRDTs in a single key. When a *create table* construct is issued, CRDTs which match the specified keys and bucket receive a mark specifying the table (possibly more than one) which they belong to. Then, when an update is generated for one of those CRDTs, all marks (tables) associated to those CRDTs are notified of the update. The tables itself also keep a list of views which refer to it and, as such, the update notification is passed to these views and the respective view update is generated as described previously.

TopK
TopCustomers

id	int [key]
name	string
age	int
country	string
totalSpent	int [order]

Map of Avg
AvgSales

country	string
value	float

Max
HighestSale

saleId	int
value	int

Figure 2: Example views

To deal with the inclusion of objects, an extra change is required. When a new CRDT is added to the database, its key and bucket are checked with the existent tables. Then, for all tables to which there's a match, the new CRDT receives a mark, which guarantees updates to it will be reflected in existing views.

PotionDB ensures the views and base data present locally are always synchronized, that is, from the point of view of a client, when one is updated so is the other. This can be done by ensuring that whenever the base data is updated, all views associated to it are also updated in the same transaction. The atomicity of PotionDB's transactions ensures that either all updates on the transaction or none happen, which in turn guarantees that base data and their views are updated atomically.

8.3 Updating views with remote data

We now drop the assumption that all base data necessary for a view is present in the same replica. This leads to the challenge of keeping the views updated even if all the base data required to calculate the view isn't present in a replica.

In PotionDB this is solved by requiring a view (more precisely, its bucket) to be replicated in every replica which contains base data relevant to the view. This by itself is enough - since (i) updating the base data automatically generates a view update and (ii) updates in a bucket are replicated to every replica of that bucket and only to those, this implies that all updates of a view will be replicated to all servers which replicate the view, even if it was generated by an object only replicated in a subset of those servers.

andre: Is this explanation okay? Should I refer any other possible alternative and why our solution is better?

8.4 Generating view updates

andre: This section might need to be better worked on. I didn't want to include the "SQL code" used to generate each view as that would take too much space, but the explanation may be too generic/difficult to understand without that.

One key property of PotionDB is the automatic updating of views whenever their base data is updated. This implies that mechanisms for generating view updates based on base data updates are required. Updates are generated differently depending on the type of view.

Consider the example tables introduced in Section 3, along with three views whose scheme can be found on Figure 2. The views represent, respectively, (i) top 100 customers who spent the most (TopK); (ii) average value of sales in each country (Map of Averages); (iii) highest value of a single sale globally (Max). Even though all three depend on data from more than one table, updates on these views only need to be triggered whenever sales are created/modified/removed. This can be infered from the *create view* operation by considering, respectively, from which table comes the field in order by, max() and avg().

Assume the addition of the following sale:

Update sales2_5 sales MAP{*id: 10, custID: 2, productID: 5, amount: 2*}

andre: Problem with TopK example: the "totalSpent" is a sum... how do I reflect that?

This new sale generates the following updates on the views:

Update TopCustomers views TOPK{*id: sales2_5.id, totalSpent: += sales2_5.amount * product5.value, name: customer2.name, age: customer2.age, country: customer2.country*}

Update AvgSales views MAP{*country: customer2.country, value: Update AVG*{*value: sales2_5.amount * product5.value, count: 1*}}

Update HighestSale views MAX{*saleId: sales2_5.id, value: sales2_5.amount * product5.value*}

Most of the procedure for generating these updates is common to all kinds of views, which can be summarized in the following steps:

1. Fetch necessary CRDTs from the other tables mencioned in the *from* clause. Use the *where* condition to know how to select which objects are relevant (in this case, use the IDs in the sales CRDTs). This retrieval is efficient if the table with the trigger refers to other objects directly via, e.g., IDs.

2. Use the *select* clause to know which fields to pick from each CRDT.

3. Generate the update using as arguments the fields picked in the step before.

The specific parts that change depending on the view type are the type of update issued (e.g., MAX, TOPK, etc.) and dealing with the "special" fields of each view CRDT (e.g., in TOPK, the key and order fields). These specificities are evidencied in the example view updates presented before.

andre: Probably need a better way to conclude this.

9. IMPLEMENTATION

10. EVALUATION

In this section we conduct multiple experiments in order to evaluate PotionDB's performance. Namely, we try to assess the advantage of all replicas having views spanning data partitioned across multiple replicas, instead of each replica having an incomplete view with only its local data (local PotionDB). E.g., if we partition customer's data based on the continent of the customer's country, the local view of a server in Europe would only concern the data of european customers, while a global view would include customers from all over the globe. We also compare PotionDB with a solution that has one server dedicated to host all indexes (single index PotionDB).

10.1 Settings

10.1.1 Common settings

For our tests, we use a dataset based on TPC-H's dataset [5]. More precisely, while we keep all the tables and use data generated by the dbgen tool [5], we focus only on a subset of the queries and, for each table, we only keep the columns necessary for such queries. We also use the following settings in common across all of our tests:

- Five PotionDB instances, each one executing in its own node. Each node has a AMD EPYC 7281 CPU with 96GB of RAM, with PotionDB only being able to use 8 cores of the CPU (this helps ensure the bottleneck is not on the client side); **andre:** Should I keep this?

- Each PotionDB instance replicates the data of a continent, e.g., a server replicates all data concerning Europe while another keeps the data of Asia. Data that isn't location based such as products is replicated in all servers, as are indexes.

- Servers are interconnected by a LAN with 10 Gbps speed;

- Clients execute on nodes with 2x Intel Xeon E5-2620 v2 CPU and 64GB of RAM, being on the same LAN as the servers but with a reduced network speed of 1Gbps;

- Each client selects one server to execute queries upon, only contacting other servers if all the data required to reply to a query isn't present in the selected server. **andre:** Should I refer that, in practice, this only happens in the local views scenario? By default, the queries executed are 3, 5, 11, 14, 15 and 18 of the TPC-H specification [5];

- Unless otherwise specified, each request only executes one query or updates one sale (including all related indexes).

All our experiments execute as follows. First, all servers are started, with each one estabilishing a connection with all others. Afterwards, a special client is started to load the initial database state into the servers, taking into consideration the locality of the data. Finally, after all data has been loaded into the servers, multiple clients are started to execute queries and, possibly, updates, with the query/update rate depending on the test.

In our experiments we use as metrics the amount of operations per second (ops/s) achieved by all clients aggregated, as well as the average latency of each client request. To determine how much each test can scale, we vary the number of clients. As the number of clients increases, so does the latency of each request and, until the servers saturate, the throughput of the servers.

10.1.2 Scenarios

We make an extensive evaluation of PotionDB, conducting multiple experiments in order to evaluate how PotionDB scales in different scenarios. With these experiments, we try to answer the following:

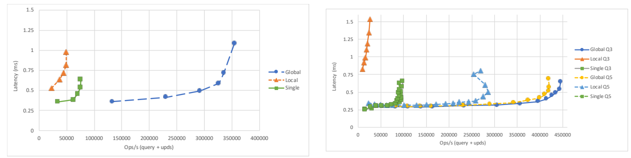
1. What is the practical benefit of having views of data replicated across different servers, compared to having only views of local data (local PotionDB) or a single index server (single PotionDB)?

2. What is the effect on PotionDB's throughput of executing updates, namely as the write/read ratio increases?

3. How much can PotionDB scale if we group operations?

4. **andre:** TODO: benchmarks

10.2 Results



(a) All queries

(b) Single query type

Figure 3: Query-only performance of PotionDB and its local and single index versions. On the left all queries are executed, while on right it is only Q3 or Q5.

10.2.1 Locality of data

andre: I wonder if it's worth mentioning that the number of clients server in single/local versions is much lower compared to normal.

To evaluate the benefits of having indexes replicated in every replica (1), we compare PotionDB with the local and single versions, namely in terms of queries per second.

Figure 3 shows the results of experiments for the three versions of PotionDB, in terms of query/s versus latency. Figure 3a includes all queries in the execution, while 3b only includes a single type of query at a time.

As expected, on both cases the performance of the local and single versions is worse than normal PotionDB. Focusing on figure 3a, it is noticeable that local version performance is about 7 times lower than normal. This happens because most queries require data from all regions, thus requiring local clients to contact all servers and wait for them all to reply. This leads to both more load on each server as well as more time waiting on the client side. The single scenario performance is roughly 1/5 of normal's. This can be explained as all queries only need to consult the indexes, which due to those being all in one server, implies that the load is all directed to one server instead of being split across five.

Figure 3b shows that the lower throughput of the local version is due to the locality of the data - Q3 requires data from all servers and has about 1/12 of the performance of normal PotionDB, while Q5 only requires data from one region (thus one server) and has about 2/3 of the performance. This smaller drop can be explained due to local clients not being sticky - as the query parameters are selected at random, local clients are forced to constantly query different servers, while each normal client can query always the same server.

10.2.2 PotionDB scaling - queries VS updates

We now evaluate the impact on PotionDB's throughput of executing updates alongside queries, with multiple read/write ratios (2). In this scenario, whenever a client wants to execute an operation, it picks at random whenever a query or an update is executed. It is worth noting that an update affects both base data and all associated indexes. Also note that while index updates are executed in every replica, we only count them once (i.e., if one update is executed 5 times, it still only counts as 1 operation).

Figure 4 shows the results of executing updates alongside queries, with varying odds of choosing an update. For low update rates (1% and 10%), the decrease in performance can be considered reasonable (aproximately 2,5% and 14%). For higher rates, two observations can be made: (i) the servers saturate earlier, in some cases even decreasing throughput as the number of clients rises; (ii) the max throughput is lower.

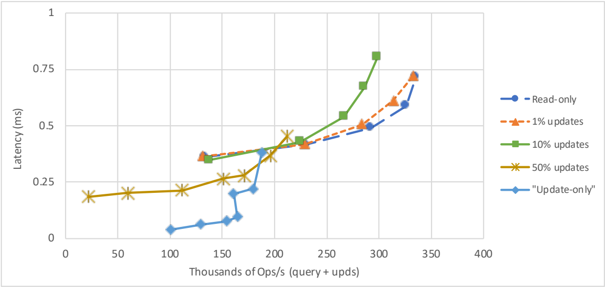
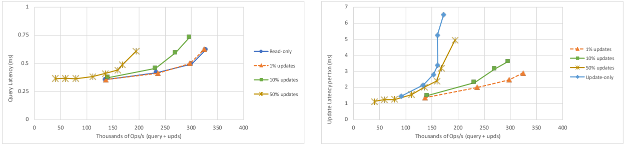


Figure 4: PotionDB performance with varying update rate.



(a) Query-latency

(b) Update-latency

Figure 5: PotionDB performance with varying update rate, with query and update latencies split. Note that updates are done in groups due to having to update both base data and indexes.

The referred observations can be explained if we consider the following. With PotionDB's internal partitioning, queries on different objects can be executed concurrently, thus query-only scenarios scale easily. However, updates in a transaction must lock the partitions they're involved and do a 2-phase commit, in order to ensure the states evolve sequentially in a replica even when concurrent transactions are issued. As such, if the update rate is high, clients executing queries or updates may often have to wait for a transaction with updates to finish executing before executing their own operations. **andre:** I might have to reffer in an early section with more detail how we do updates in our experiment, namely that our granularity is at order-level and that indexes, orders and items are updated in the same txn..

Figure 5 shows the same experiment as figure 4, but showcases the latencies of queries and updates splitted, instead of together. It is noticeable that updates have quite higher latencies and scale much less, which is due to both 2-phase commit and grouping of operations (as updates need to be reflected on both base data and indexes).

andre: Probably some concluding note that reffers that for PotionDB's intented usages it is likely that updates are much less frequent than queries, or maybe even that updates could maybe be delayed?

10.2.3 Batching

On the previous scenarios, each client only requested one query at a time, even through updates were executed in groups (whenever one update to a base object is used, all associated indexes are updated in the same transaction). With this we've noticed that, with a low number of clients, an update-only scenario actually has better throughput than a query-only scenario. Thus, we now evaluate the effect observable on PotionDB's throughput if multiple queries are included in the same transaction (3).

Figure 6 shows how PotionDB scales in terms of query/s depending on how many queries are executed in each transaction. As expected, as we increase the amount of queries per transaction, so does the latency, as there is more data to send, receive and process. However, the beneficts for throughput are visible - comparing 1 query/txn with 8 query/txn, we can see the throughput more than double before the server saturates, and latency increase by less

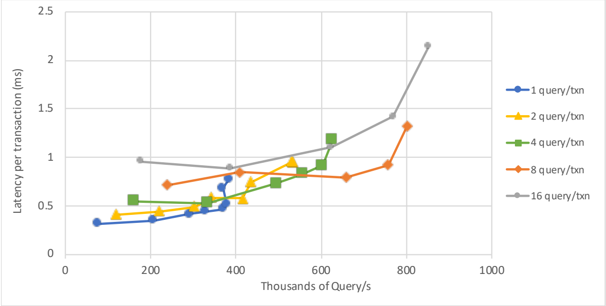
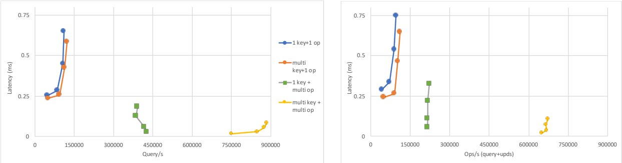


Figure 6: PotionDB performance when batching queries.



(a) Read-only

(b) 10% update rate

Figure 7: Single server performance when executing random operations on set CRDTs. 1 key corresponds to a single CRDT shared between all clients, while multi key corresponds to multi CRDTs per client. 1/multi op refer to single or multi operation transactions.

than double. Increasing the number of queries/txn also leads to PotionDB saturating with less clients, as more time is spent on processing queries instead of waiting/switching clients, thus making better usage of machine resources.

10.2.4 CRDT Benchmark

andre: I didn't know what to title this

We now evaluate how a single PotionDB server scales outside of the TPC-H scenario. Namely we do a few benchmarks to determine the raw throughput of PotionDB when executing random operations, instead of specific queries or updates.

Figure 7 contains results on the execution of benchmarks on set CRDTs. Namely, figure 7a concerns a read-only scenario, while 7b has a 10% update rate.

Focusing on figure 7a, it can be observed that increasing the number of operations per transaction greatly increases the throughput (up to 7x more in the tested scenario). Increasing the number of CRDTs only seems to have a considerable effect when there's multiple operations in a transaction. This may be due to the fact that a lookup operation in a set CRDT executes so quickly that most of the overhead may be in communication between threads or on converting data to the wire, thus not taking much benefit from the CRDTs being split across different partitions.

Considering figure 7b, considerably lower latencies can be observed when using multiple keys with only 1 operation per transaction. Based on other tests we done, we can affirm that this difference is bigger the higher the rate of updates are. This is due to the fact that updates must lock a partition, thus if different keys are used, and each client only manipulates one key at a time, some reads may be executed in parallel with updates, as long as it is for different partitions. With a single key used by all clients, no concurrency is possible when an update is being executed.

The observed performance is worse than in a read-only scenario, which is specially noticeable in scenarios with multiple operations per transaction. IThis is even more noticeable in the case with only one key, as the vast majority of transactions will include updates and thus the different clients end up having their transactions executed sequentially. Having multiple keys eases this, as only partitions which get updated need to be locked, thus some concurrency is still possible.

andre: This subsection VERY LIKELY needs to be rewritten/better thought of. Maybe even choose different data to show...

11. RELATED WORK

Both strongly and weakly consistent solutions exist to support services that require geo replication of their data. Strong consistency solutions like Spanner [4] and CockroachDB [17] provide the ilusion of a single replica, thus making it easier to provide a consistent view for clients. However, the CAP theorem [8] implies that availability may be compromised in case of network partitions, and high latency is expected as multiple datacenters need to be contacted for executing operations. Weak consistent solutions such as Dynamo [7] and COPS [12] can provide highly available, low latency operations, but providing a consistent view of the database to clients is challenging. Causal consistency is a sub-form of weak consistency that alleviates this problem, however anomalities can still be observed even with cross-object causality [12, 2].

Systems like Dynamo [7], COPS [12] and CockroachDB [17] provide geo-replication. Dynamo provides partial replication, however reads and updates for multiple keys are done independently, thus there's no causal read of multiple reads. COPS provides causal+ consistency and supports transactions for reads, however it doesn't support neither partial replication or views. CockroachDB is strongly consistent and, thus, may fail due to network partitions, and does not provide partial replication.

CRDTs [16] are replicated data types that guarantee state convergence, assuming all updates are eventually delivered. Thus, they are often used in weakly consistent systems, e.g., Redis [6] and Riak [1]. Some types of CRDTs have been introduced that can help with representing views of data, namely computational CRDTs [13] and non-uniform CRDTs [3]. The former computes some result over data (e.g., a sum), while the later focus on minimizing the information that needs to be replicated to correctly reply to queries (e.g.: a top-k CRDT doesn't need all entries to be replicated). We leverage on both in our solution.

Alternative solutions to provide global queries on partially replicated data includes using distributed processing systems like Pixida [10] or Hourglass [14]. However, this impose challenges both consistency-wise, as well as in terms of latency and data transferred. The amount of data to be transfered is specially concerning, as if the underlyng systems only provide simple get operations, very high amounts of data may have to be transfered to reply to a small query like a top 10. PotionDB avoids this by having materialized views, which allows to have only the required data replicated in every server and thus reply to such query efficiently.

12. CONCLUSIONS

13. REFERENCES

[1] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak dt map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, pages 1–1, 2014.

[2] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. *Microsoft Research Technical Report MSR-TR-2013-39*, 2013.

[3] G. Cabrita and N. Preguiça. Non-uniform Replication. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS 2017)*, 2017.

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

[5] T. P. P. Council. Tpc benchmark h (decision support) standard specification revision 2.18.0. `tpc.org/tpch/`, 2018.

[6] L. Cro. Diving into crdts. `https://redislabs.com/blog/diving-into-crdts/`, Apr 2020.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, Washington, USA, 2007. ACM.

[8] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[9] Gomez. Why web performance matters: Is your site driving customers away?, 2018. Accessed May/2018.

[10] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues. Pixida: optimizing data parallel jobs in wide-area data analytics. *Proceedings of the VLDB Endowment*, 9(2):72–83, 2015.

[11] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *Proceedings 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, Prague, Czech Republic, 2013. ACM.

[12] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, 2011. ACM.

[13] D. Navalho, S. Duarte, and N. Preguiça. A study of crdts that do computations. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–4, 2015.

[14] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE, 2006.

[15] E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at velocity web performance and operations conference. `http://slideplayer.com/slide/1402419/`, 2009.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[17] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[18] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, Jan. 2009.

14. TOPICS

This contains the topics that were initially discussed during the first meeting and some afterthoughts.

15. SYSTEM OVERVIEW

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

16. IMPLEMENTATION

17. EVALUATION

18. RELATED WORK

19. CONCLUSIONS

20. SYSTEM OVERVIEW

Possiveis pontos mais detalhados?

20.1 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)

20.2 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?)

20.3 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?

andre: I ended up describing the topics of system description in other subsections, apart from Structure. I don't recall going into much detail of what a CRDT is, but that shouldn't be necessary anyway.

21. 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)

andre: A good part of the implementation is already included in other sections, at least indirectly. Mainly Replication and to an extent Transactions/Communication. We need to decide what really is important to refer in the "implementation" section.