Global-Scale Placement of Transactional Data Stores

Victor Zakhary[1], Faisal Nawab[2], Divyakant Agrawal[1], Amr El Abbadi[1]
[1]University of California, Santa Barbara, CA 93106
[victorzakhary,agrawal,amr]@cs.ucsb.edu
[2]University of California, Santa Cruz, CA 95064
fnawab@ucsc.edu

ABSTRACT

Global-Scale Data Management (GSDM) empowers systems by providing higher levels of fault-tolerance, read availability, and efficiency in utilizing cloud resources. But, *at which datacenters should data be placed?* Current cloud providers offer tens of datacenters and hundreds of edge datacenters that are globally distributed all over the world. Unlike networks within a datacenter, the topology of the Wide-Area Network (WAN) is asymmetric and diverse—the latency connecting a pair of datacenters can be an order of magnitude larger than the latency connecting another pair. This makes placement a significant factor in performance. However, it is not only placement. The specifics of the transaction management protocol play a crucial role in deciding which placement is ideal. In this paper, we develop GPlacer, a placement optimization framework that embeds the transaction protocol constraints into an optimization to derive both the data placement and the transaction protocol configuration that minimize the overall transaction latency. In developing GPlacer, we discover counter-intuitive lessons about data placement and transaction execution practices. Our evaluation shows that applying these lessons in addition to known best practices generate deployments that reduce the average transaction latency by up to 68%.

1 INTRODUCTION

Internet applications strive for high-performance 24/7 service to clients dispersed around the world. Achieving this is threatened by complete datacenter outages; either planned or unplanned. To overcome these challenges, application services and their backend databases are increasingly being deployed on multiple datacenters spanning large geographic regions (*geo-replication*). F1 [29], Spanner [11], and Tao [8] are examples of deployed systems that are geographically replicated for fault-tolerance and performance reasons.

Moving to Global-Scale Data Management (GSDM), despite its benefits, raises many challenges that are not faced by traditional deployments. The large WAN communication latency is orders of magnitude larger than the traditional LAN communication latency. Figure 1 illustrates the latency difference between communication messages that occur within the same machine, among different machine in the same datacenter, or in multiple datacenters in different geographical regions. This large communication latency of the WAN motivates systems like Yahoo's PNUTS [9], Facebook's Tao [8] and others [17, 24] to trade off replica consistency and/or multi-row transaction support with high availability and scalability. However, enterprise applications and applications with complex and evolving schemas have more interest in data management systems that provide transactional

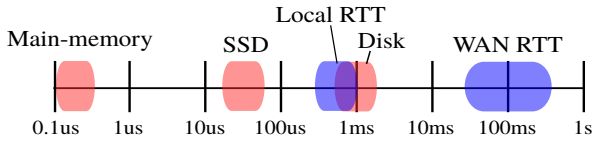


Figure 1: Latency of the Wide-Area Network Round-Trip Time communication (WAN RTT) compared to memory access latency [27] and network latency within the datacenter (local RTT).

ACID properties [5, 11, 30]. Application developers spend significant time to build transaction semantics and complex mechanisms, that are error-prone, on top of the eventual consistent datastores in order to handle stale data items and reason about inconsistency [11, 29]. Therefore, in the past few years, many solutions have emerged to provide strongly consistent transactions for geo-replicated databases [11, 16, 18, 21–23]. These solutions use different replication and isolation techniques in order to minimize the number of WAN messages required to achieve strong ACID transactional guarantees for geo-replicated databases, hence reducing the transaction latency.

Data placement is the problem of deciding the subset of datacenters to host a full or a partial replica of the data to achieve a certain objective such as minimizing the transaction latency, minimizing the deployment monetary costs, and any combination of these and other user-defined objective functions.

In this paper, we propose GPlacer; an optimization framework that solves the data placement problem. GPlacer embeds the commit protocol constraints into an optimization to derive both the data placement and the commit protocol configurations that minimize the overall transaction latency. In developing GPlacer, we discover counter-intuitive lessons about data placement and transaction execution practices. These lessons exploit the latency diversity and asymmetry of the WAN links and are widely applicable to Paxos-based commitment protocols [13, 21] and leader-based commitment protocols [5, 11]. GPlacer incorporates these lessons, the commitment protocol constraints, and the application requirements in an optimization to find the placement that minimizes the average transaction latency.

WAN links are diverse and asymmetric; a link connecting a pair of datacenters can be an order of magnitude larger than a link connecting another pair. Table 1 shows the average measured Round-Trip Time (*RTT*) between every pair of nine Amazon AWS datacenters in **C**alifornia (*C*), **O**regon (*O*), **V**irginia (*V*), **S**ão **P**aulo (*SP*), **I**reland (*I*), **Sy**dney (*Sy*), **Si**ngapore (*Si*), **T**okyo (*T*), and **Se**oul (*Se*). As shown, the average RTT between California and Oregon datacenters is 22ms while the average RTT between Singapore and São Paulo datacenters is 329ms. Therefore, the number of WAN messages required per transaction is not the only factor that dominates the transaction latency. Transaction latency is a product of both the transaction commit protocol, which

	C	O	V	I	Si	T	Se	Sy	SP
C	0(1)	22(2)	65(13)	136(5)	189(12)	113(5)	142(12)	159(2)	185(11)
O	22(2)	1(1)	88(14)	125(2)	166(13)	101(11)	131(13)	178(3)	182(11)
V	65(13)	88(14)	1(16)	73(13)	220(22)	156(16)	179(20)	219(13)	121(16)
I	136(5)	125(2)	73(13)	0(0)	180(18)	211(10)	233(14)	301(5)	185(12)
Si	189(11)	166(12)	220(22)	180(17)	1(9)	68(8)	97(13)	169(8)	329(21)
T	113(5)	101(11)	156(18)	211(10)	68(9)	0(3)	32(9)	104(2)	263(15)
Se	142(9)	131(13)	179(20)	233(13)	97(13)	32(10)	1(9)	133(8)	290(16)
Sy	159(2)	178(3)	219(12)	301(5)	169(10)	104(2)	133(8)	1(0)	338(11)
SP	185(13)	182(12)	121(17)	185(13)	329(23)	263(16)	290(18)	338(14)	1(11)

Table 1: The average RTT latencies between different datacenters in milliseconds and the standard deviation inside parentheses.

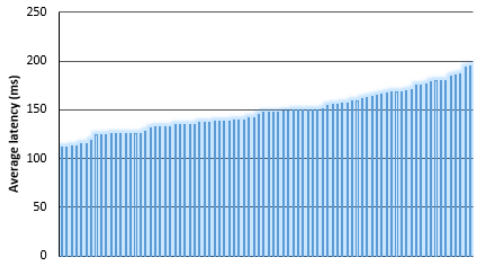


Figure 2: The average latency, of all the clients in 9 datacenters, to reach the closest quorum (2 out 3) for all the possible $\binom{9}{3} = 84$ different placements sorted by latency.

controls the number of the WAN messages required per transaction, and the locations of the replicas, hence the placement, which controls the latency per a WAN message. To illustrate the placement effect on the average obtained transaction latency, we hold the following experiment. We equally distribute clients among the nine AWS datacenters. Three out of the nine datacenters are chosen to host a data *replica*. The time to reach the closest quorum, two replicas out of these three, is measured for all the clients for all the **possible placements** and the average latency is reported. Figure 2 shows the effect of only changing the placement on the average obtained latency for all the clients while fixing the protocol. As seen in Figure 2, changing only the placement while fixing all the other parameters (the protocol, the workload distribution, etc.) can lead to a significant change of 1.75x between the minimum and the maximum reported average latency. This latency difference amplifies for real workloads when transactions are executed in chains [20].

Unlike GPlacer that optimizes the placement for *multi-row transactional workload with strong consistency requirements*, many works focus on optimizing the placement for weaker consistency levels and single-row operations. SPANStore [33] develops an optimization framework to optimize the monetary cost of deploying a geo-replicated *key/value* store. This framework optimizes the total cost of processing, storage, bandwidth, and I/O and finds the placement that achieves the minimum overall cost while meeting the application requirements. Liu et al. [19], like SPANStore, optimize the deployment monetary cost. However, they consider cost savings exploiting resource reservation payment model instead of the pay-as-you-go payment model while avoiding over reservation. Ping et al. [26] propose the use of a utility function to derive a placement that achieves a balance between the availability and the speed of data access. Volley [4] analyzes data access logs and generates a migration plan for data partitions to minimize the access latency.

Sharov et al. [28] optimize the placement for *strong consistent transactions* using *leader-based* protocols. Sharov assumes that a database is sharded into multiple *partitions* and each partition is **replicated** independently. Each partition has a **leader replica** that serializes all the transactions that span this partition to achieve isolation. This leader replicates the updates to a *majority quorum* of the partition replicas to achieve fault tolerance. Although they provide placements for strong consistent transactional workloads, their optimizations are tightly coupled with leader-based protocols and it does not apply to the many non-leader-based protocols that are widely used such as [6, 13, 16, 21, 25]. Also, their resulting optimal placement allocates all the partition leaders together in one datacenter. Placing all partition leaders in one datacenter introduces the risk of losing access to the entire data until the leaders are re-elected. In addition, the transactions that span a single partition might incur higher latency than the latency observed when the leader of each partition is placed closer to the clients that access this partition.

The rest of the paper is organized as follows. Section 2 explains the transaction model, the client requests, and the assumptions and limitations of application requirements. Although GPlacer can optimize the placement for different classes of commitment protocols, a Paxos-based protocol is used to explain the details of GPlacer. Section 3 formalizes the placement problem into an exhaustive search problem. Although the exhaustive search finds the optimal placement, it does not efficiently scale with the number of datacenters. Therefore, we introduce several placement heuristics that find sub-optimal placements while efficiently scale with the number of datacenters. Section 4 describes the counter-intuitive lessons learned during the development of GPlacer and their effect on the transaction latency. In Section 5, we evaluate the effect of the placement lessons on the transaction latency and the abort rate. We also evaluate the output and the performance of the proposed heuristics compared to the exhaustive search. In Section 6, we explain the changes that need to be done to extend GPlacer to optimize for other protocols. The paper is concluded in Section 7.

2 BACKGROUND

Global-scale placement is the problem of deciding which datacenters will store a full or a partial replica of an application's data subject to a certain objective function. Objective functions can vary between minimizing the deployment monetary cost [19, 33] or minimizing the data access latency for a defined set of client operations [28]. Objective functions are always constrained by the application requirements (e.g., availability, upper bound access time, or bandwidth usage). In this section, we present our storage model and our assumptions about the workload distribution, the application requirements, and the objective function.

The universe of datacenters, denoted by DC, is defined as all the datacenters that can host an application[1] instance and/or a replica[2] of the database. We assume that the application is deployed on a subset of the datacenters $DC_{app} \subseteq DC$. The clients of the application are scattered around the globe and for simplicity, we assume that clients are collocated with their closest datacenter. The application is deployed in all the datacenters that have clients. However, these datacenters can be different

[1] *Application* refers to the middle tier logic.
[2] *Replica* refers to a copy of the backend database.

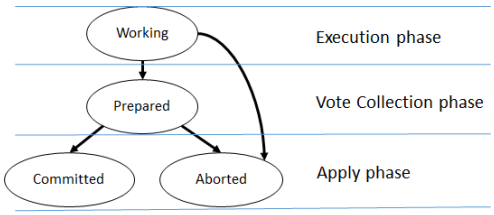


Figure 3: The state-transition diagram of a distributed transaction adopted from [13].

from the datacenters that host replicas. $DC_{db} \subseteq DC$ is the subset of datacenters that host a database replica. We assume that the database is *partitioned* and all the partitions are *fully replicated* in DC_{db}.

2.1 The Transaction Management Protocol

The clients of the application access the globally-distributed storage by issuing transactions, which are collections of read and write operations followed by a commit or an abort. GPlacer considers transactions with *strong guarantees*, *i.e.*, serializability [7]. Strong consistent transactions on globally-distributed data require more coordination than weaker forms of access like eventual consistency or single-key atomicity— thus making strongly consistent transactions more expensive. A strong consistency transactional interface is more natural to programmers and is required by many applications. Thus, we adopt such strong access semantics for GSDM as others did from both academia [16, 21] and industry [11].

We adopt the distributed transaction model proposed by Gray and Lamport [13]. Figure 3 shows the different states of a transaction and the corresponding execution phases. A client drives the execution of a transaction in three phases. The details of these three phases differ across different transaction management protocols. However, the abstract semantics behind these phases are the same for all the protocols that provide the same strong transactional guarantees. The three phases of a transaction are: the execution phase, the vote collection phase, and the apply phase.

During the execution phase, the transaction is in the working state when read and write operations are processed. We assume that writes are locally buffered at the client and the updates are sent to the data replicas in the second phase. This assumption is widely used in many geo-replicated transaction management protocols [11, 16, 21]. For a read operation, clients communicate with their read coordinator, r_c. r_c processes a read request and responds back to the client. The RTT between a client c and r_c is denoted by RTT_{c-to-r_c} and the time for r_c to process the read request is denoted by P_{r_c}. The total execution phase latency is denoted by $L_e = n_r.(RTT_{c-to-r_c} + P_{r_c})$ where n_r is the average number of read requests per transaction. The transaction management protocol determines the values of n_r, RTT_{c-to-r_c}, and P_{r_c}. Some protocols assume that the client is the read coordinator. In such case, $RTT_{c-to-r_c} = 0$. Also, some protocols require that the client issues read requests one by one and others require that the client should batch all the reads in one request. The processing time P_{r_c} depends on how many replicas r_c should communicate with to serve a read request. In our model protocol, r_c has to communicate with a majority quorum to serve each read (we also consider read optimizations later in Section 2.2. As write requests are locally buffered, their effect on the execution phase latency is negligible. During the execution phase, a client might decide to abort the transaction by simply moving the transaction to the aborted state. However, if the client decides to commit the transaction, the transaction is moved to the prepared state and the vote collection phase starts.

During the vote collection phase, the client sends the transaction's details to the commit coordinator c_c which is responsible for coordinating with the other replicas to decide either to commit or to abort the transaction. Typically, the c_c uses either two-phase commit (2PC) with two-phase locking (2PL) [11] or quorum-based approaches (*e.g.*, Paxos) with 2PL [21]. The vote collection phase can be mapped to the first phase of the 2PC or the first round of Paxos. The latency of the voting phase is denoted by $L_v = \frac{RTT_{c-to-c_c}}{2} + RTT_{c_c-to-p}$ where RTT_{c-to-c_c} is the RTT between c and c_c and RTT_{c_c-to-p} is the round-trip time between the c_c and the furthest participant p included in the voting process.

If the decision of the vote collection phase is to abort, the client is notified, the transaction is moved to the aborted state, the other participants are asynchronously updated, and the obtained locks are released. However, if the decision is to commit, c_c starts the apply phase by sending the apply message to all the participants. Upon receiving the apply message, the participants commit the transaction, release the locks, and respond back to the coordinator. c_c notifies the client and the transaction is moved to the committed state. The latency of the apply phase is denoted by $L_a = RTT_{c_c-to-p} + \frac{RTT_{c-to-c_c}}{2}$ as the apply phase takes a round of communication with the participants in addition to the time to inform the client about the decision. A transaction commit latency L_c is the time spent in the vote collection phase and the apply phase combined: $L_c = RTT_{c-to-c_c} + 2 \cdot RTT_{c_c-to-p}$. The transaction latency L_t is the time from the beginning till the end of a transaction: $L_t = L_e + L_c = n_r \cdot (RTT_{c-to-r_c} + P_{r_c}) + RTT_{c-to-c_c} + 2RTT_{c_c-to-p}$.

GPlacer optimizes the average overall transaction latency over all the clients in different datacenters. It is designed to optimize the placement for a wide class of transaction commitment protocols. In this paper, we focus on optimizing for multi-master Paxos-based protocols [13, 21] and for leader-based protocols [11] both on *partitioned fully replicated databases*. In multi-master Paxos, each replica can act as the commitment coordinator role and uses the two rounds of Paxos for both transaction isolation and replication. However, in leader-based protocols, a transaction can fall into one of two categories: single-partition transactions or multi-partition transactions. Single-partition transactions span only one partition and the isolation between transactions that span this partition is managed by the leader of this partition. Multi-partition transactions span multiple partitions and typically 2PC is used between the leaders of the partitions involved in a transaction to achieve isolation. In both categories, partition leaders replicate the updates of committed transactions to a majority quorum of their partition replicas using only the second round of Paxos.

In Section 3, we formalize GPlacer. We use Replicated Commit [21] as our protocol model where reads are served from a majority of the replicas and commits are done using the two rounds of Paxos for isolation and replication. In Section 2.2, we explain some commonly used optimization to reduce the execution phase latency. In Section 6, we explain how to extend GPlacer to optimize placement for leader-based protocols like Spanner [11].

2.2 Read optimizations

In this section, we present two widely-used read optimizations that are considered in GPlacer. A read request latency $L_r = (RTT_{c-to-r_c} + P_{r_c})$. The first optimization, **optimistic read**, aims to eliminate the read processing time P_{r_c}. The second optimization, **passive replica read** aims to eliminate the time to reach the coordinator RTT_{c-to-r_c} and the processing time P_{r_c} by bringing a copy of the data to the client's datacenter. We define two different types of replicas a datacenter can host: *active replica* or *passive replica*. An active replica contributes synchronously in the voting collection and the apply phases and can act the coordinator and the participant roles. However, a passive replica is a read-only replica. It is asynchronously updated after the transactions are committed.

Optimistic read aims to eliminate the read request processing time by optimistically reading data values from the closest active replica without any coordination with other active replicas. This optimization has been introduced before as early as in Postgres-R local reads [15] and as fast reads in Zookeeper [14]. Applying optimistic reads require validating the value read in the commit phase to guarantee the freshness of the optimistically read values in the execution phase. In Spanner [11], reads are served by the leader of each partition. However, optimistic reads can be beneficial by reading from the closest partition replica instead from the partition leader. In Replicated Commit [21], a client is required to read from a quorum of the replicas. Applying optimistic read reduces the read latency by reading from one replica instead of a quorum.

Passive replica read aims to completely eliminate the read latency by processing read requests from a local read-only replica or a **passive replica**. The reason behind this naming is that a passive replica does not participate actively in the commit decision. Therefore, adding more passive replicas does not affect the commit latency. However, these replicas need to be asynchronously updated which increases the bandwidth required per committed transaction. Also, having many passive replicas increases the deployment cost. Data read from a passive replica needs to be validated in the commit phase to guarantee freshness. If the data is frequently updated at the active replicas, the data values read from a passive replica will be stale which increases the transaction abort rate. The concept of passive replica read has also been introduced in [28] as *weak reads*.

GPlacer chooses the set of active replicas and the set of passive replicas. In addition, it assigns r_c and c_c for clients in every datacenter. Application requirements are given as inputs to the framework. GPlacer takes as an input the fault tolerance level f, the total number of replicas t, and the workload distribution. f determines the number of active replicas and t determines the number of passive replicas. The workload distribution determines which datacenters should have active replicas, which should have passive replicas, and which should not have a replica at all. GPlacer finds placements that optimize the overall average transaction latency for strongly consistent multi-row transaction workloads. However, systems that require non-transactional or weakly consistent operations can easily be tuned in GPlacer's prototype but we do not discuss them since they were treated in previous works [4, 33].

3 FRAMEWORK FORMULATION

GPlacer finds the placement that minimizes the average transaction latency for partitioned fully replicated databases. As explained in Section 2, Paxos-based protocols use majority quorums for both transaction isolation and replication while leader-based protocols use majority quorums only for replication. Placement for Paxos-based protocols requires finding the subset of datacenters that should host replicas and the majority quorums used by the protocol. Leader-based protocols requires an additional step of placing the leaders of different database partitions on the replicas chosen in the first step. In Section 3.1, we formulate the placement problem into an exhaustive search model for Paxos-based protocols. This model evaluates all the possible placement combinations and returns one placement that achieves the minimum average transaction latency for a given workload. The model finds the placement $DC_{db} \subseteq DC$ and the majority quorums for each replica in this placement that optimizes the objective function. Although the model finds the optimal placement, due to the model complexity, it does not scale with the number of datacenters *when multiple cloud providers and edge datacenters are considered*. Therefore, in Section 3.2, we introduce two replica-placement heuristics to find placements that are close to optimal among hundreds of datacenters. The performance and the resulting placements of these heuristics are evaluated in Section 5.

3.1 Model formulation

The **inputs** of GPlacer fall into *two* categories:

- **Datacenter information**: this includes the number of datacenters $|DC|$ and the average RTT between every pair of the datacenters.
- **Application information**: this includes the number of datacenter scale outages f the deployment should tolerate and the application workload distribution. The workload distribution is denoted by c_i and represents the number of clients c at datacenter i.

The **outputs** of GPlacer include:

- The list of datacenters that should host a database replica.
- The read and the commit coordinator of clients at each datacenter. Clients at one datacenter share the same read and commit coordinators.

As the placement problem can be represented as an optimization model, we first implemented the placement model as an integer program and used the open source GLPK solver [2]. However, the solver could not efficiently scale with the number of datacenters. Many of the optimization constraints are conditional and to convert them to linear constraints, multiple binary output variables are introduced. The binary outputs and their related constraints are quadratic in the number of the datacenters $O(|DC|^2)$. In addition, GLPK solver introduced performance overhead. Therefore, to conduct a fair comparison with the replica-placement heuristics, we implement both the exhaustive search and the heuristics in Java. The **objective function** of the placement model is to minimize the average transaction latency of all the clients in all the datacenters. Algorithm 1 shows the details of the exhaustive search model.

Algorithm 1 evaluates all the possible subsets of the input datacenters of size $2f + 1$ and returns the one that minimizes the average transaction latency. The function *evalLat*, in line 3,

Algorithm 1 Evaluates all the possible placement combinations and returns the one that achieves the minimum average latency for given application requirements.

Input: f, $|DC|$, RTT_{ij} $\forall i, j \in DC$ and the Set $C = \{c_i \ \forall i \in DC\}$
Output: DC_{db}, DC_{rc}, and DC_{cc}

1: $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$, $minL \leftarrow MaxInt$
2: **for each** Set $S \subset DC, |S| = 2f + 1$ **do**
3: $l, S_{rc}, S_{cc} \leftarrow evalLat(S, RTT, C)$
4: **if** $l < minL$ **then**
5: $minL \leftarrow l$, $DC_{db} \leftarrow S$
6: $DC_{rc} \leftarrow S_{rc}$, $DC_{cc} \leftarrow S_{cc}$
7: **end if**
8: **end for**

has different implementations based on the enabled read optimizations. When all the read optimizations are disabled, *evalLat* assumes that the read coordinator and the commit coordinator are collocated with the client who issues a transaction and reads are served from a majority quorum of replicas. However, if *optimistic read* is enabled, the read latency is updated to the RTT to the closest chosen replica from the client. Also, if *passive replica read* is enabled, the read latency is updated to zero as all the clients perform read operations from a local replica.

3.2 Replica-placement heuristics

Although Algorithm 1 finds the optimal placement among all the possible placements, it does not efficiently scale when the total number of the datacenters, $|DC|$, or the number of the replicas, $|DC_{db}|$, increases. Our experiments show that choosing 7 replicas out of 60 datacenters $\binom{60}{7}$ takes 2 hours while choosing 7 replicas out of hundreds of datacenters (which is the case when we consider edge datacenters) could take years. Therefore, we present two replica-placement heuristics that efficiently find placements with sub-optimal average transaction latency. These replica-placement heuristics consider the *two main aspects* that affect the transaction latency; the latency between the clients and the replicas and the latency between the replicas each other. The running-time of these heuristics is polynomial in the total number of the datacenters. The performance and the resulting placements of these heuristics are compared to the exhaustive search results in Section 5.2.

The first replica-placement heuristic is shown in Algorithm 2. It uses an iterative greedy algorithm to choose the replicas. It starts with an empty set of chosen replicas $DC_{db} \leftarrow \{\}$, line 1, and at each iteration, it adds one replica to DC_{db} until $2f + 1$ replicas are chosen. The inner loop, lines 4-14, evaluates the effect of adding each unchosen replica to DC_{db} on the average transaction latency and the replica that achieves the minimum latency is added to DC_{db}, line 15. *evalLat* is the same evaluation function introduced in Algorithm 1 line 3. The intuition behind this heuristic is that choosing the best candidate at each step should lead to a solution that is optimal or close to the optimal.

The second replica-placement heuristic is presented in Algorithm 3. It is based on the K-Means algorithm. It assigns weights to every datacenter, initially equals to the number of clients in this datacenter; line 4. A datacenter weight is updated according to the number of quorums it participates at; line 13. Datacenter weights are iteratively updated and datacenters are sorted by their weights. The top $2f + 1$ datacenters are chosen to host replicas in lines 5 and 18. The algorithm evaluates the placement

Algorithm 2 Greedily adds one replica at a time achieving the minimum average latency at each iteration.

Input: f, $|DC|$, RTT_{ij} $\forall i, j \in DC$ and the Set $C = \{c_i \ \forall i \in DC\}$
Output: DC_{db}, DC_{rc}, and DC_{cc}

1: $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$
2: **while** $|DC_{db}| < 2f + 1$ **do**
3: $S \leftarrow DC_{db}$, $minL \leftarrow MaxInt$, $minDC \leftarrow \phi$
4: **for all** $dc \in DC$ **do**
5: **if** $dc \notin S$ **then**
6: $S \leftarrow S \cup \{dc\}$
7: $l, S_{rc}, S_{cc} \leftarrow evalLat(S, RTT, C)$
8: **if** $l < minL$ **then**
9: $minL \leftarrow l$, $minDC \leftarrow dc$
10: $DC_{rc} \leftarrow S_{rc}$, $DC_{cc} \leftarrow S_{cc}$
11: **end if**
12: $S \leftarrow S \setminus \{dc\}$
13: **end if**
14: **end for**
15: $DC_{db} \leftarrow DC_{db} \cup \{minDC\}$
16: **end while**

in every iteration and stops when the average transaction latency converges. To avoid fast convergence to a local minimum, a minimum iteration count is required before terminating the algorithm; lines 1 and 7. The minimum evaluated placement is saved to make sure that the final placement does not achieve higher transaction latency than any placement that has been evaluated before.

Algorithm 3 Assigns weights to datacenters and iteratively chooses the top weighted $2f + 1$ to host replicas.

Input: f, $|DC|$, RTT_{ij} $\forall i, j \in DC$, t and the Set $C = \{c_i \ \forall i \in DC\}$
Output: DC_{db}, DC_{rc}, and DC_{cc}

1: $minIter \leftarrow t$, $iter \leftarrow 0$
2: $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$
3: $l_{n-1}, l_n \leftarrow MaxInt$
4: $Weights \leftarrow \{c_0, c_1, ..., c_{|DC|}\}$ // Initialize weights with the number of clients at each datacenter.
5: $DC_{db} \leftarrow top(sort(Weights), 2f + 1)$ // Sort on weights and choose a placement of the top $2f + 1$.
6: $l_n, DC_{rc}, DC_{cc} \leftarrow evalLat(DC_{db}, RTT, C)$
7: **while** $l_n < l_{n-1}$ || $iter++ < minIter$ **do**
8: $l_{n-1} \leftarrow l_n$
9: $NewW \leftarrow \{0, 0, ..., 0\}$ // New Weights
10: **for all** $dc1 \in DC$ **do**
11: **for all** $dc2 \in DC$ **do**
12: **if** $dc2 \in nearestQuorum(dc1)$ **then**
13: $NewW[dc2] += Weights[dc1]$
14: **end if**
15: **end for**
16: **end for**
17: $Weights \leftarrow NewW$
18: $DC_{db} \leftarrow top(sort(Weights, 2f + 1)$
19: $l_n, DC_{rc}, DC_{cc} \leftarrow evalLat(DC_{db}, RTT, C)$
20: **end while**

4 SURPRISING PLACEMENT LESSONS

During the development of GPlacer, we learned some counter-intuitive lessons about data placement that exploit the diversity

and the asymmetry of the WAN links to decrease the execution and the commit latencies, hence the transaction latency. (The transaction latency L_t is sum of the execution latency L_e and the commit latency L_c.) In this Section, we explain the details of these placement lessons and their effect on the transaction latency.

4.1 Request handoff

A client executes either read or commit requests. The latency of these two requests can be abstracted as the sum of: RTT_{c-to-c}, the round-trip time between the client and the request coordinator and L_p, the time for the coordinator to process the request.

Therefore, the request latency is mainly affected by the distance between the client and the coordinator, the distance between the coordinator and the participants, and finally the number of communication rounds required between the coordinator and the participants to serve the request. Different transaction management protocols choose the coordinator based on some intuitive heuristics. In [21], Mahmoud et al. assume that the *client* is the coordinator of a transaction. In Spanner [11], the 2PC coordinator is randomly chosen from the leaders of the partitions involved in a multi-partition transaction. In [23], Nawab et al. choose the coordinator to be the closest replica to the client. However, the choice of the coordinator can drastically affect the request latency. To illustrate this effect, we provide two examples of 2PC and Paxos deployments to show that carefully choosing the coordinator can save up to 48% of the average latency.

Two-phase commit: assume there are three data partitions X, Y, and Z deployed in three AWS datacenters in SP, V, and I respectively. Now, assume a client in datacenter I wants to commit a transaction t that updates the elements $x_1 \in X$, $y_1 \in Y$, and $z_1 \in Z$. The commit latency at any coordinator equals to double the RTT between the coordinator and the furthest involved partition leader. Therefore, if the client chooses the leader of partition Z in datacenter I to be the commit coordinator, the resulting commit latency is $2 \cdot max(RTT_{IV}, RTT_{ISP}) = 2 * 185 = 370ms$. Although the time between the client and the coordinator is neglected, the latency is still high because the coordinator is relatively far from SP. However, if the client chooses the leader of partition Y in datacenter V to be the commit coordinator, the resulting commit latency is $RTT_{IV} + 2 \cdot max(RTT_{VI}, RTT_{VSP}) = 73 + 2 * 121 = 315ms$ saving around 15% of the commit latency without modifying any constraint of the original 2PC protocol. Also, when datacenter V is the 2PC coordinator, the participant at datacenter I will be notified about the commit decision after $\frac{RTT_{IV}}{2} + max(RTT_{VI}, RTT_{VSP}) + \frac{RTT_{IV}}{2} = 36.5 + 121 + 36.5 = 194ms$. The participant at datacenter I can directly inform the client with the decision saving around 48% of the latency obtained when I is chosen to be the coordinator.

Paxos: assume there are five replicas of the database in datacenters I, V, SP, O, and C as shown in Figure 4. A client in datacenter SP wants to commit a transaction which requires to execute the two rounds of Paxos to reach a consensus about the commit decision. The latency of the two rounds of Paxos equals to double the RTT between the coordinator and the furthest replica in the closest majority to the coordinator. Therefore, if the client in SP chooses the replica in SP to be the coordinator, the resulting commit latency equals to $2 \cdot max(RTT_{SPSP}, RTT_{SPV}, RTT_{SPO}) = 2 \cdot max(1, 121, 182) = 2 * 182 = 364ms$. However, if the client in SP, delegates the coordination to the replica in V, the resulting commit latency will be $RTT_{SPV} + 2 \cdot max(RTT_{VV}, RTT_{VI}, RTT_{VC}) = 121 + 2 * 73 = 267ms$ saving around 26.6% of the commit latency obtained when SP is chosen to be the coordinator.

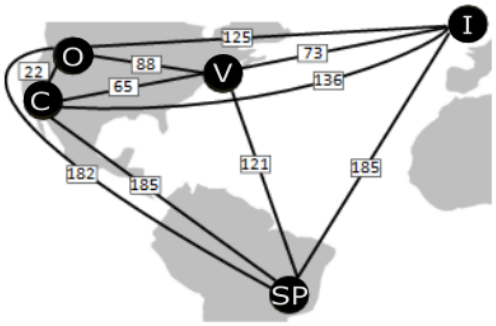


Figure 4: Five replicas of the database are deployed in datacenters I, V, SP, O, and C.

We presented a primitive version of the handoff idea in [34]. To generalize, for any request R from a client at datacenter A, it might be beneficial to handoff this request to a replica at datacenter B if the summation of RTT_{AB} and the time for datacenter B to serve this request L_B are less than L_A, the time to serve this request at datacenter A. In other words, request handoff from datacenter A to datacenter B is beneficial if $L_A > RTT_{AB} + L_B$. This optimization is widely applicable on different protocols and different request types.

4.2 Cover all the optimization aspects

During our SIGMOD demo [34], we ask the participants to place 5 data replicas in 5 out of the 9 datacenters shown in Figure 5. The participants are told that the data is fully replicated in the 5 chosen datacenters and the commitment protocol uses the two rounds of Paxos for isolation and replication and reads are served from the closest quorum. In addition, optimistic read and passive replica reads can be used and there is no restriction on the number of passive replicas that can be used. Finally, the workload at each site is represented by the number of blue clients at each datacenter and handoff can be used when possible.

When passive replica read is enabled and there is no restriction on the number of passive replicas assuming low contention, the commit latency contributes the most to the average transaction latency. As explained in Section 2, the commit latency is expressed as: $L_c = RTT_{c-to-c_c} + 2 \cdot RTT_{c_c-to-p}$.

Most of the demo participants tried to optimize the time to reach the commit coordinator c_c by placing the active replicas at the datacenters that have clients. Although this strategy is intuitive and reduces the time to reach c_c to zero, it does not find the placement the minimizes the overall average commit latency. This happens because the datacenters that have clients happen to be far from each other and the time to reach a quorum of participants is maximized using this strategy.

Figure 5 shows that the placement that minimizes the commit latency must consider all the optimization aspects of the commit latency. It places quorums of replicas close to each other (quorums are shown using the dotted curves) and uses handoff to handoff the commitment to replicas that can quickly form a quorum (handoff is shown using solid arrows). **Surprisingly, in this example, non of the chosen replicas are placed in datacenters that have clients.**

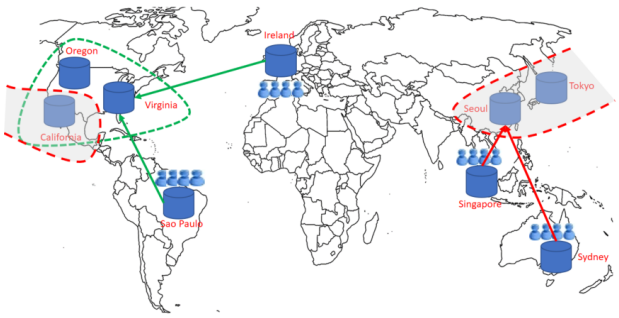


Figure 5: A placement scenario that shows the importance of considering different optimization aspects to minimize the average transaction latency.

5 EVALUATION

A performance evaluation study of the request handoff, the read optimizations, and the proposed heuristics is conducted in this section. In our study, we first evaluate the effect of the read optimizations and the request handoff on execution and commit latencies in Section 5.1. In Section 5.2, we evaluate the performance of the replica-placement heuristics introduced in Section 3.2. We compare the running time and the resulting placement latencies of these heuristics to the running time and the placement latencies of the exhaustive search algorithm in Algorithm 1.

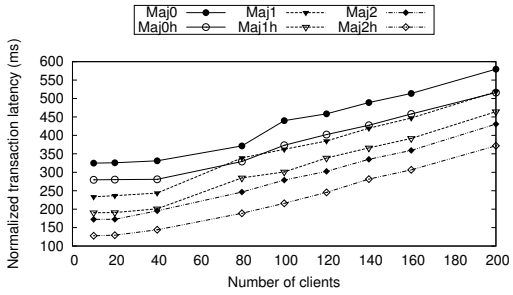
5.1 Placement optimizations

5.1.1 Experimental setup. We use the placement scenario in Figure 4 to evaluate the effect of read optimizations and request handoff on the transaction latency. Request handoff exploits the diversity of the WAN links to decrease the transaction latency and this scenario shows a good example of this diversity, $RTT_{SPC} > 8RTT_{OC}$. Amazon EC2 machines in Ireland (I), Virginia (V), São Paulo (SP), Oregon (O), and California (C) datacenters are leveraged as infrastructure for our experiments. Larger machines are used in datacenters C and O so that we can measure the handoff effect without causing throttling in datacenters C and O. Compute optimized machines are used because computing is the main source of contention in our experiments. We use one compute optimized (c4.large) machine with 2 vCPUs and 3.75 GB of RAM in datacenters V, I, and SP while we use one compute optimized (c3.4xlarge) machine with 16 vCPUs and 30 GB of RAM in datacenters C and O. We assign active replicas to servers in C, O, and V while we assign passive replicas to servers in I and SP. These machines use HBase [3] as the underlying persistent data store. The average RTTs observed between different datacenters are shown in Table 1. The observed RTTs are sampled over 48 hours using AWS nano machines pinging each other. The data is fully replicated in all five datacenters and an optimistic Paxos-based concurrency control protocol is used. A transaction requires two majority rounds to commit and the read-set is validated at commit time. We implemented multiple versions of the protocol based on how read requests are processed in the execution phase. $Maj0$ is conservative and requires read requests to be processed from a majority of the active replicas. $Maj1$ implements the optimistic read optimization and requires read requests to be processed from one active replica. $Maj2$ implements the optimistic read and passive replica optimizations and processes read requests from either active or passive replica. Transaction commitment is implemented the same way in all three versions. The commit handoff optimization is applied on all three versions and it only changes the way a commit coordinator is chosen. For this, we implemented $Maj0h$, $Maj1h$, and $Maj2h$ to apply the handoff optimization on the three protocol implementations. We compare the average obtained commit and transaction latencies for all the three implementations with and without applying the handoff optimization. In addition, we compare transaction throughputs and abort rates for all three implementations.

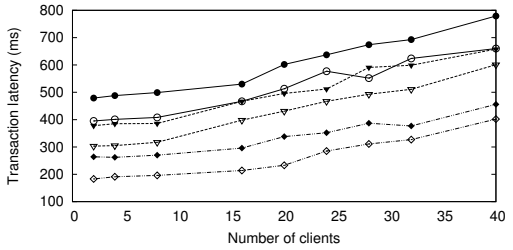
Dedicated client machines in each datacenter generate client workloads. Each client machine is configured with a read coordinator and a commit coordinator. Also, client machines execute a workload thread per client. Clients are uniformly distributed among the 5 datacenters in all the experiments unless otherwise stated. Client machines use YCSB [10] to generate workloads. Since YCSB is not designed to generate multi-record transactions, we use Transactional YCSB (T-YCSB) [12], an extended version of YCSB that generates multi-record transactions, for this purpose. T-YCSB generates transactions that consist of read and write operations on different data records followed by a commit. Each transaction is configured to have five operations. The ratio of read to write operations is 1:1 unless otherwise specified. Read and write operations choose a key from a pool of 50000 keys following a zipfian distribution. This small number of keys enables us to observe the performance of the system under contention. Each client can have only one outgoing transaction. Clients submit a new transaction as soon as they receive a decision for their outgoing transaction. Each experiment runs for 10 minutes.

5.1.2 Experimental results. **Transaction latency**. Active replicas are placed in only three datacenters C, O, and V. Therefore, a majority quorum consists of *two* active replicas. The $Maj0$ implementation assumes that clients at each datacenter drive their transactions(no handoff). Also, it assumes that reads have to be processed from at least two active replicas and commits have to be accepted by and applied to at least two active replicas. $Maj0h$ allows clients in SP to handoff their commit to O and clients in I to handoff their commits to C. $Maj1h$ and $Maj2h$ allow the same handoff plans while enabling optimistic reads in $Maj1h$ and optimistic reads and passive replica reads in $Maj2h$.

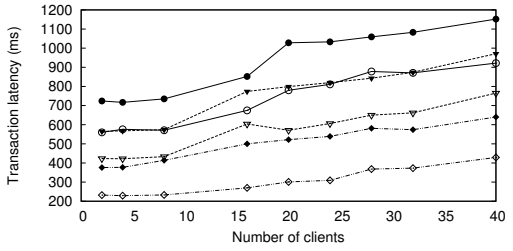
Figure 6 shows the effect of increasing the number of clients from 10 to 200 on transaction latency. As a transaction requires two round trips to a quorum of two active replicas to commit, clients in C and O have their location as an advantage that they can always achieve lower transaction latency and higher throughput than clients in other sites as long as the number of clients is equal in all the datacenters. Therefore, to measure the effect of the placement optimizations on transaction latency in isolation from the throughput, we use the normalized transaction latency as a comparison metric between different implementations. The normalized transaction latency L_{norm} is the average of the average transaction latency in all the datacenters $L_{norm} = \frac{L_C+L_O+L_V+L_I+L_{SP}}{5}$ where L_i is the average transaction latency at datacenter i. As shown in Figure 6a, applying read optimizations in $Maj2$ significantly enhances L_{norm} by 48% compared to $Maj0$. Also, the handoff in $Maj2h$ enhances L_{norma} by 26% compared to $Maj2$ leading to a total enhancement of 60% compared to $Maj0$. Increasing the number of clients beyond 100 (20 at each datacenter) causes throttling in server machines. This throttling leads to an increase in the overall transaction latency and a decrease in the benefit obtained from the applied optimizations. Figures 6b and 6c shows the effect of applying read optimizations and handoff on transaction latencies at I and SP



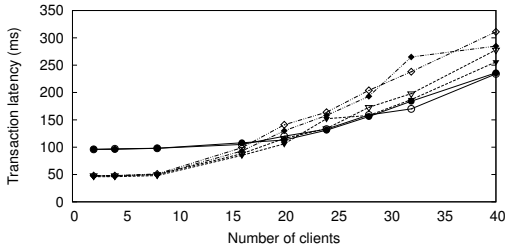
(a) Overall normalized transaction latency.



(b) Transaction latency in Ireland.



(c) Transaction latency in São Paulo.



(d) Transaction latency in California.

Figure 6: Transaction latency as number of clients increases. Figures 6a, 6b, 6c, and 6d share one plotting legend.

respectively. As shown, read optimizations and handoff together in *Maj*2*h* enhances transaction latency compared to *Maj*0 by 62% and 68% in *I* and *SP* respectively. Also, handoff in *Maj*2*h* saves 30% and 38% of the transaction latency compared to *Maj*2 for clients in *I* and *SP*. Figure 6d presents the effect of the placement optimizations on the transaction latency in *C*. As shown, read optimizations significantly reduce the transaction latency in *C* by 49% as reads are served locally. This applies until throttling happens. After throttling, the transaction latency in *C* increases for *Maj*2 because serving reads locally in all the datacenters increases the frequency of the transactions that are ready to commit in the system causing more contention in datacenters *C* and *O*. The handoff slightly increases the transaction latency in *C* and its negative effect is negligible before the throttling happens.

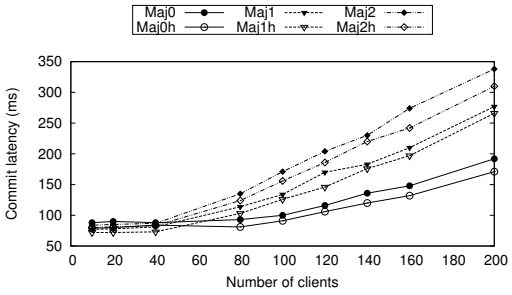
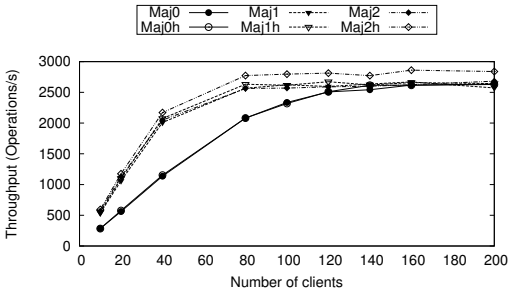
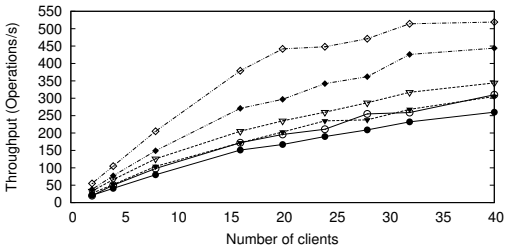


Figure 7: Overall average commit latency as number of clients increases.

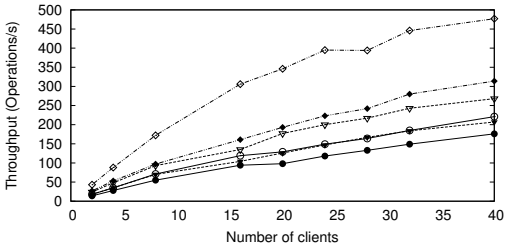
Commit latency. Figure 7 shows the effect of the placement optimizations on the overall average commit latency. While the normalized transaction latency is significantly enhanced by applying read optimizations, read optimizations negatively affect the overall commit latency. By reducing the execution phase latency, the number of active transactions that are ready to commit increases and leads to an increase in the commit latency. However, applying handoff enhances the overall average commit latency by $10 - 15\%$ in *Maj*0*h*, *Maj*1*h*, and *Maj*2*h* compared to *Maj*0, *Maj*1, and *Maj*2 respectively.



(a) Overall throughput.



(b) Throughput in Ireland.



(c) Throughput in São Paulo.

Figure 8: Throughput as number of clients increases. Figures 8a, 8b, and 8c share one plotting legend.

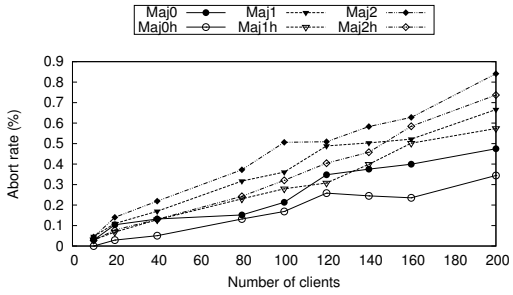


Figure 9: Overall abort rate as number of clients increases.

Throughput. The throughput, measured by number of operations per second, is presented in Figure 8. Figure 8a shows that applying read optimizations in $Maj1$ and $Maj2$ achieves $2x$ the throughput in $Maj0$ until hitting the thrashing point($\geq$ 100 clients). After that, throughput is slightly higher in $Maj1$, $Maj2$, and $Maj1h$ and about 8% higher in $Maj2h$. Throughput results in I and SP are shown in Figures 8b and 8c. These figures show a significant increase of 100% between $Maj0$ and $Maj2h$ in I and 170% between the same implementations in SP. Applying handoff not only significantly benefits I and SP but also benefits the overall throughput.

Abort rate. The abort rates are shown in Figure 9. The abort rate is a result of many factors, such as the amount of contention, the number of concurrent transactions, the lifetime of a transaction, among others. As shown, the overall abort rate is below 1% for all six different implementations. However, we observed two important patterns that are worth analyzing. First, read optimizations increase the abort rate by 100% for some experiment runs. Obtaining the read-set from a local copy increases the chances of reading a stale value and hence increasing transaction aborts. However, these stale values has a small life-time as all the passive replicas are asynchronously updated. Second, handoff decreases the abort rate by $25-30$% because a transaction's lifetime is shortened by reducing the overall transaction latency and specifically the high latency transactions in I and SP.

5.2 Replica-placement heuristics

We evaluate the replica-placement heuristics in this section. This evaluation tries to answer two questions: *How fast can these heuristics find a placement? and how good is this placement compared to the optimal placement?*. For that, we compare the performance and the resulting placements of the proposed heuristics in Algorithms 2 and 3 to the performance and the resulting placements of the exhaustive search in Algorithm 1 at scale. We assume that optimistic reads and passive replica reads are enabled. Therefore, we use commit latency as a comparison metric as the transaction execution latency is negligible when reads and writes are served locally and none of the replicas are overloaded with requests. The proposed heuristics and the exhaustive search programs are all implemented in Java which allows us to conduct a fair comparison. The exhaustive search algorithm evaluates all possible placement combinations and returns the placement that achieves the minimum average commit latency for a certain workload. Algorithm 2 introduces a greedy heuristic that adds one replica at a time achieving the minimum average transaction latency at each iteration. Algorithm 3 is inspired by the K-Means algorithm and it assigns initial weight to each datacenter equals to the number of clients at this datacenter. Weights are updated based on the quorums a datacenter participates at and based on handoff.

Finding the optimal placement of five replicas within ten datacenters can be efficiently done. It requires the evaluation of only 252 different placements and the exhaustive search is sufficient in this case. However, in a more realistic setting, the number of datacenters around the globe, including edge datacenters, may easily exceed 4000 datacenters [1]. Also, it has been shown in [33] that it is economically efficient to deploy storage in datacenters of different cloud providers as non of them provides cheaper storage in all the deployment regions. To choose five datacenters out of 4000 datacenters requires to evaluate 8.5e+15 different placements. To get a sense of the space size and the running time, we evaluated the exhaustive search algorithm and the heuristics proposed in Section 3.2 using different datasets.

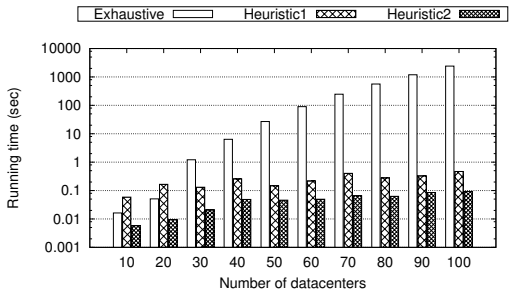
First, we generate multiple datasets of datacenters DC distributed around the globe with randomly chosen round-trip time $0 \leq RTT \leq 500$ ms. We also make sure that the triangle inequality holds among any three datacenters such that $\forall_{A,B,C \in DC}$ $RTT_{AB} + RTT_{BC} \geq RTT_{AC}$. Second, we distribute the workload around the generated datacenters with ratios between $0-10$. We use the generated data as inputs to both the exhaustive search program and the placement heuristics. These experiments are run locally on an Intel Core i5-3210M CPU 2.50GHz with 8GB of RAM.

Running time. In this part of the evaluation, we answer the first question, namely *How fast can these heuristics find a placement?*. Figures 10a and 10b show a running time comparison between the exhaustive search and the placement heuristics when the number of replicas are 5 and 7 respectively. As shown, the running time of the exhaustive search grows exponentially with the number of datacenters while the running time of both heuristics are negligible(< 1 second). Also, the exponential power significantly increases as the number of replicas required to be placed increases. This shows that it is infeasible to use the exhaustive search when the datacenter set size exceeds few tens.

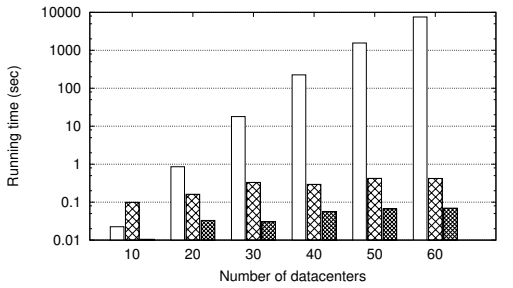
Resulting placements. The second part of the evaluation answers the second question about the quality of the placements found by the proposed heuristics. We compare the commit latency of the resulting placement to the optimal commit latency of the resulting placement decided by the exhaustive search. Figures 11a and 11b show the relative commit latency of the heuristics compared to the optimal commit latency when the number of placed replicas are 5 and 7 respectively. In these figures, the optimal commit latency is represented by 1.0. A relative comparison of the commit latency is shown for both the placement heuristics and the best of the two heuristics as well. As shown, the best of the two heuristics is optimal in 70% of the cases and within $5\% - 11\%$ of the optimal in the rest of the cases. As the running times of both heuristics is negligible, we can always run both the heuristics and choose the best placement out of the two results. Figure 11 suggests that neither heuristics beats the other in all cases.

6 GPLACER EXTENSIONS

In this section, we discuss how GPlacer can be extended to optimize the placement for leader-based protocols. In leader-based protocols, a transaction can fall into one of two categories: single-partition transactions or multi-partition transactions. Single-partition transactions span only one partition and the isolation between transactions that span this partition is managed by the

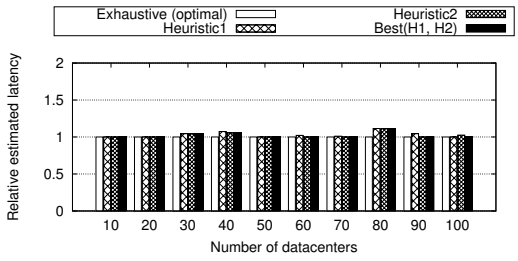


(a) 5 replicas.

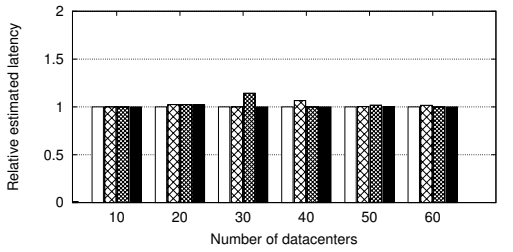


(b) 7 replicas.

Figure 10: The running time (seconds), in log scale, of exhaustive search and placement heuristics as number of datacenters increases. Figures 10a and 10b show the running time when 5 replicas and 7 replicas are chosen respectively. Both figures share one plotting legend.



(a) 5 replicas.



(b) 7 replicas.

Figure 11: A comparison of the resulting commit latency of placements by exhaustive search and placement heuristics as number of datacenters increases. Figures 11a and 11b compare the estimated latency when 5 replicas and 7 replicas are chosen respectively. Both figures share one plotting legend.

leader of this partition. Multi-partition transactions span multiple partitions and typically 2PC is used between the leaders of the partitions involved in a transaction to achieve isolation. In both categories, partition leaders replicate the updates of committed transactions to a majority quorum of their partition replicas using only the second round of Paxos.

The average transaction latency for leader-based protocols is affected by the following factors:

- The distance between the client and the partition leader
- The distance between the partition leader and its replicas
- The distance between different partition leaders involved in multi-partition transactions
- The percentage of multi-partition transactions P_{mp-txn} (how often 2PC is required to be executed)

The first two factors mainly affect single-partition transactions while the last two factors mainly affect multi-partition transaction. Finding the optimal placement for leader-based protocols can easily become impractical. Consider a database with p partitions and we want to place the leaders of these partitions on r replicas. There are r^p different placement combinations and finding the optimal placement by checking all the combinations is impractical. For example, if a database has 500 partitions and we want to place these partitions among 5 replicas. To find the optimal leader placements, 5^{500} different combinations need to be evaluated. Therefore, different heuristics are usually used to limit the search space.

6.1 Leader-placement heuristics

A solution that *considers all the optimization aspects* should adapt the placement based on the percentage of the multi-partition transactions P_{mp-txn}. Sharov et al. [28] place the leaders of all the partitions in one datacenter. Algorithm 4 implements the leader-placement heuristic introduced in [28]. It iterates over all the replicas, line 4, and evaluates the latency assuming that all the partition leaders are placed in the currently evaluated replica. The replica that achieves the minimum latency is returned. This heuristic optimizes the placement when P_{mp-txn} is high. However, when P_{mp-txn} is low, placing the leaders of all the partitions in one datacenter can hurt the performance in addition to introducing a single point of failure.

The second heuristic is to *independently* place the leaders of different partitions. For every partition, place its leader at the same datacenter where it is accessed the most. This heuristic optimizes the placement when P_{mp-txn} is low and partitions are mostly accessed from one datacenter.

Algorithm 4 Finds datacenter $dc_l \in DC$ that achieves the minimum average transaction latency assuming all the partition leaders are put together in one datacenter.

Input: DC, RTT_{ij} $\forall i, j \in DC$, and c_i $\forall i \in DC$
Output: dc_l

1: $dc_l \leftarrow \emptyset, l \leftarrow MaxInt$
2: **for all** $j \in DC$ **do**
3: $tempL \leftarrow 0$
4: **for all** $i \in DC$ **do**
5: $tempL += c_i \cdot (RTT_{ij} + q_j)$ // q_j is the time for replica j to reach its closest quorum from DC.
6: **end for**
7: **if** $tempL < l$ **then**
8: $l \leftarrow tempL, dc_l \leftarrow j$
9: **end if**
10: **end for**

Placing all the partition leaders in one datacenter favors multi-partition transactions while independently placing them in multiple datacenters favors single-partition transactions. When the workload is a mixture of both transaction categories, both heuristics fail to optimize the placement. Therefore, we present a third heuristic that optimizes the placement when the workload is divided between the two categories. This heuristic uses GPlacer to find the set of $2f + 1$ datacenters that should host a replica DC_{db} according the workload distribution. Then it runs Algorithm 5 to independently place the leaders of each partition among the chosen replicas. The second heuristic independently places partition leaders in the universe of all datacenters DC while the third heuristic limits the placement to the set of datacenters that are chosen by GPlacer DC_{db}. In Section 6.2, we compare the resulting placements of the three heuristics.

Algorithm 5 Places the leader of each partition among the chosen replicas and closer to the clients who access this partition the most.

Input: DC_{db}, and p_i $\forall i \in DC$ and $\forall p \in P$ // P is the set of all partitions and p_i is the percentage of access for partition p from datacenter i.

Output: $\forall_{p \in P}\ l_p$

1: **for all** $p \in P$ **do**
2: $\quad i \leftarrow max(\forall_{j \in DC}\ p_j)$
3: $\quad l_p \leftarrow nearest(dc \in DC_{db}, i)$ // returns the nearest datacenter $dc \in DC_{db}$ to datacenter i.
4: **end for**

6.2 Leader-placement heuristics evaluation

We compare the expected average commit latency of the resulting leader placements using the three heuristics. The percentage of distributed multi-partition transactions is varied and the expected commit latency is calculated for the three heuristics. Before placing partition leaders in the third heuristic, we use the exhaustive search algorithm to find DC_{db}. Then, we use the heuristic to place partition leaders among the chosen replicas.

The commit latency of a single partition transaction is estimated as the RTT from the client datacenter to the partition leader datacenter plus the RTT from the partition leader datacenter to a majority of the partition replicas. The commit latency of a multi-partition transaction requires an addition 2PC between the involved partitions. In our evaluation, we assume that the 2PC added latency is negligible if all the involved partition leaders are placed together and two round-trips to all the partition leaders if they are not placed together. This assumption favors algorithm 4 over our proposed heuristic in algorithm 5.

Figure 12 shows the expected commit latency when the three heuristics are used to place partition leaders. The expected commit latency is shown in a log scale in the y-axis and the percentage of the multi-partition transaction is shown in the x-axis. In this scenario, clients are distributed among 10 datacenters and 5 datacenters host replicas for heuristic 3. A transaction has to be replicated to 3 replicas before it is committed. As heuristic 1 places all partition leaders in one datacenters, the average commit latency does not change with the percentage of multi-partition transactions. Therefore, the average estimated commit latency is a horizontal line. However, for heuristics 2 and 3, increasing the percentage of multi-partition transactions boosts the average commit latency as the cost of the 2PC between all partition leaders increases. Figure 12 suggests that heuristic 2 should be used to place partition leaders as long as P_{mp-txn} is low (below 9%). Heuristic 3 should be used when $9\% < P_{mp-txn} < 25\%$ and heuristic 1 should be used if P_{mp-txn} is high (above 25%). Typically, the percentage of multi-partition transactions is < 10% [31, 32].

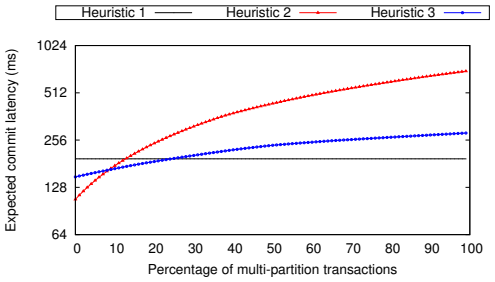


Figure 12: A commit latency comparison between leader-placement heuristics as the percentage of multi-partition transaction increases.

It is important to mention that the commit latency crossing lines between the three heuristics are different for different scenarios and estimates should be calculated a priori to decide which leader placement achieves the minimum commit latency for a given scenario. Our framework evaluates the outcomes of the three heuristics and choses the placement that achieves the minimum latency.

7 CONCLUSION

In this paper, we address the data placement problem of geo-replicated databases with strong consistency guarantees. We present different placement optimizations to reduce transactions execution latency and commit latency. These placement optimizations are widely applied on different distributed transaction management protocols. Our evaluation shows that applying the read optimizations and the request handoff optimization could reduce transaction latency by 68% and increases throughput by 170%. To address the placement problem at scale, we propose different placement heuristics that can efficiently find sub-optimal placements within 5–10% of the optimal placements. Experiments show that these heuristics are able to scale without significantly reducing the quality of the resulting placements from the optimal placement. Finally, we discuss three partition leader placement heuristics to place partition leaders. Experiments show that non of the three heuristics is superior when the percentage of multi-partition transactions changes. Unlike in [28] which uses one heuristic to place partition leaders regardless of the percentage of multi-partition transactions, our framework switches between different heuristics when the percentage of multi-partition transactions changes.

8 ACKNOWLEDGEMENT

This work is partially funded by the NSF grant CNS-1703560.

REFERENCES

[1] 2017. Datacenter Map. http://www.datacentermap.com/datacenters.html/. (2017).
[2] 2017. GLPK: GNU Linear Programming Kit. https://www.gnu.org/software/glpk/. (2017).
[3] 2017. HBase. https://hbase.apache.org/. (2017).
[4] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: Automated Data Placement for Geo-Distributed Cloud Services.. In *NSDI*, Vol. 10. 28–0.

[5] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alex Lloyd, Sergey Melnik, Rajesh Rao, Dave Shue, Chris Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proc. SIGMOD 2017*. 331–343.

[6] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services.. In *CIDR*, Vol. 11. 223–234.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, and others. 2013. Tao: FacebookâĂŹs distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[9] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. 2013. Spanner: GoogleâĂŹs globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[12] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.

[13] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 133–160.

[14] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. 9.

[15] Bettina Kemme and Gustavo Alonso. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication.. In *VLDB*. Citeseer, 134–143.

[16] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 113–126.

[17] Avinash Lakshman and Prashant Malik. 2009. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 5–5.

[18] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1659–1674.

[19] Guoxin Liu and Haiying Shen. 2016. Minimum-cost Cloud Storage Service Across Multiple Cloud Providers. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE, 129–138.

[20] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions.. In *OSDI*. 135–150.

[21] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment* 6, 9 (2013), 661–672.

[22] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2013. Message Futures: Fast Commitment of Transactions in Multi-datacenter Environments.. In *CIDR*.

[23] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1279–1294.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and others. 2013. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.

[25] Stacy Patterson and others. 2012. Serializability, not Serial: Concurrency Control and Availability in Multi-Datacenter Datastores. *PVLDB* (2012).

[26] Fan Ping, Jeong-Hyon Hwang, XiaoHu Li, Chris McConnell, and Rohini Vabbalareddy. 2011. Wide area placement of data replicas for fast and highly available data access. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*. ACM, 1–8.

[27] Moinuddin K Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. Phase change memory: From devices to systems. *Synthesis Lectures on Computer Architecture* 6, 4 (2011), 1–134.

[28] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take me to your leader!: online optimization of distributed storage configurations. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1490–1501.

[29] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle LittleïnĄeld, and Phoenix Tong. 2012. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *SIGMOD*. Talk given at SIGMOD 2012.

[30] Michael Stonebraker. 2010. Why Enterprises Are Uninterested in NoSQL. http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext/. (2010).

[31] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[32] Alexander Thomson and others. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*.

[33] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. 2013. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 292–308.

[34] Victor Zakhary, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2016. Db-risk: The game of global database placement. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2185–2188.