Data Placement in a Scalable Transactional Data Store

Björn Patrick Swift
bst360@few.vu.nl
Student number: 1999672

Master's thesis
Parallel and Distributed Computer Systems
Vrije Universiteit, Amsterdam

Supervisor: Guillaume Pierre

February 19, 2012

Abstract

Distributed data stores have recently become increasingly popular. This can be largely attributed to their impressive ability to scale. However, to achieve this scalability, they sacrifice properties such as atomicity and consistency, important properties when constructing simple control flows and asserting correctness.

CloudTPS is a transactional layer on top of distributed data stores, bringing back transactional consistency. Its request latency is related to how many servers are involved in servicing each transaction, and in this thesis we explore methods to reduce this number. First, we introduce a data placement mechanism to CloudTPS, providing a real environment to test and benchmark placement policies. Second, we implement a distributed approach to minimum k-cut partitioning including load-balancing and a tunable migration cost-model, both of which can bring important benefits to CloudTPS's performance.

Contents

Chapter 1

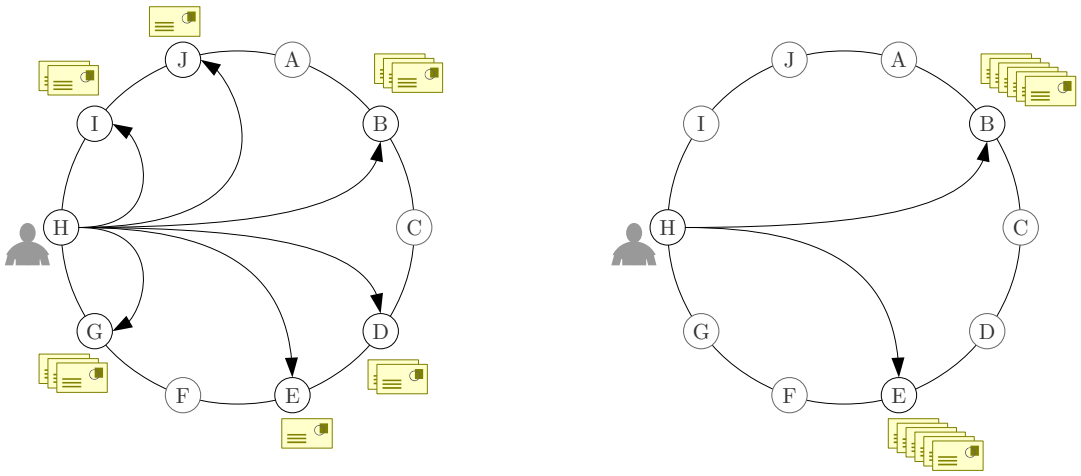
Introduction

Data intensive applications commonly rely on Relational Database Management Systems (RDBMS) to store their structured data. Relational databases offer important guarantees such as atomicity, consistency, isolation and durability, which are often referred to as the ACID properties. These properties have become an important foundation for developers when constructing simple control flows and asserting correctness.

As applications grow in size, storage systems can become bottlenecks and RDBMSs have proven difficult to scale. To address their limited scalability, a number of distributed data stores have surfaced, both for structured and unstructured data [4, 6, 10]. These systems have been designed from the get-go with high-scalability and high-availability as their foremost requirements, while features that are difficult to implement in distributed systems, such as atomicity and consistency, have come second and have been sacrificed. These design decisions have allowed these data stores, sometimes referred to as NoSQL, to reach impressive scale.

The relaxed consistency model of distributed data stores traces back to the CAP theorem which states that a distributed shared-data system cannot guarantee data consistency, system availability and tolerance to network partition—only two can be achieved at any given time [3]. Some argue that network partitions in large distributed systems are a given, so there are only two choices: consistency or availability. Favoring availability requires a relaxed consistency model, whereas favoring consistency implies that under certain conditions the system may not be available [17].

However, the importance of transactional consistency should not be underestimated. Besides situations where users prefer consistency to reduce complexity, there are several applications which simply cannot make use of data stores that are weakly-consistent. Financial applications are a typical example. Working with weakly-consistent data stores can be complex, to the point where Jeff Dean stated that, in retrospect, not supporting multi-row consistent transactions in BigTable was a mistake. Application developers within Google wanted transactions semantics and ended up implementing their own protocols, however, often incorrectly [9]. Google has been working on a new distributed data store which will have both a strong and weak consistency model, and support for distributed transactions [8]. We argue that a scalable transactional data



(a) Random distribution **(b)** Intelligent placement

Figure 1.1: By careful item placement, transaction span can be reduced.

store is an important foundation for current and future applications.

CloudTPS bridges the gap between RDBMSs and distributed data stores by combining the ACID properties of relational databases with the scalability characteristics of distributed data stores [18]. CloudTPS is a transactional layer on top of distributed data stores, serializing item reads and writes in a distributed manner, where each storage node acts as a transaction manager for locally stored items. As items are randomly placed on CloudTPS's Chord-like ring, this approach scales well in terms of number of items and nodes. It is, however, limited by the number of nodes involved in each transaction, the *transaction span*. Another system, called Schism, showed that the more nodes are involved in any given transaction, the longer it will take [7]. Schism further shows that by clustering related data items, transaction span can be reduced and, as a result, latency reduced and throughput increased.

Schism's approach is interesting. However, it has two limitations. Firstly, it uses a centralized approach when deriving its item placement, which will not scale well. Secondly, its approach harvests traces from a completed run, brings down the system, rearranges items, brings the system back up and reruns the same benchmark on the new placement. This may not always be feasible.

The focus of this project is to address these limitations in the context of CloudTPS. First, we need a method to estimate which items are being accessed together in transactions. This estimation needs to be scalable, so aggregating logs for central analysis in not an option. Second, we need to interpret the estimations and use them to devise a new item placement. Third, we have to migrate items from one node to another, while the system is running and accepting requests, without affecting system availability. Finally, as item location can no longer be calculated via hashing, the system has to keep track of item location and correctly route requests to

their current location. All this item-location communication introduces overhead. However, if done sensibly, we stand to gain performance as fewer nodes will be involved in each transaction.

Our solution is presented in two parts, mechanism and policy. The mechanism includes a distributed transaction history for workload analysis, a distributed lookup table for item placement and request routing, and finally, support for live item migration within the CloudTPS transactional framework. Our policy relies on a distributed minimum k-cut partitioning, performing local optimizations through gossiping, gradually moving the system closer to an optimal state. The policy also includes support for load-balancing the system and features configurable cost-model for item migrations.

The placement mechanism contributes towards a real-world platform for researchers to develop and benchmark placement policies on a transactionally consistent data store. The contribution of our distributed placement policy is twofold. Firstly, it presents an implementation that is likely to scale well, making it a practical option for large deployments. Second, it introduces a migration cost-model, where an operator can tune the cost of migration and thus the incentive to migrate items.

This thesis is structured as follows. Chapter 2 presents background and related work. Chapter 3 covers the mechanism implemented in CloudTPS to support item placement and live migration. Chapter 4 focuses on our distributed placement policy and Chapter 5 evaluates the performance of our modified CloudTPS compared to the performance of Vanilla CloudTPS. Finally, we present our conclusions in Chapter 6 and discuss future work.

Chapter 2

Background and related work

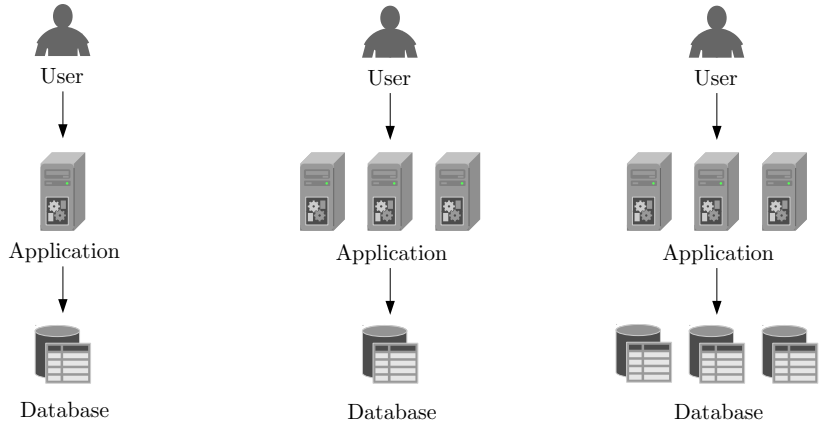
Before discussing our solution we will view prior work on three topics: scaling relational databases, distributed data stores and data placement policies.

2.1 Scalable Relational Databases

Relational Database Management Systems (RDBMSs) are arguably the most widely used storage engines for structured data. They are based on the relational model as introduced by Codd in 1970 [5] and implement a well-defined feature set. Most RDBMSs guarantee the four ACID properties; atomicity, consistency, isolation and durability. These provide an environment that many programmers have become comfortable with and is widely accepted.

As the applications grow in size, storage systems become bottlenecks and RDBMSs have proven difficult to scale. Whereas the application layer is often stateless and easily scalable, a replicated database layer is a shared-state system in which all servers must see all writes (see Figure 2.1). Each database server processes $\frac{1}{n} * Read\_Queries + Write\_Queries$, where n is the number of replicas. Even with infinite replica count, all servers process all write-queries, which may overload each replica. As a result, more advanced scaling techniques are required.

The two most common approaches to scaling the database layer are vertical and horizontal partitioning. With vertical partitioning, the data set is partitioned logically, whereas in horizontal partitioning logical partitions are spread amongst multiple servers, typically by range or random placement. Figure 2.2 illustrates how a bank might vertically partition users, savings accounts and checking accounts, and horizontally partition savings and checking accounts. Even though partitioning has proven to scale well [15], it presents two challenges. Firstly, it requires changes to application logic, as the application must handle routing requests and merging data. Secondly, by partitioning the dataset onto multiple independent RDMBs, atomicity, consistency and isolation have all been sacrificed.



(a) No scaling **(b)** Application scaling **(c)** Database scaling

Figure 2.1: Whereas the application layer typically scales well, the same does not hold for the shared-state RDMBs layer.

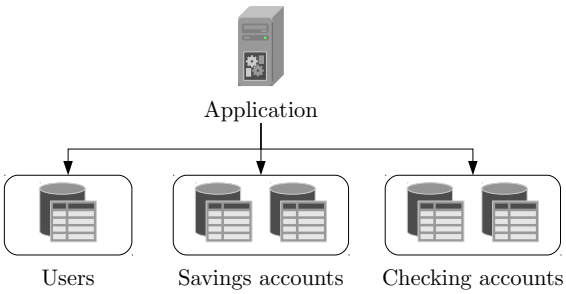


Figure 2.2: Whilst partitioning the dataset achieves greater scalability, this sacrifices the atomicity, consistency and isolation guarantees of RDBMSs (ACI in ACID).

2.2 Distributed Data Stores

Distributed data stores have been designed from the get-go with high-scalability and high-availability as their foremost requirements, while features that are difficult to implement in distributed systems, such as atomicity and consistency, came second and were sacrificed. Google and Amazon described their proprietary data stores, Bigtable [4] and Dynamo [10], in papers which were influential if only for demonstrating that non-relational databases—frequently coined NoSQL—can be used as building blocks for highly-available applications.

Whilst the two systems are frequently cited at one and the same time, the two systems are quite different and built with different requirements. Bigtable is a sparse, distributed, multi-dimensional sorted map, built to scale to petabytes of data. Reads and writes of single data items are atomic and consistent; however, multi-row atomicity is not supported. Dynamo's prime design goal was high-availability, where writes could be accepted, even in the event of a network partition. To achieve this, Dynamo employes a weak-consistency model where multiple nodes can

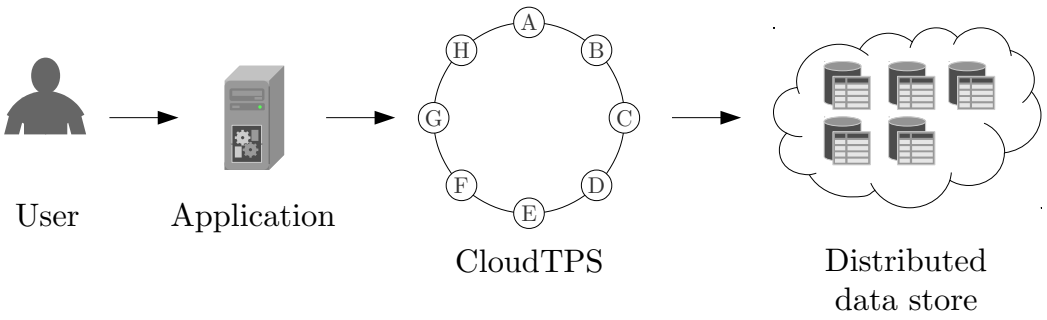


Figure 2.3: CloudTPS is a transactional layer between the application and the data store (adapted from [18]).

accept writes which later get reconciled, either automatically or with the help of the application, depending on causal relations. The data model is simple, Dynamo is a key-value store. Yahoo's PNUTS [6] is another example, focusing on geographical scalability. PNUTS offers a configurable per-record consistency model and read-modify-write-register semantics, allowing applications to implement transaction-like semantics on a per-record basis.

These papers sparked several open source data stores. HBase and Riak are examples of projects based on Bigtable and Dynamo, respectively, and Cassandra aims at combining Dynamo's eventual consistency model and Bigtable's column family data model. More recently, a surge of smaller distributed data stores have come to be, such as Redis, MongoDB and CouchDB, each with their own different feature sets. One thing they have in common, however, is their lack of transactional consistency.

2.2.1 CloudTPS

CloudTPS is a scalable transaction manager, bringing transactional consistency back to the table. CloudTPS nodes implement transaction managers (TM) which are randomly placed on a Chord-like ring [16]. Items are placed on the same ring, based on their item key-hash, and belong to their successive TM. Item access is routed through the item's TM, making CloudTPS system a per-item single-master system, guaranteeing consistent items reads, multi-item atomicity and isolation.

Membership is global and shared by all TMs. As a result, routing is a local computation: find the item key-hash and dispatch request to the successive TM. Because multi-item transactions run across all the data-items' TMs, there is a strong incentive to cluster items that are frequently accessed together. This would lower the number of TMs involved, and therefore lower the transaction's service time.

These properties make CloudTPS a convenient platform on to which our project can be built.

2.3 Data Placement Policies

Data placement policies have been studied in various settings for over 20 years [13], and now with the recent popularity of distributed data stores, the topic has received renewed interest. Recent work can be split largely into two categories. First, geographical placement policies, such as Volley, move content between geographically sparse data-centers, minimizing latency between data host and consumer [2]. Second, data partitioning policies, such as Schism, move items between nodes, minimizing the total number of nodes involved in carrying out transactions [7]. In our work we explore the latter category, in the context of CloudTPS.

Schism is a graph based placement policy which makes its decisions based on workload traces. It constructs a graph based on transactions in the trace and employs graph partitioning algorithms to find optimal placement. This approach has been shown to be effective and is the inspiration for our partitioning strategy. Schism's policy is interesting. However, the system as a whole has several practical limitations, for example, live item migration and live trace analysis. In addition, Schism is inherently a centralized approach, and does not scale well with the number of items.

Chapter 3

Data Item Relocation Mechanism

Our solution is presented in two parts, mechanism and policy. This chapter focuses on mechanism and discusses the components that build on top of CloudTPS.

3.1 Request Routing

When a distributed data store receives a read request for a particular item x, it first needs to route this request to the node responsible for storing x. CloudTPS uses hash placement, where the location of x is determined by $hash(x)$. This method of determining x's location is consistent amongst all nodes in the system and can be calculated locally.

When we start moving items around, we need some other mechanism to determine the current location of item x. Instead of the hashing function $hash(x)$ we need some sort of lookup function $lookup(x)$. Whatever its implementation, all nodes must adhere to this same protocol.

3.2 Distributed Lookup Table

We introduce a *lookup table* that maps unique data item identifiers to physical nodes, to implement the lookup function $lookup(x)$. The granularity is currently one-to-one, so each data item in CloudTPS will have a pointer in the lookup table. The CloudTPS operator can choose, on a per-table basis, whether tables should use hash or lookup placement.

In the following subsections we will describe the components that together implement the lookup table.

3.2.1 Knowledgeable Nodes

When using hashing, all nodes can locally compute where item x is stored. These local routing decisions, however, are not possible when items are placed using an arbitrary placement policy. Instead, we place pointers to items at fixed nodes, and call these nodes *knowledgeable*. The pointers are placed using hashing, and so each node can locate item pointers just as they could previously locate the items themselves.

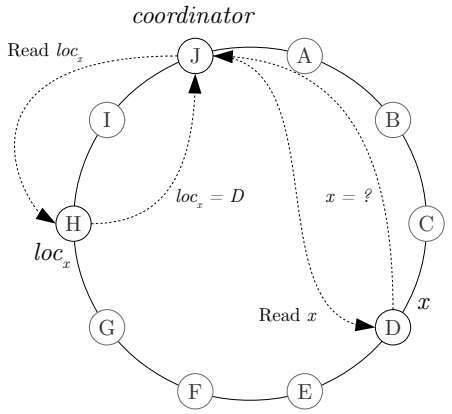


Figure 3.1: Routing a read request for item x.

We denote the location pointer for item x as loc_x and define the node knowledgeable of x's location as the node that holds the loc_x pointer:

$$knowledgeable(x) = hash(loc_x)$$

Figure 3.1 illustrates how node J would route a read request for item x.

1. Node J does not know where item x is located. Therefore, it needs to ask the node that is knowledgeable of x's location. We find that $knowledgeable(x) = hash(loc_x) = H$ and dispatch a read request to H for loc_x.
2. Node H returns the value of $loc_x = D$.
3. Node J dispatches a read request to D.
4. Node D returns the value of item x.

Consistency vs. Performance

There are two ways to perform these lookups. One is using CloudTPS's transactional framework, where J would coordinate a distributed transaction involving H and D. This would provide strong consistency and guarantee that the location pointer loc_x would be consistent with the actual location of x. However, this would be prohibitively costly, especially in the context of CloudTPS which currently does several optimizations for read-only requests, which would no longer be feasible.

The other approach is to perform these lookups outside of CloudTPS's transactional framework and allow for the occasional inconsistent read. This both avoids the transactional overhead and allows us to do optimizations, such as to cache location pointers, as we will discuss in Section

3.2.3. We have chosen this latter approach and perform lookups in a separate component, the placement manager.

3.2.2 Local Knowledge

Since location lookups are performed outside of transactions, we may occasionally route requests incorrectly. For example, looking back at Figure 3.1, we see that item x could have been migrated away from node D between the time that J received x's location and the time that J dispatched the request to D. In this case, the request needs to be rejected and sent back to the coordinator, or rerouted (further discussed in Section 3.2.4).

We find that two pieces of information need to be consistent: the actual location of item x and the value of the loc_x pointer on the knowledgeable node. Therefore, the node that currently stores x and its knowledgeable node have to be in agreement on x's location.

Whereas before each node was only responsible for item pointers, we now add that each node is *also* responsible for knowing which items are stored locally. Together, we call this *local knowledge* (LK), and LK is always consistent. In Figure 3.1, both H and D will have a strongly consistent local knowledge of the location of item x. The methods used to keep the LK consistent are discussed in Sections 3.2.6 and 3.3.

3.2.3 Local Cache

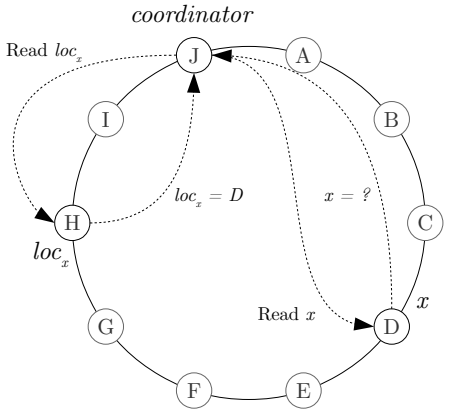
The lookup table previously described has the ability to route requests based on location pointers and the ability to handle incorrectly routed requests. However, the cost of the location-pointer lookup is so costly that it can outweigh even the benefit of optimal item placement. To tackle this, we add a *local cache* to each node which caches recently read pointers.

Figure 3.2 compares the required requests for caches misses (3.2a) and cache hits (3.2b). Once the cache has been populated, the $lookup(x)$ function has the same complexity as $hash(x)$. The cache itself is limited in size and entries are evicted using a Least Recently Used (LRU) policy.

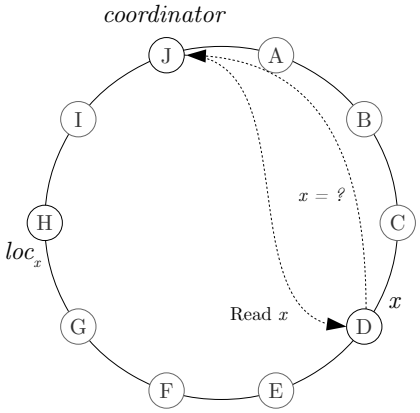
3.2.4 Inconsistent Values

With each node caching item location, the chance of location pointer inconsistencies and false request routes increases substantially. Figure 3.3 illustrates how node J would incorrectly route a request for item x to D, based on an outdated, locally cached lookup pointer. In this case, node D can do one of two things. First, in the case of read-write transactions, node D will send an abort message to the coordinating node J, and piggyback an invalidation request for the outdated location pointer. Node J will then clear the offending value from its local cache and may restart the transaction[1]. Second, in the case of read-only transaction, as a performance

[1]Due to a performance optimization in CloudTPS we cannot restart transactions directly, as CloudTPS makes the assumption that a restarted transaction will be carried out on the same set of nodes. Therefore, we need to abort the transaction and start from scratch.

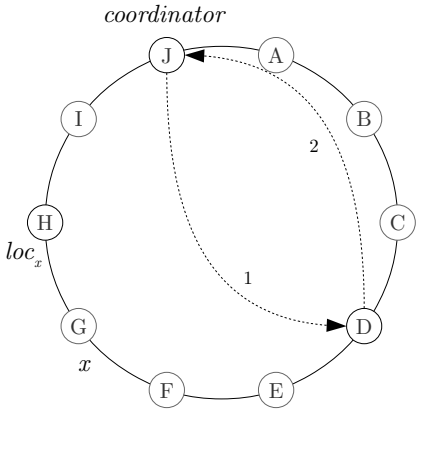


(a) No cache, or local cache miss.

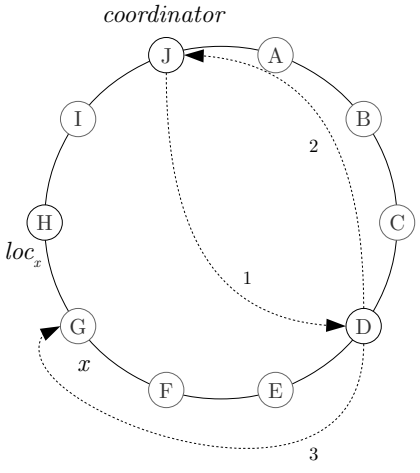


(b) Local cache hit.

Figure 3.2: Request routing with and without a cocal cache.



(a) Request aborted.



(b) Request forwarded.

Figure 3.3: Request routes, based on outdated location pointers.

optimization, node D can forward the request to the node currently storing item x. This is possible since read-only transactions do not follow the same two-phase commit protocol as read-write transactions.

This latter performance optimization is inspired by transposition table driven scheduling, where tasks are passed on to other workers without explicitly asking permission [14]. Although our setup is somewhat different, this does allow for several performance optimizations. For example, in Figure 3.3b the invalid route only causes one extra message to be sent, namely from node D to node G. This example makes two assumptions. First, node D is aware of x's current location, which can be stored in D's local cache upon migration. Second, we do not count the purge request from D to J as this is an asynchronous message that does not affect the request's service time.

Caching dramatically reduces the number of location pointer lookups, at the cost of increasing the inconsistency window, and therefore increasing the number of incorrect routes. Usually, the cost of incorrect routes is far outweighed by the benefit of caching pointers, and therefore we find this to be a sensible trade-off to make.

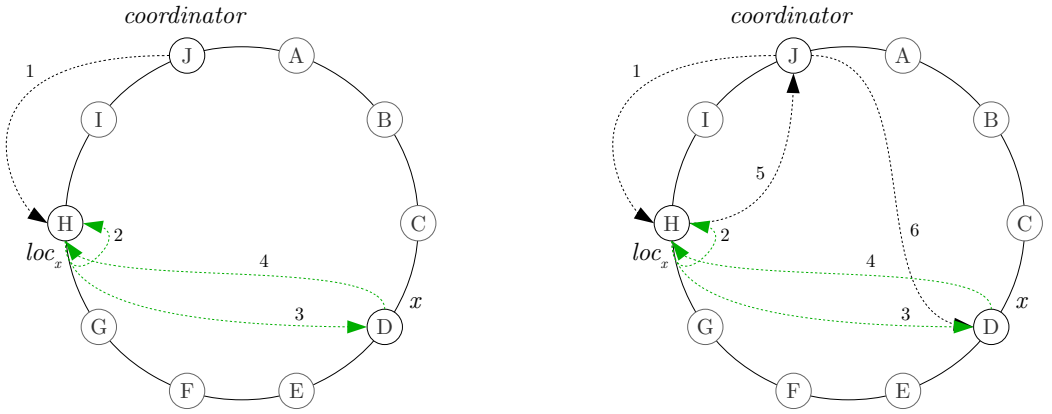
3.2.5 Underlying Storage

The lookup table can grow quite large, holding up to one location pointer for each item in the data store. We make use of CloudTPS itself for storing this metadata, as any other table. We choose the knowledgeable nodes to be the same nodes as the ones CloudTPS chooses to store the location pointer data items themselves. This way, CloudTPS keeps the underlying data table consistent, and our placement manager keeps the local knowledge consistent, on both the knowledgeable node as well as the node currently holding the item.

3.2.6 Lazy Population

The local knowledge is populated lazily, both to limit startup time and memory usage. This is illustrated by Figure 3.4.

1. Node J dispatches a message to node H, requesting to read loc_x, item x's location.
2. Node H does not have the loc_x in its local knowledge, so it asks its local placement manager to read the lookup table entry from the underlying CloudTPS table.
3. The placement manager on H determines that item x should be stored on node D. A message is dispatched to D, requesting that its local knowledge is updated accordingly.
4. Node D acknowledges the local knowledge population requests.
5. At this point, the local knowledge on x's knowledgeable node H and x's home node D is consistent, and we can send loc_x to J
6. Node J will cache $loc_x = D$ and dispatch the read request to D.



(a) First request triggers population.

(b) Request carries on after population.

Figure 3.4: The lookup table is populated lazily on first item access.

Initial Placement

The previous example assumes that loc_x, the location of item x, exists in the underlying CloudTPS datastore. For this to be a general assumption, the system would have to iterate through all items and create location pointers, if we wanted to enable lookup placement for a given table. This can be costly, and so we choose not only to populate the local knowledge lazily, but also to lazily populate the underlying CloudTPS table.

This introduces a special case within step 2 in Figure 3.4a. The local placement manager *attempts* to read loc_x from the underlying CloudTPS datastore. If the value exists, it is returned as previously demonstrated. If not, the placement manager will create an initial placement and write loc_x to persistent storage before continuing. The initial placement is currently chosen to be local to loc_x, that is $loc_x = hash(loc_x)$. This is essentially random placement, as before; however, it is based on a different key (prefixed with `CloudTPS__placement_`). The benefit is that we reduce the cost of steps 3 and 4 in the previous example, as these operations are now local.

Other initial placement strategies could be explored, such as placing a new item in proximity to other items in the write transaction. For example, if a user adds an item to a shopping cart, it may be sensible to store this item on the same node as the shopping cart itself.

3.2.7 Yielding Transactions

Local cache misses may involve remote lookups and cause the transaction to block. However, CloudTPS is based upon message-passing between single-threaded components, and so blocking is not an option. To accommodate for this we yield transactions upon a cache miss and place them in a blocked-transaction queue. The placement manager requests the location pointer and

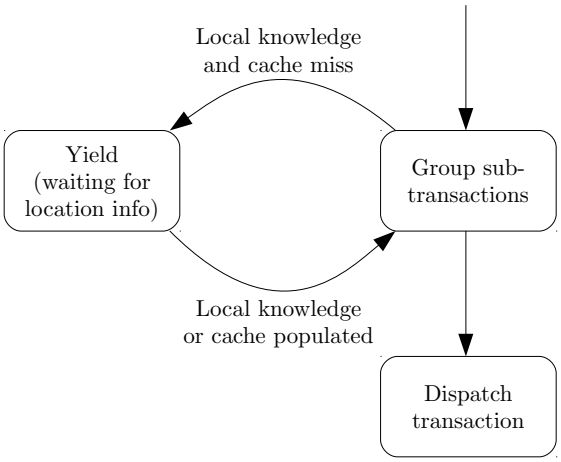


Figure 3.5: Transactions are yielded while location information is resolved.

once retrieved, notifies the local transaction manager, requesting that the blocked transaction be resumed (see Figure 3.5).

3.2.8 Initial Cache Miss

When a read misses the cache we spawn a worker to read the remote value and populate the cache. After that, subsequent reads will hit the cache, until the entry is evicted. However, there is a corner case, where a second transaction attempts to read the same value before the worker manages to populate the local cache. This requires special handling, because otherwise multiple workers could be spawned to resolve the same pointer.

The placement manager guards itself from this situation by creating a *pending lookup* value and keeping a queue of transactions pending the returned value. For example, transaction T_1 might request loc_x from its local placement manager, which would create a pending entry for loc_x and add T_1 to the notification queue. If transaction T_2 requests loc_x before the value has been resolved, T_2 is added to the notification queue and the transaction yielded. Once the placement manager populates loc_x, it will signal T_1 and T_2, which resume processing.

This prevention method is employed on two distinct layers in CloudTPS. First, the placement manager (PM) will hold a queue for local transactions blocked on a particular lookup key. Secondly, the PM will hold a separate queue with remote PMs which are reading location information. Using the example in Figure 3.4, we can both have several local transactions on J pending loc_x, and also have several nodes waiting on H to perform the lazy population or initial placement. In the first case we keep account of local transactions and in the latter case we keep account of remote nodes.

3.3 Item Migration

So far we have discussed how item location is stored and how requests are routed. The missing piece of the puzzle is how items are migrated from one node the other. The requirement is for CloudTPS to migrate items, one-by-one or several at a time, without affecting the availability of the system.

An item migration involves several nodes. In addition to the source and destination nodes, the item's knowledgeable node also has to update its location pointer. An item migration needs to be atomic, consistent, isolated and durable. Coincidentally, these are precisely the guarantees that CloudTPS provides. Thanks to its extensible transaction framework, we were able to implement live migrations by adding a new transaction type, a *migration transaction* with only minimal modifications to the framework itself.

There are three steps in migrating an item. First, the item itself must be copied from source to destination. As CloudTPS may be running on top of a weakly consistent data store, this needs to happen within CloudTPS itself, persisting the value at the source and reading from persistent storage at the destination is not guaranteed to be correct. Second, the location pointer on the knowledgeable node needs to be updated. Third and finally, the local knowledge on all three nodes must be updated to reflect the change.

3.3.1 An Example of a Migration

Figure 3.6 illustrates a transaction where data item x is migrated from node D to node E. In addition to nodes D and E, the transaction will involve a Transaction Coordinator (TC) and x's knowledgeable node. Note that while the figure separates these four roles, they will typically not be carried out on more than three separate nodes.

1. The Transaction Coordinator (TC) initiates a data item migration. Two sub-transactions are created; an UpdateTransaction which gets dispatched to H, and a MigrationTransaction which gets dispatched to D.

2. H receives the UpdateTransaction, which is handled like any other transaction, and H sends back a *ready* vote.

3. D receives the MigrationTransaction, which contains information on which item should be migrated and where. D spawns a derived MigrationTransaction containing the current value of x and sends this transaction back to TC, along with its *ready* vote.[2]

4. TC receives D's derived sub-transaction, expands the parent transaction and dispatches the new sub-transaction to E.

[2]This usage of derived transactions is analogous to how CloudTPS implements updates to secondary indices.

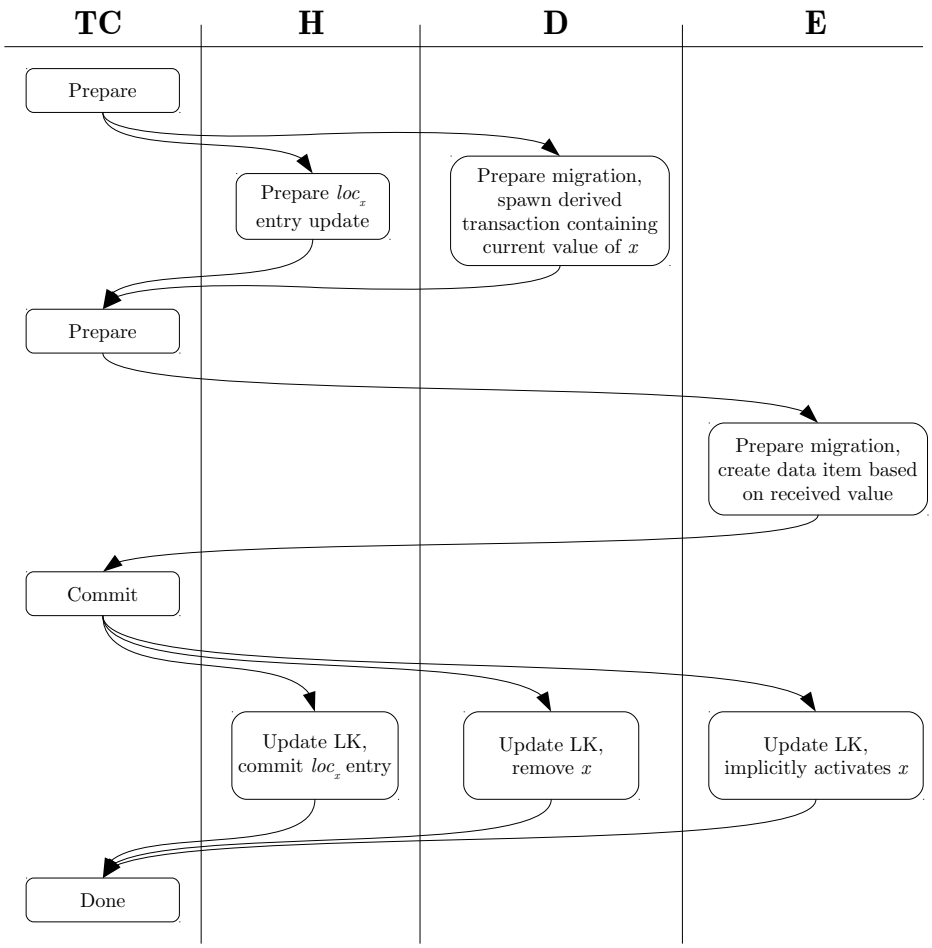


Figure 3.6: A migration transaction.

5. E stores the current value of x in its table buffer, is now prepared to take ownership of the item and sends back its *ready* vote. Note, however, that since E's local knowledge has not been updated, E will still reject incoming requests for x.

6. Once all parties have handed in their *ready* votes, the coordinator will instruct nodes to *commit*.

7. The knowledgeable node H will persist the lookup table entry loc_x to the underlying CloudTPS table the same way it would persist any other item in an UpdateMigration. At this point, loc_x has been persisted in CloudTPS. In addition, we have added a hook to the UpdateMigration that will update the node's local knowledge to keep it consistent with the underlying CloudTPS store. From this point on, incoming requests for loc_x are replied to with a pointer towards E.

8. D will start by removing x from its local knowledge[3], effectively preventing D from accept-

[3]If D and H are the same node, x's entry is not removed, but updated to point to E.

ing any further transactions involving x. At this point, there might be several transactions locally queued for x which need to be cleared. Read-only transactions are restarted while read-write transactions are aborted and can be re-executed[4]. In both cases, we piggyback a request to purge the migrated item's pointer from cache. Finally, x is removed from the table buffer.

9. All E has to do upon commit message arrival is to update its local knowledge (everything was prepared in the vote phase). Once the local knowledge has been updated, E is ready to serve requests for x.

There may be a window where local knowledge structures are not completely synchronized, for example when one node performs the commit before another. This inconsistency window does not break correctness, although it can result in incorrect routes. For example, if the knowledgeable node commits its changes before the source and destination nodes, a new request might be routed to the destination node before the destination node has officially taken ownership of the item. This would result in the destination node either aborting the transaction or forwarding it to whichever node it believes to home the item. This may cause some extra messages to be sent, but the system will eventually converge.

3.4 Fault Tolerance

Our current implementation is not resilient to failures nor ring resizes. The current approach would be to flush all local knowledge and local caches upon ring membership change, and have the system slowly populate the structures as if the system had been rebooted. Whilst this should guarantee correctness, it will bear a high performance penalty. This makes it impractical for production where capacity might be added to increase throughput, but would result in degraded performance as all cache entries would be flushed.

Fault tolerance and membership change could be handled similarly to vanilla CloudTPS, where virtual nodes get rearranged on membership change. The challenge is how to keep loc_x consistent between the knowledgeable node and the node that homes the item.

As a final remark on fault tolerance, when taking failures into account we find that the $lookup(x)$ method may return a pointer to a dead node. In this case, the pointer needs to be deleted and a new initial placement derived.

3.5 Implementation Details

There are several details to the relocation and migration mechanism beyond what has been covered in this chapter. This includes techniques to minimize the number of messages sent,

[4] As an optimization, CloudTPS stores read-write transaction state on remote nodes between restarts. Therefore, we cannot restart read-write transactions, as the set of involved nodes has changed.

suspending and resuming transactions during item location lookups and other optimizations. These details are intentionally left out, as they do not add to the overall design of the system. Further information can be found in this project's patch to CloudTPS.

Chapter 4

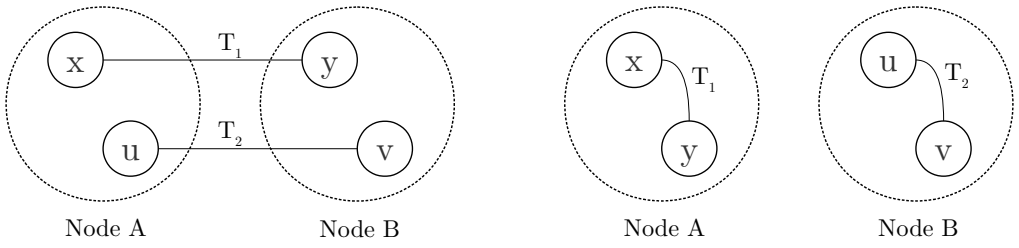
Data Item Placement Policy

In the previous chapter we discussed separation of policy and mechanism, and described the mechanism we have implemented within CloudTPS. This chapter will focus on the placement policy we have implemented on top of the mechanism.

The number of nodes required to service a given transaction is determined by item placement. Figure 4.1 illustrates two transactions, each involving two data items. In one case, both transactions T_1 and T_2 cross node boundaries and are carried out as distributed transactions. However, by rearranging data items we find that both transactions can be executed as local transactions. In general, the fewer nodes involved in a transaction, the lower its service time, and so the goal is to arrange items in such a way that this is minimized. To summarize: n data items are to be placed on m server nodes, such that we minimize the number of nodes involved in each transaction.

A placement policy may have more objectives than minimizing the number of nodes involved. For example, a policy may aim to spread work evenly amongst nodes in the cluster. These are both desirable goals, but may work against each other. The greatest item locality is achieved by placing all items on a single node as demonstrated in Figure 4.2. However, this would not meet load balancing goals.

There are several approaches as to how a placement policy could be implemented. This



(a) Distributed transactions.

(b) Local transactions.

Figure 4.1: Two possible placements: a) with both transactions distributed and b) with both transactions local.

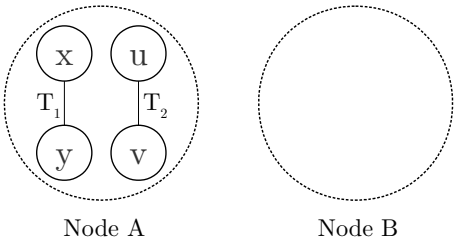
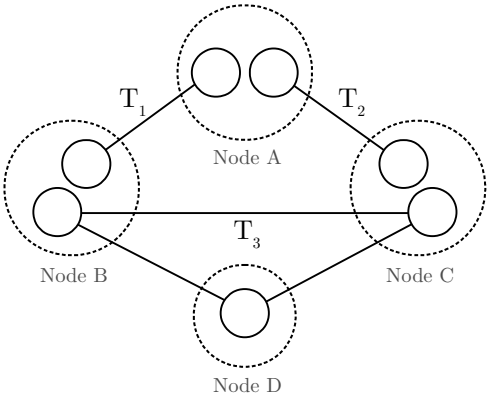


Figure 4.2: High locality can be achieved by naively placing all items on a single node. However, this causes load imbalance.



Node	Local history
A	T_1, T_2
B	T_1, T_3
C	T_2, T_3
D	T_3

Figure 4.3: Three transactions, T_1, T_2 and T_3, are executed on nodes $A + B$, $A + C$ and $B + C + D$. Each node keeps record of the transactions that it took part in.

depends on what input is used for decision-making. For example, placement policing can be carried out within an application, or by mining its data. However, these approaches require semantic information and are thus likely to be application-specific. We have chosen to log transactions and use workload analysis to feed our policing, as this approach has been shown to work for different workloads and access patterns [7].

4.1 Collecting the Transaction History

The first challenge with analyzing workload is collecting a transaction history for inspection in a scalable way and with minimal performance and resource usage impact. A centralized aggregation will not scale and so we choose to distribute the transaction history in such a way that each node keeps record of transactions that involved local items (see Figure 4.3).

Figure 4.4 illustrates how the transaction history is populated and transmitted, from the coordinator to other participating nodes. Once the coordinating Global Transaction Manager (GTM) on A completes processing the transaction, it notifies the local Workload Manager (WM). The WM logs the transaction locally and appends a log entry to a local send buffer which gets flushed at regular intervals (currently set to 10 seconds), gradually propagating remote

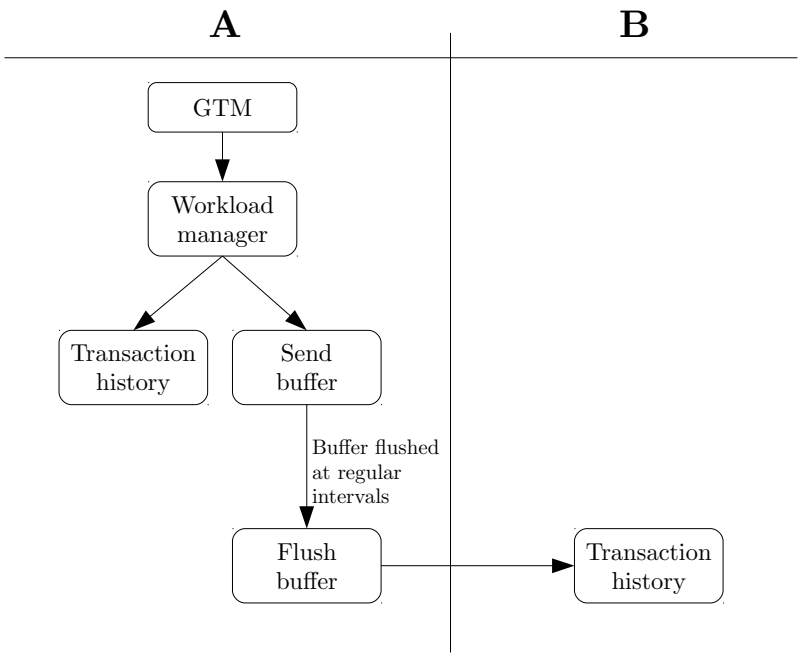


Figure 4.4: Transaction history message propagation.

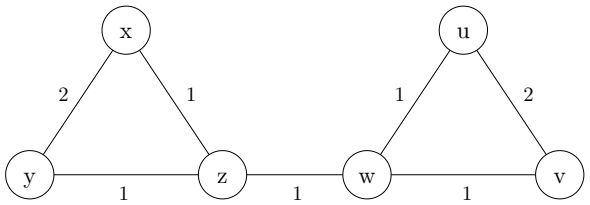
transaction logs. We should note that participating nodes cannot log the transaction at an earlier stage, for example when they receive their sub-transactions, as only the coordinator has the full transaction view and is therefore the only node that can construct the history message properly.

The transaction history is weakly consistent. In fact, it is not even guaranteed to keep record of all executed transactions. The current implementation uses a bounded list that is garbage collected to 60% usage at regular intervals. If items are added at a rate where they fill the history between garbage collection runs, new entries may be discarded. This design aims to limit the performance impact of collecting the transaction history. The Workload Manager runs as its own CloudTPS component, on its own thread, and thus has minimal impact on general transaction processing.

The transaction history can be seen as a sliding window, where the window size determines the amount of entries stored. The cluster is likely to adapt sooner to changes in the access pattern with a small window, at the risk of wave effects. A larger window will be more stable, at the cost of memory consumption and slower reaction to changes in the access pattern. The Workload Manager currently aims to log and equally weigh all transactions, but could for instance be adapted for sampling and apply exponential smoothing.

Transaction	Items
1	x, y, z
2	u, w, v
3	z, w
4	x, y
5	u, v

(a) Transaction log.



(b) Graph representation.

Figure 4.5: A graph representation of a transaction log.

4.2 Workload Interpretation and Partitioning

The global transaction history can be represented as a graph where vertices represent items, with edges drawn between items accessed within a transaction. Finally, edge weights represent the number of times items accessed together. Figure 4.5 illustrates a graph representation of a transaction history.

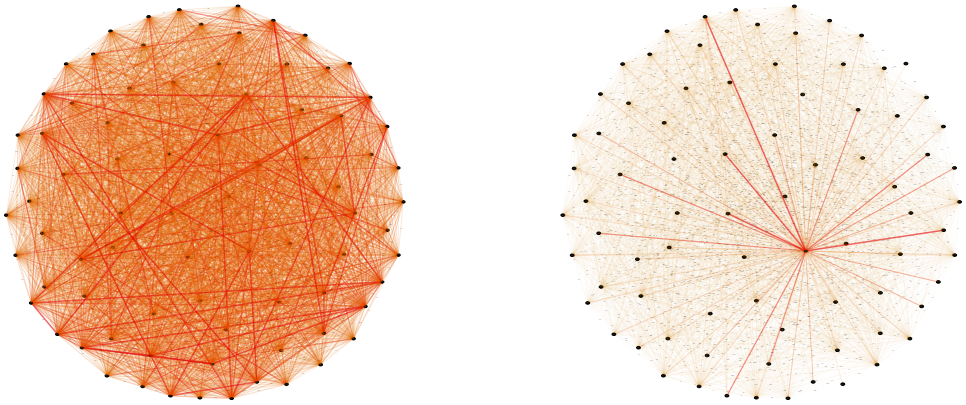
Once the graph has been constructed it should be split in k partitions, where k is the number of nodes in the CloudTPS cluster. The goal is to minimize the number of transactions that cross node boundaries. In other words, we would like to minimize the weight of edges crossing partitions, the *total edge cut*. This problem can be solved using partitioning algorithms such as Minimum k-cut, as done by Schism [7].

The advantage of using Minimum k-cut compared to hash placement is visualized in Figure 4.6, which illustrates a CloudTPS TPC-W trace simulated on an 80 node cluster. CloudTPS server nodes are drawn as vertices, with edges color coded based on the number of transactions spanning between any two nodes. The total edge cut when using hash placement (Figure 4.6a) is 3,435,423, compared to 497,549 when using minimum-k-cut (Figure 4.6b).

This advantage of minimum k-cut placement is further illustrated by the frequency graphs in Figure 4.7. A large number of transactions are carried out on 4 or fewer nodes when using minimum k-cut (4.7b), whereas a fair number of transactions required 7, and up to 11 nodes, when using random placement (4.7a).

4.3 Distributed Approach

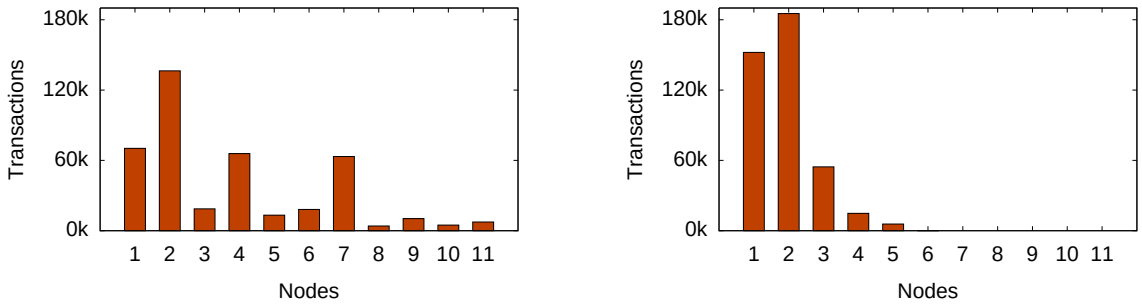
Whilst a centralized minimum k-cut partitioning scheme provides good partitions, it will soon become a bottleneck in a large distributed system. Firstly, aggregating the transaction history to a central machine becomes increasingly difficult as system throughput increases. Secondly, graph partitioning algorithms are complex and do not scale linearly. Therefore, there is an upper bound to how much a central component could partition in real-time. In this section we will focus on our research on how to implement minimum k-cut in a distributed setting; alleviating



(a) Hash placement.

(b) Minimum k-cut.

Figure 4.6: Graph representation of an 80 node partitioning using minimum k-cut. The total edge cut is reduced from 3,435,423 (hash) to 497,549 (minimum k-cut). Yellow denotes light weights whereas red denotes heavy weights. Yellow is better.



(a) Hash placement.

(b) Minimum k-cut.

Figure 4.7: Frequency graph showing the number of nodes involved per transaction.

the need for a central component.

We choose a gossip-style peer-to-peer approach, where nodes in the cluster contact other nodes at regular intervals. When gossiping, the two nodes exchange their transaction histories and make local optimizations to the data partitioning scheme. In theory, if these local optimizations work towards better over-all partitioning, this gossip-based approach should converge to close-to optimal state.

Each gossip session consists of the following steps:

1. **Gossip initiation**

   In the initiation phase, the "initiating node" chooses a "participating node" at random and attempts to establish a gossip session. As CloudTPS maintains complete membership, we simply choose a node at random from this global view. Although not tested, we believe that our approach will also work in an environment where the global view is unknown and nodes are drawn from a local view, maintained with gossip based peer-sampling [11]. The

participating node will either accept and establish the gossip session or reject the gossip, if already engaged.

2. **Exchange transaction logs and build graph**
   The participating node sends its complete transaction log over to the initiator, which constructs the transaction graph, as described in Section 4.2.

3. **Run Minimum k-cut**
   The graph is partitioned in $k = 2$ partitions using the METIS library [12]. When constructing the graph in our Java implementation, we take special care to use the same data structures as the METIS library. This way, using the Java Native Interface, we can pass references to these data structures directly to the METIS C-library, avoiding unnecessary memory copies.

4. **Interpret partitioning and migrate items**
   Finally, the METIS output is interpreted and items are migrated, using MigrationTransactions as described in 3.3. The current implementation migrates items serially, one-by-one. Items could be migrated in bulk, and multiple MigrationTransactions submitted in parallel, but this could have a higher performance impact on the cluster. Only once all items have been migrated, the nodes will round up their gossip session and become available to engage in upcoming gossips.

4.3.1 Simulation

The gossip-based distributed partitioning was simulated using the same TPC-W trace as referenced in Section 4.2. The simulation was carried out in two phases. In the first phase the complete trace was processed, building up a transaction history. In the second phase, gossips were triggered one cycle at a time, recording the over-all placement after each cycle.

Figure 4.8 shows the results of our first simulation as total edge cut (total weight) over time (gossip cycles). For reference, the hash and optimal k-cut placement's edge cut is drawn in green and blue, respectively. The red curve represents the total edge cut after each cycle, and we find that it converges to the optimal k-cut placement.

4.3.2 Anchor Nodes

Under normal circumstances, graph partitioning has complete graph information. This is not the case in our distributed approach, which reveals a problem where the number of items migrated per cycle does not converge to zero. Figure 4.9 illustrates that even after 100 cycles, long after the gossip-based and optimal k-cut total edge cut curves have converged, the system is still migrating a large number of items ($\sim 6\%$) each cycle.

The root cause for this high migration count is that the graph does not represent the item's current placement. This is best explained by means of an example. Let us say that nodes A and

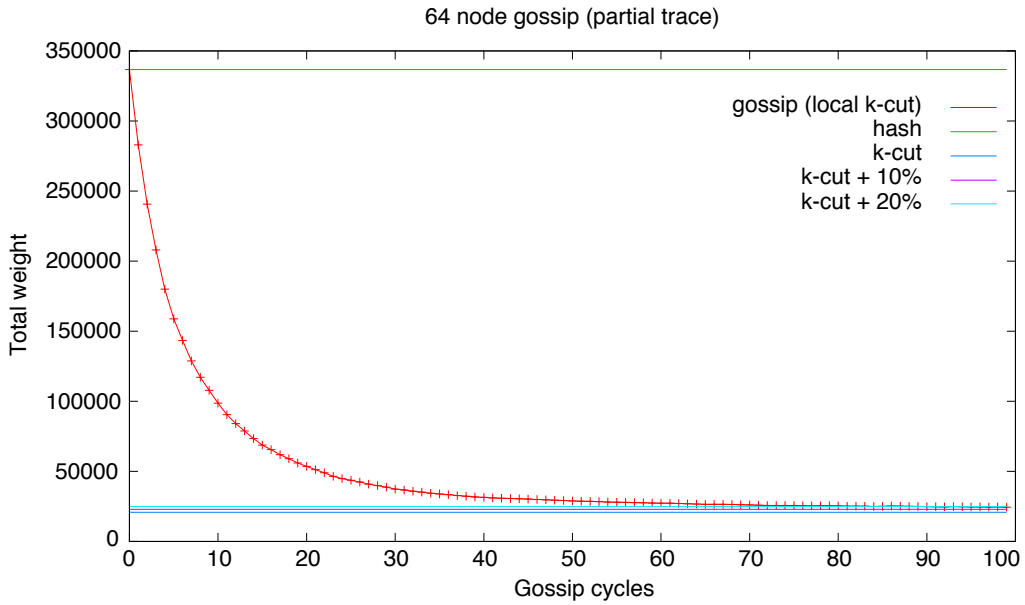


Figure 4.8: Total edge cut (total weight) for hash, optimal k-cut and our gossip policy over time (gossip cycles).

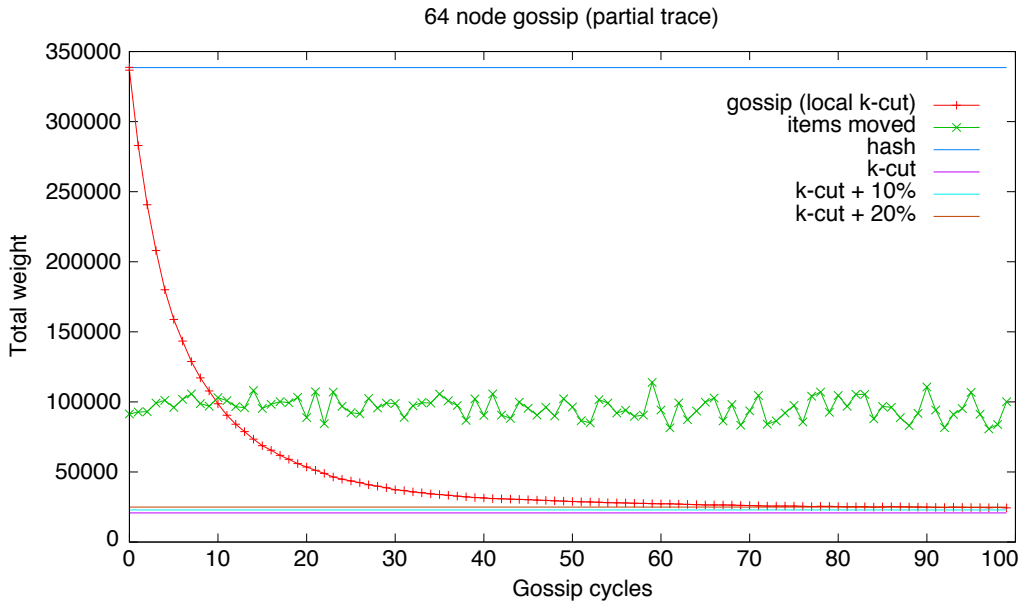
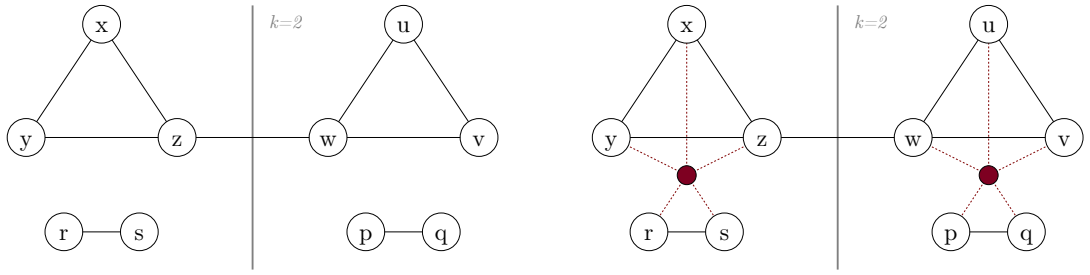


Figure 4.9: Number of items migrated does not converge.



(a) Basic graph.

(b) With anchor nodes.

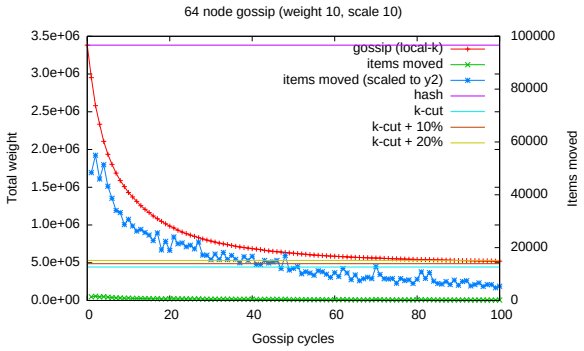
Figure 4.10: Anchor nodes give the partitioning algorithm an incentive to keep items in place, if their location does not affect the edge cut.

B are gossiping and their collective transaction histories are represented by the graph in Figure 4.10a. The transaction graph is not necessarily connected and thus certain tuples, such as (r, s) and (p, q), are placed in either partition, based on chance. As a result, these two tuples can drift from one node to another, even if the migration has no effect on the total edge cut.

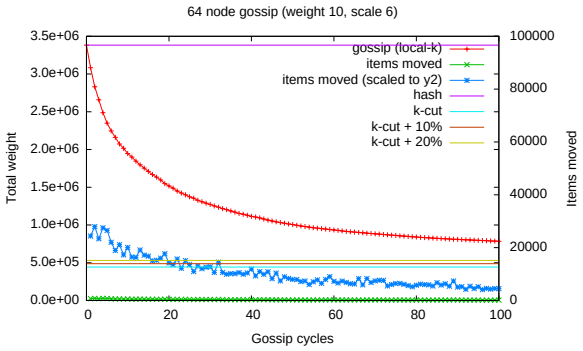
To moor these drifting items we introduce *anchor nodes*. Two anchor nodes are created, one for each gossip participant, and each node connects to all items stored on the respective server node. Following the previous example, Figure 4.10b demonstrates how the two anchor nodes connect to local items. By holding on to items, the anchor nodes give the partitioning algorithm an incentive to leave nodes in place. Figure 4.11a shows the same TPC-W trace simulated, but with anchor nodes. First, we note that the number of items migrated decreases over time. Second, we note that far fewer items are migrated than before. Previously there was an average migration count of $\sim 6\%$ whereas now the count starts at $\sim 3\%$ but quickly drops below $\sim 1\%$.

The anchor nodes provide incentive to keep items in place, and this incentive can be increased or decreased at run-time by changing the weights of the edges between anchor nodes and local items. The heavier the weight, the harder the anchor node will fight to keep the items locked in place. This incentive can be used to take the cost of migration into account when evaluating whether to migrate data items. Figure 4.11 demonstrates how varying anchor edge weight affects total edge cut convergence and the number of items moved. The higher the weight, the fewer items are moved during each cycle and the slower the system converges.

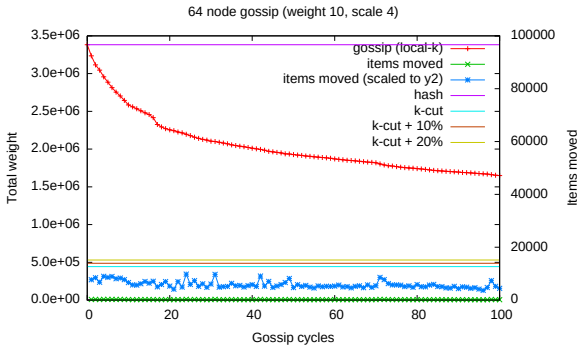
The varying anchor weight could also be used to focus on moving only the more important items. For example, if after running the k-cut partitioning the number of item migrations were above a certain threshold, the partitioning could be rerun, with increasing anchor weight, until the number of items to migrate drops below the set threshold. This should provide much better results than migrating only a subset of the first output. We should note that the graph does not need to be rebuilt to change the anchor weight. The same structures can be reused, slightly modified, and passed by reference to the METIS library.



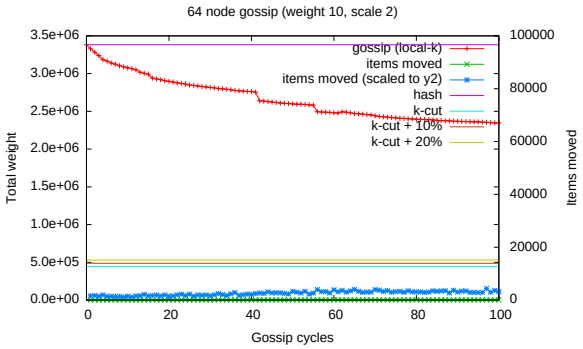
(a) $weight = 1$



(b) $weight = 1.6$

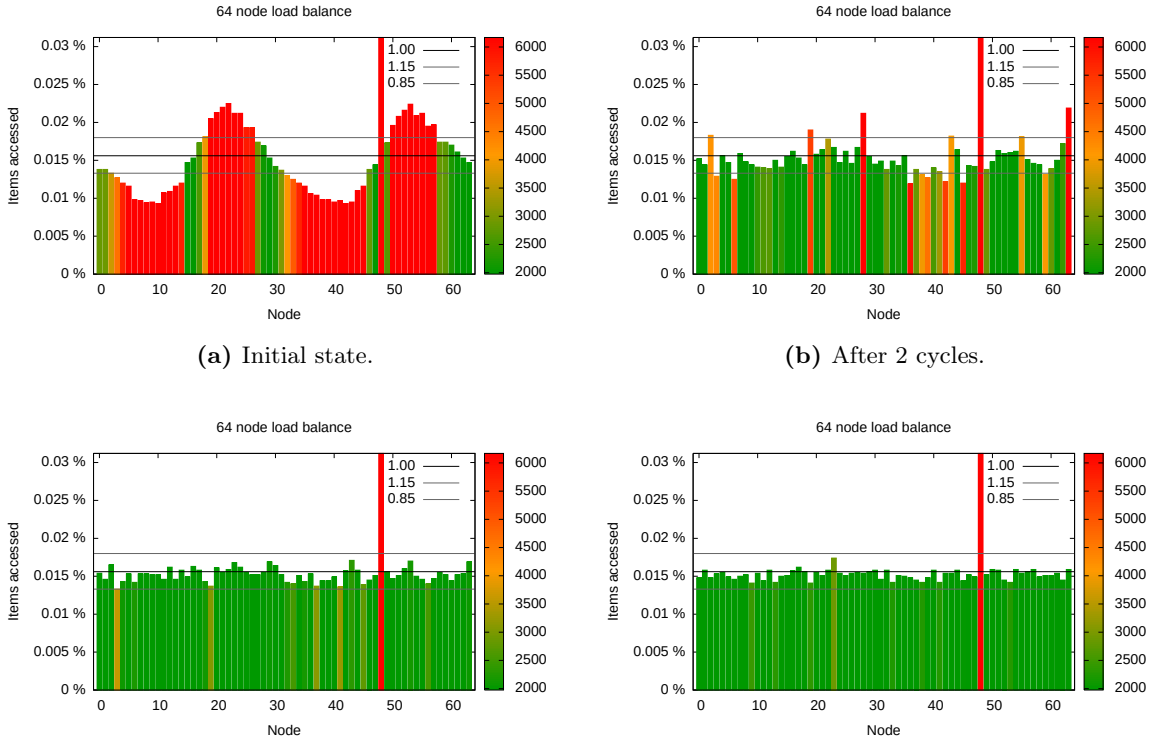


(c) $weight = 2.4$



(d) $weight = 5$

Figure 4.11: The item migration count and speed of convergence can be adjusted by changing the anchor edge weight.



(a) Initial state.

(b) After 2 cycles.

(c) After 4 cycles.

(d) After 6 cycles.

Figure 4.12: The system quickly balances load amongst nodes.

4.4 Load Balancing

By utilizing the multi-constraint k-way partitioning features in the METIS library, it was also possible to incorporate load-balancing into the partitioning. We added weights to the vertices in the graph, having weights denote the item hit count.

Figure 4.12 shows the same TPC-W as before starting out in an unbalanced state and quickly converging to a balanced state.[1]

Node 48 is of special interest. It remains read, and according to our metrics, highly loaded. Upon further inspection, it turns out that this node is only storing a single item. The item in question was `country: USA`, which appears in a high percentage of transactions. This shows that using item hit count alone may not be sufficient.

[1]We define a balanced state such that load imbalance is less than 15%.

Chapter 5

Evaluation

This chapter presents a performance evaluation of the lookup table (LT) mechanism. The evaluation is organized in two parts. Firstly, we evaluate the costs introduced by using lookup tables. This was done by comparing unmodified CloudTPS (Vanilla) and our modified LT CloudTPS with random data placement (LTrand). Secondly, we evaluate the gains obtained by using a good placement policy. This was done by preloading the lookup table with a placement derived from running the policy described in Chapter 4 (LTopt).

All benchmarks are performed using CloudTPS's TPC-W Zipf-enabled benchmark, with pre-generated sequences to work towards repeatable results. We deploy CloudTPS over twelve nodes on the DAS3 cluster at Leiden University [1]. Each node has one dual-core 2.6 Ghz processor and 4 GB of RAM. We used nine additional nodes for the benchmark: two clients nodes, one time-stamp node and six HBase nodes, implementing the underlying persistent cloud storage.

5.1 Lookup Table Overhead

The first metric of interest is the overhead introduced by the lookup tables, regardless of placement policies. We compare our LT CloudTPS version (LTrand), without any policy, to Vanilla CloudTPS. When lookup tables are used without a policy, items are randomly placed using hashing, analogous to Vanilla CloudTPS, and thus comparing the two will allow for estimating the added overhead.

As expected, LTrand latencies in Figure 5.1 are higher than the Vanilla latencies, averaging at 13.3 ms compared to Vanilla's 8.3 ms. The increased latency is mostly caused by two factors: cache misses, where the item location is present neither in the local cache nor the local knowledge; and initial placements, where the item's location is completely unknown and has to be created, and written to persistent storage, before continuing. These operations involve internode communication and can be blocked by other transactions, and are therefore costly. For example, an initial write of item x will involve a cache read miss from the local node's cache, a request to the knowledgeable node, a cache read miss from the knowledgeable node's local knowledge and an attempted read from persistent storage. Since there is no placement persisted, an initial

placement will be devised and written to persistent storage. This is an example of the worst case scenario, which, however, will occur once for each new item added to LT CloudTPS.

5.2 Optimal Placement

To find a good data placement, we recorded a trace from the previously described TPC-W benchmark and fed through 50 cycles of the gossip based simulator introduced in Chapter 4.[1] The output was then preloaded into persistent storage for the lookup table to use (LTopt).

Figure 5.2 extends Figure 5.1 by adding request latency for the close-to-optimal placement. With item placement we make up for much of the overhead introduced by lookup tables. Even so, at a 7.90 ms average request latency, LTopt only marginally outperforms Vanilla CloudTPS, at 8.38 ms. Further, while the average LTopt latency is lower than Vanilla (Figure 5.2a), we note that the LTopt 99th percentile is still considerably higher (Figure 5.2b). As discussed in Section 5.1, these higher 99th percentiles are expected. All three configurations have similar throughput during the experiment.

5.3 Discussion

In this section we will discuss three topics and how they affected these results: cache misses, node imbalance and the workload generation.

5.3.1 Cache Misses

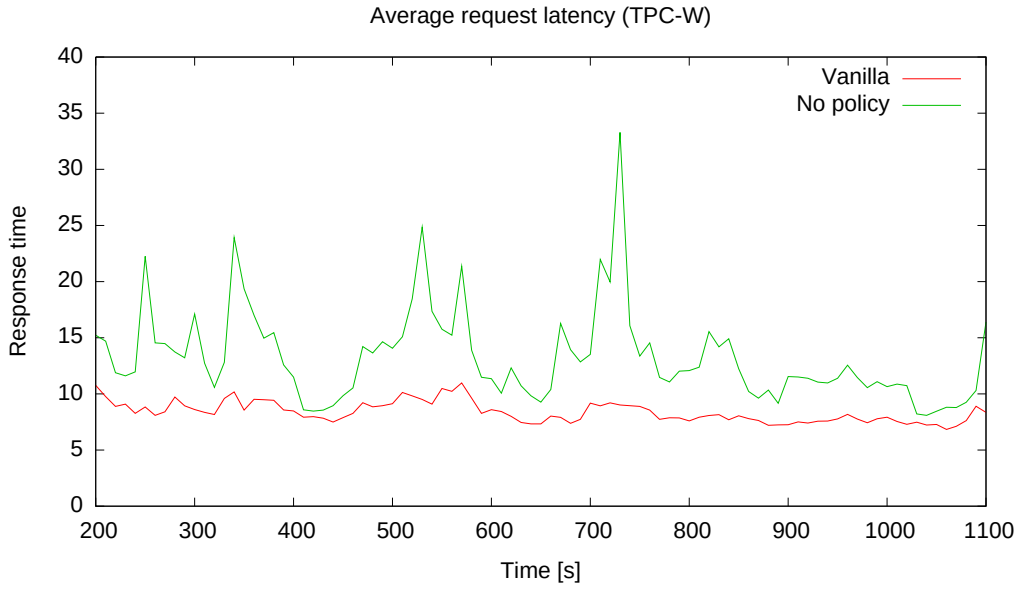
Performance is determined by the number of nodes involved in transactions, and by the cache hit ratio. The higher the hit ratio, the fewer pointers need to be read from persistent storage, resulting in lower overhead.

Figure 5.3 plots Vanilla and LTopt latencies over time on the y-axis, and the cache miss ratio over time on the $y2$-axis. This strongly suggests a direct relation between cache miss ratio and LTopt latencies. Figure 5.3 draws out this this relation and shows that for our workload, LTopt outperformed Vanilla when cache misses were below 17.4%. These data show that LTopt could do better than Vanilla over-all when fronted with a lower cache miss ratio workload. For example, between times 420 and 470, when the cache misses drop below 7%, LTopt's average latency of 5.42 ms is 33% lower than Vanilla's average of 8.11 ms.

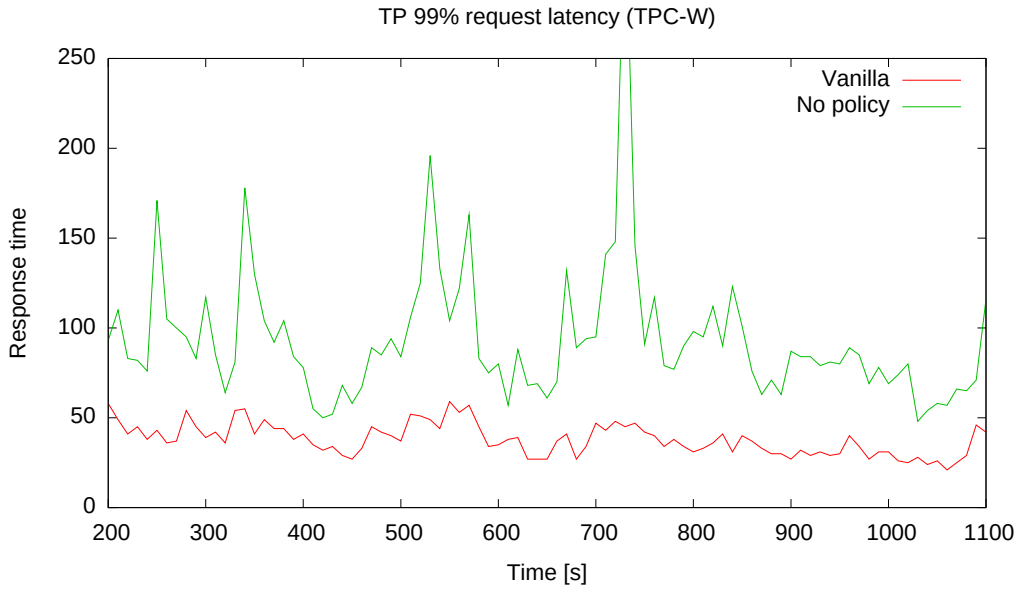
5.3.2 Node Imbalance

The policy presented in Chapter 4 included methods for load balancing items across nodes. Figure 5.4a plots the number of items accessed per individual node. Judging by the deviation graph in Figure 5.4b items are read fairly uniformly across nodes. Transient highs and lows are

[1] The trace was fed to the simulator as a single sample, and the simulation did not therefore account for changes in access pattern.

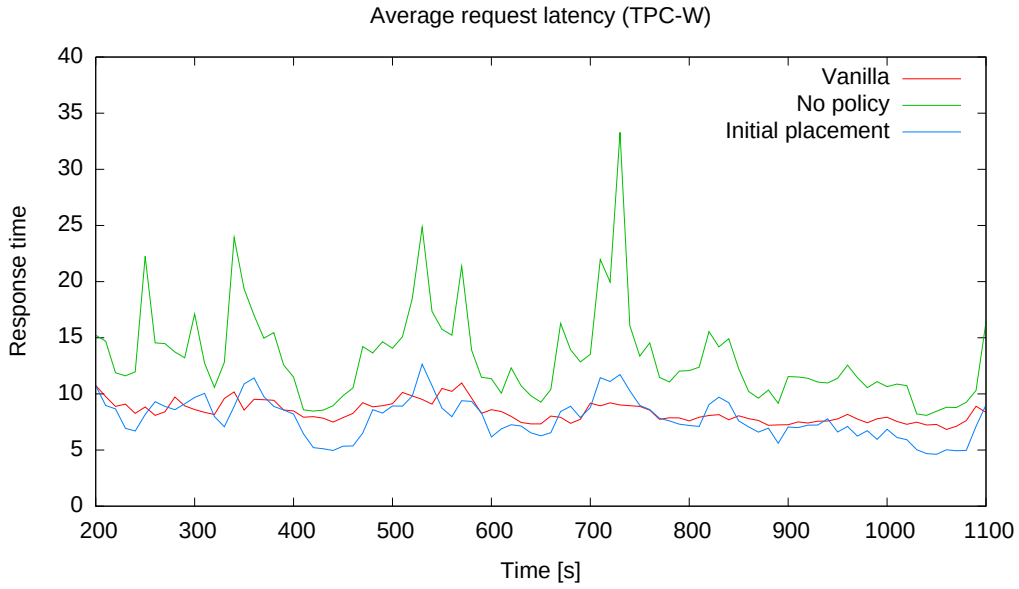


(a) Average request time.

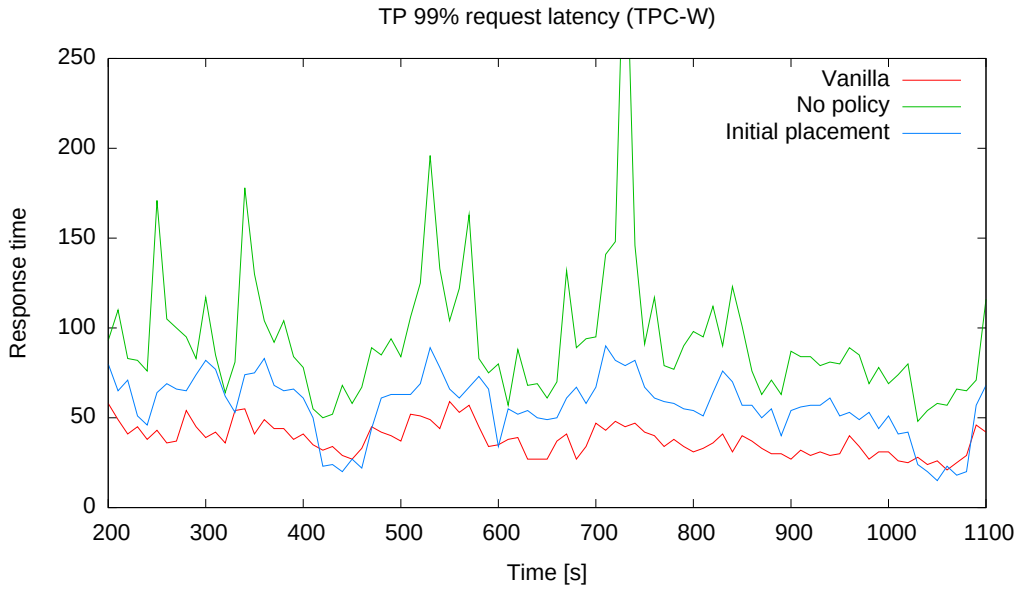


(b) 99th percentile of response times.

Figure 5.1: Lookup table overhead is the area between the Vanilla CloudTPS and lookup table CloudTPS curves.

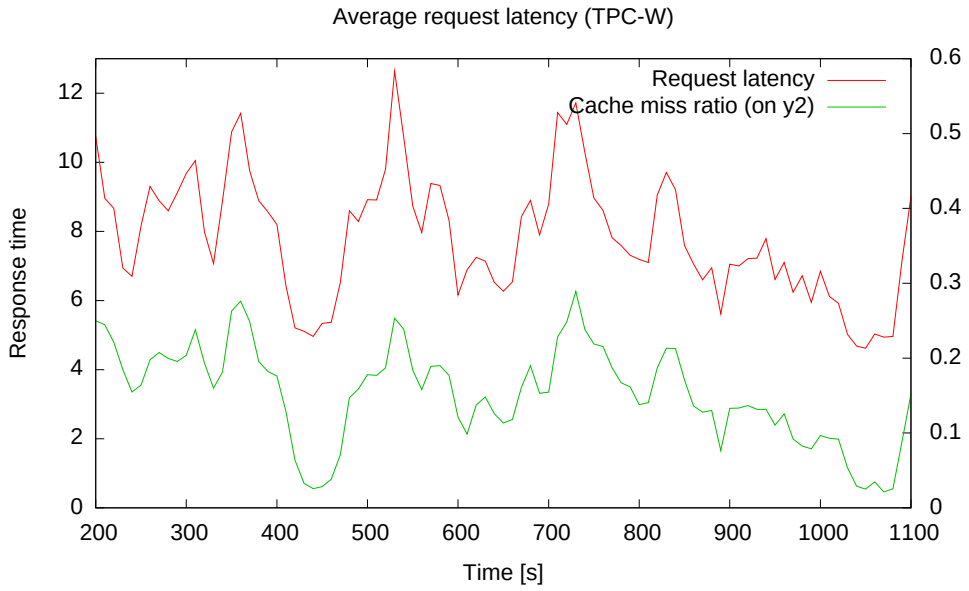


(a) Average request time.

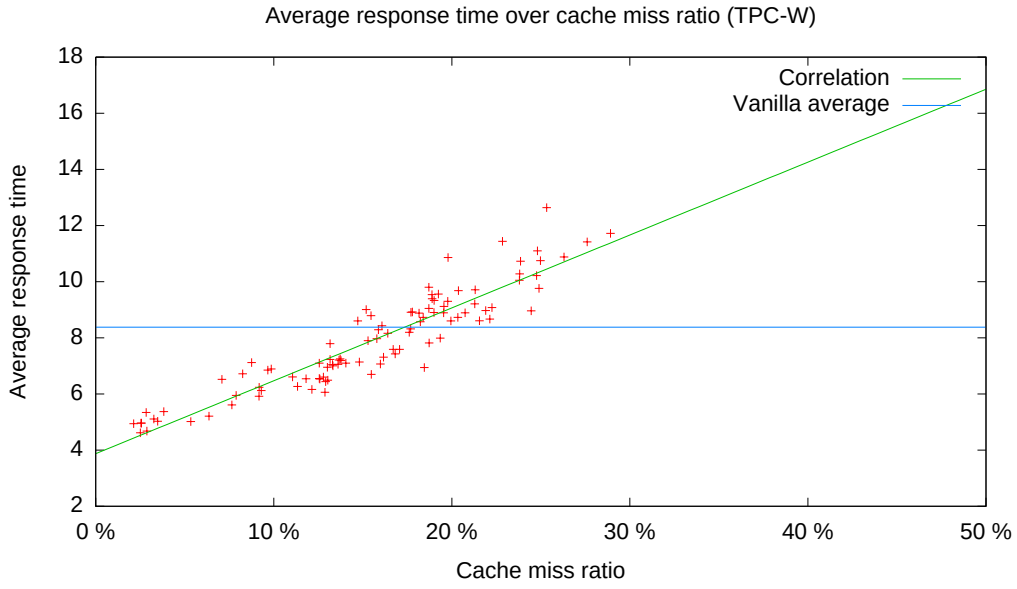


(b) 99th percentile of response times.

Figure 5.2: Comparison of Vanilla CloudTPS, lookup table CloudTPS with no policy and lookup table CloudTPS with close-to-optimal placement.



(a) Latency and cache miss ratio over time.



(b) The intersection at 17.4% shows that lookup tables outperformed Vanilla when cache misses were below 17.4%.

Figure 5.3: Relation between latency and cache miss ratio.

expected, as the load balancer operates and averages over the complete period, and therefore does not account for items read times within the period.

The same cannot be said for transaction coordination. The number of locally coordinated transactions is not properly balanced, as demonstrated by Figures 5.5a and 5.5b. Since the node coordinating a transaction does considerably more work than other participating nodes, this can have a detrimental effect on overall performance and latency.

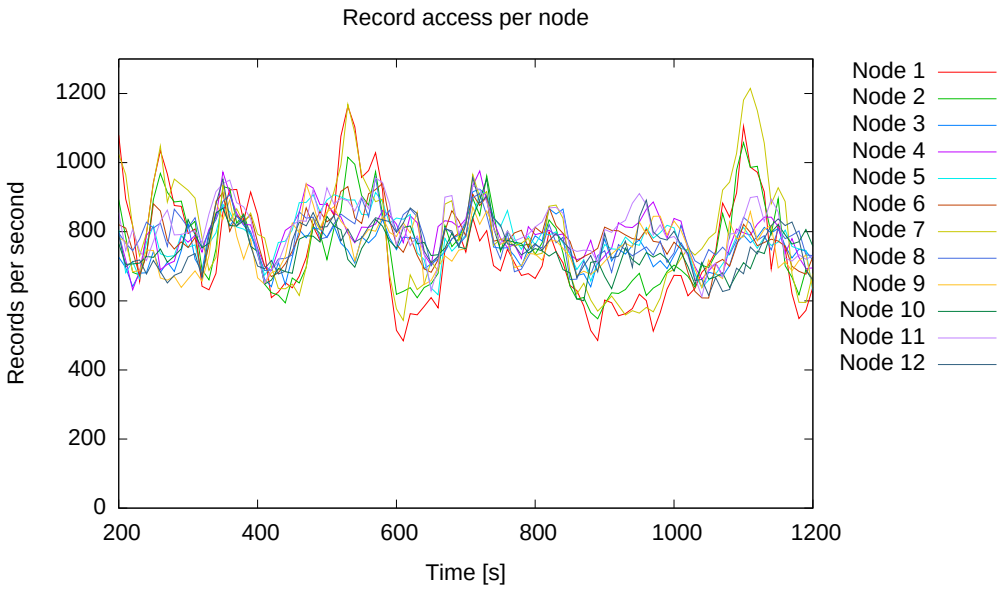
This imbalance was caused by an optimization in transaction dispatching. When a Vanilla ClouldTPS client submits a transaction involving items x and y, it will submit the transaction to a random node in $\{hash(x), hash(y)\}$. If items are evenly placed, then transactions will similarly be evenly coordinated. In the lookup table implementation, a faulty optimization broke this random distribution. Instead of choosing from $\{hash(x), hash(y)\}$, LT chooses from $\{lookup(x), lookup(y)\}$. If neither of those pointers are locally cached, the client will attempt to resolve both pointers and submit the transaction according to the first dereferenced pointer. However, and this is where the optimization violates even distribution, if any pointer in the set is locally cached, the set is reduced to include only the locally cached pointers. For example, let us assume that a client has cached pointers for $lookup(x)$ and $lookup(y)$ and is about to submit a transaction involving items x and p. It will first construct the set $\{lookup(x), lookup(p)\}$, but the optimization phase will reduce this to $\{lookup(x)\}$, causing the transaction to be always submitted to $lookup(x)$ and never to $lookup(p)$. A potential solution, in the case of transaction x and p, is to dispatch the request to x and asynchronously resolve the location pointer for p, resulting in subsequent submissions to be randomly dispatched. This imbalance, which initially appeared to be caused by faulty placement policy, demonstrates the importance of benchmarking both policy and mechanism.

5.3.3 TPC-W Trace

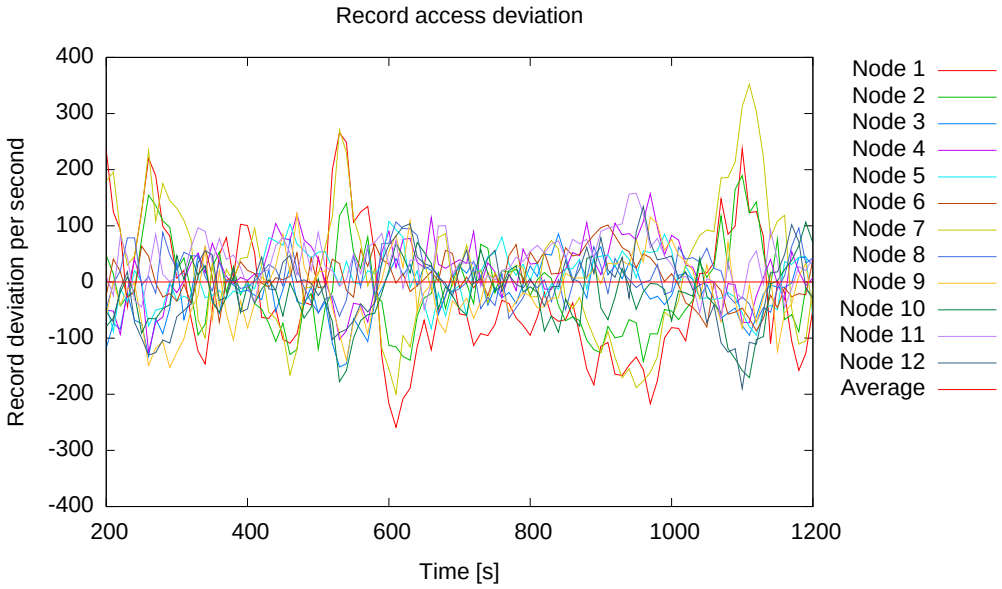
Our evaluation was based on a CloudTPS TPC-W workload, modified to emulate locality. The locality emulation was implemented by drawing customer and item IDs from a Zipf distribution.

This makes certain items more popular than other, and certain customers more active than others. This is desired, but the default Zipf exponent of 1 resulted in a very skewed workload with several extremely hot items and a very long tail. The single most popular item, for example, was referenced in over 8% of all transactions. This does not skew the LTopt to Vanilla comparison, but it does create a bottleneck on the node servicing this item, as item access is serialized by the item's local transaction manager, and this may therefore not demonstrate the system's full potential.

Lookup table CloudTPS's greatest opportunity is with transactions involving several items. For example, in our simulations, the policy reduced transaction span of large transactions from 7 - 11 nodes to 2 - 4 nodes (see Figure 4.7). Table 5.1 lists latency improvement for two popular transactions involving the most common TPC-W order ID. These transactions involve 7 - 9 items and make up for 1% of transactions each. The placement policy reduced the span from 6

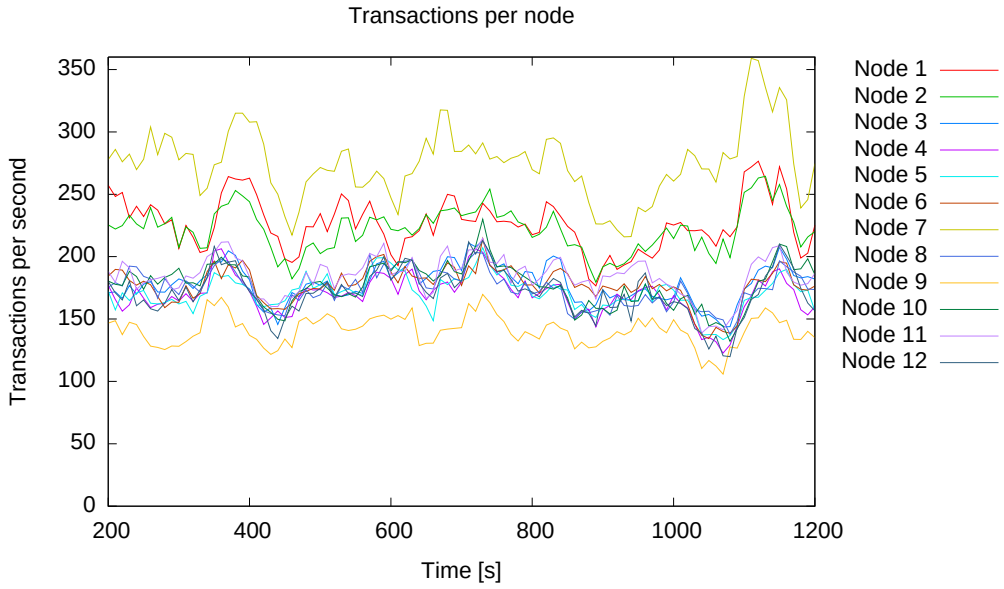


(a) Item access.

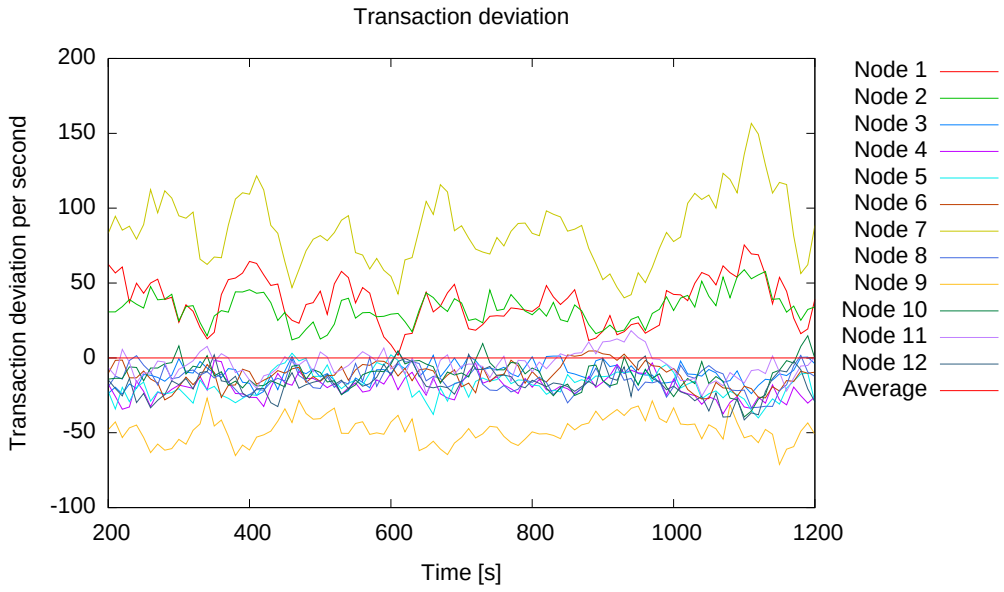


(b) Deviation.

Figure 5.4: Item access appears properly load balanced amongst nodes.



(a) Coordinated transaction.



(b) Deviation.

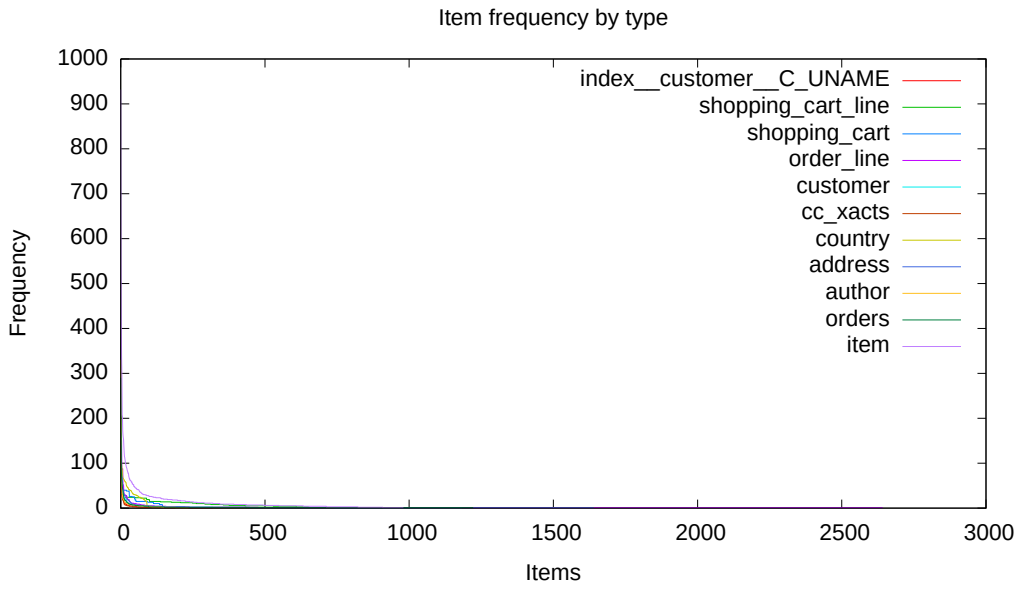
Figure 5.5: Transaction coordination is not properly load balanced. Three nodes dominate transaction coordination.

	Vanilla	LTopt	Improvement
Average	7.8 ms	1.5 ms	80%
50th percentile	6 ms	1 ms	83%
90th percentile	12 ms	2 ms	83%
95th percentile	17 ms	3 ms	82%
99th percentile	38 ms	6 ms	84%

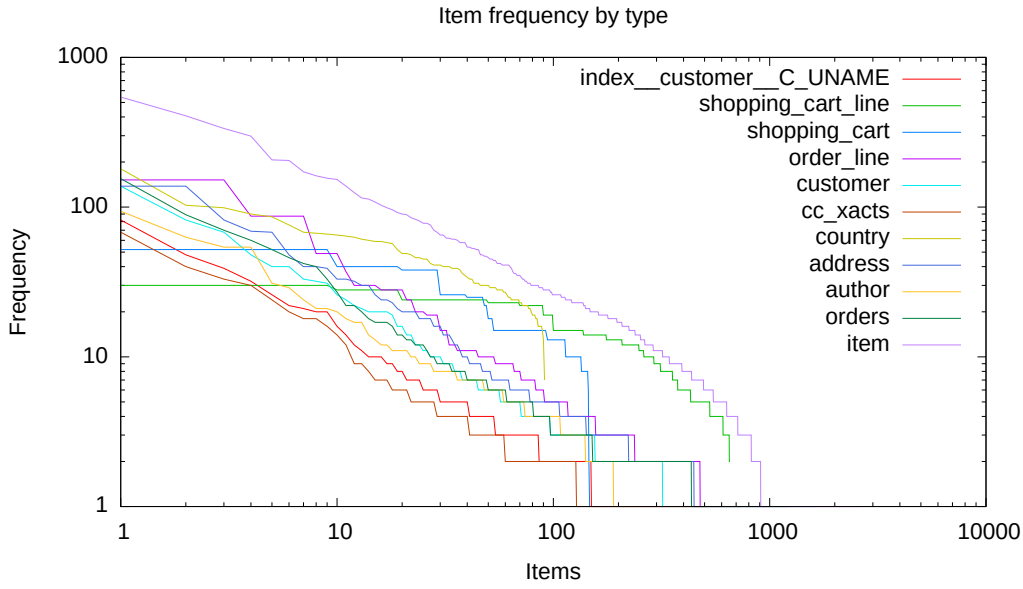
Table 5.1: Latency improvement for two types of transactions involving the most common order, as seen by the transaction coordinator. Transaction span reduced from six nodes in Vanilla to one in LTopt.

nodes in Vanilla down to 1 node in LTopt, and as a result, average latency dropped by 80%. In addition to average improvements, the 95th and 99th percentiles show similar improvement, an important metric when reducing jitter and striving towards good overall performance. Despite the improvements on these most popular transactions, the vast majority of the larger transactions are spread over the Zipf's distributions long tail and thus executed only a few times, or in the worst case, only once. As a result, they do not get the opportunity to make up for the cost of their initial cache miss.

In this context it is worth remembering that LT CloudTPS enables the operator to select whether to use LT or hash placement on a table by table basis. An operator could therefore choose to use lookup tables for tables with frequent transaction access patterns, while leaving mostly-write or random-accessed tables in hash placement mode.



(a) Linear scale.



(b) Log-log scale.

Figure 5.6: Item frequency, by type.

Chapter 6

Conclusion and Future Work

The focus of this project was to study data placement and whether latency can be reduced by clustering related items and thus limiting the number of nodes involved in each transaction. We developed a distributed placement policy, and an item placement mechanism on top of CloudTPS, to evaluate our policy. The mechanism features an item placement lookup table, distributed transaction history aggregation and transactionally-consistent live item migrations.

Our gossip-like policy has the potential to scale very well, in contrast to previous centralized approaches, while still converging on similar results. The policy includes load-balancing and a novel cost-model which can prioritize item migrations, so that items that greatly benefit from relocation are migrated first. This allows operators to throttle the rate of convergence, depending on system load.

Using lookup tables, backed by a good placement policy, can clearly outperform random placement for certain workloads. When less than 17% of requested items are new to the system, the lookup table version outperforms random placement with up to 80% reduction in latency averages and 99th percentiles. Our work reinforces the results of Schism and extends these by demonstrating that data placement can realistically be done live at runtime, without the need for a centralized component.

Future efforts should be directed towards gathering more, and different, traces, and analyzing the system under various load characteristics. In this respect, starting with several artificial workloads, which in theory should demonstrate both sweet spots and weaknesses, could be beneficial to gain a deeper understanding of system behavior. Additionally, there are several aspects that should be understood better, such as the impact of varying transaction history window size, transaction sampling, performance of the cost-model under different workloads and the feasibility of partitioning and placing items at a coarser granularity than item-by-item.

Bibliography

[1] The Distributed ASCI Supercomputer 3. http://www.cs.vu.nl/das/.

[2] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[3] E.A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[4] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[6] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[7] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[8] J. Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.

[9] J. Dean. Building Software Systems at Google and Lessons Learned. *Stanford EE Computer Systems Colloquium*, November 2010.

[10] 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. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[11] M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8–es, 2007.

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359, 1999.

[13] M. Mehta and D.J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.

[14] J.W. Romein, A. Plaat, H.E. Bal, and J. Schaeffer. Transposition table driven work scheduling in distributed search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 725–732, 1999.

[15] J. Rothschild. High performance at massive scale – lessons learned at facebook, 2009.

[16] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, 2001.

[17] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[18] W. Zhou, G. Pierre, and C. Chi. CloudTPS: Scalable transactions for Web applications in the cloud. *Services Computing, IEEE Transactions on*, (99):1–1, 2011.