Armada : an evolving database system
Groffen, F.E.

Link to publication

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
6.1 Introduction

The power of the Armada model to point to the appropriate direction from each point in the system, yields in a converging search for data. However, instead of reaching the data directly after a single catalog lookup, multiple steps can be necessary to reach the data being looked for.

With an Armada system growing, the lineage trails grow along in size. The effect of such longer trails is the growing probability of needing more steps to reach the data being looked for. In this chapter we study the process of following the redirects from the Armada model. We focus on the costs associated to the process of following for different Armada systems. Since a traditional non-distributed system would have direct access to the data in any case, in comparison an Armada system introduces extra work caused by the redirects. As this is a given, next to identifying its costs, we experiment with different approaches to minimise their effects. The content of this chapter has been presented as workshop paper [31].

Throughout this chapter we frequent the terms agent, site and box. While these terms are defined in previous chapters, their definitions may be blurred due to various usages. In this chapter when we refer to an agent, we refer to the entity in the system that interacts with the data nodes (sites) in the Armada system. A site in there is a data node, capable of storing boxes in the Armada system. A box is a logical block of data hosted on a site, being either active or inactive. An active box points to data on the host site, an inactive box points to one or more other boxes that should be searched instead of that box.
6.2 Connection Costs

Each connection that is made to a database has some overhead caused by handshakes as part of the initialisation rituals on both the protocol level, as well as on the TCP stack. Table 6.1 depicts the results from a small experiment conducted to show that creating a connection to a database is an expensive operation. In the Table, the wall-clock times for performing 1000 “SELECT 1” queries using a client tool in seconds are shown for three different Open Source database systems. For each database, the left column shows the time it took to perform the thousand trivial queries over a single connection, while the right column shows the time for the same queries, but each over its own connection using a new client tool invocation. While the numbers are bound to the used software versions, the table clearly shows that creating connections is substantially more expensive than reusing the same connection.

In the experiment we tried to eliminate the overhead of query parsing, processing and execution, by taking a very simple “SELECT 1” query. The used operating system is OpenSolaris snv_101a on an AMD Athlon64 3800+. Performing 1000 executions of a very simple application (echo), to try and determine the operating system costs of executing the client utility averages to 7.3 seconds. This time is not subtracted from the right columns in Table 6.1. We used the 64-bits versions of the database software, PostgreSQL 8.2.7, MySQL 5.1.21.beta and MonetDB Nov2008. Subtracting the operating system overhead from the 1000 connections measurements, the latter still are 158, 150 and 189 times slower for PostgreSQL, MySQL and MonetDB respectively. For this reason it seems beneficial to try and reduce the number of connections an agent has to make during the query process, since this takes a substantial amount of time.

<table>
<thead>
<tr>
<th>run</th>
<th>PostgreSQL</th>
<th>MySQL</th>
<th>MonetDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.131 28.415</td>
<td>0.099 22.159</td>
<td>0.143 30.824</td>
</tr>
<tr>
<td>2</td>
<td>0.135 28.440</td>
<td>0.098 22.117</td>
<td>0.116 30.998</td>
</tr>
<tr>
<td>3</td>
<td>0.147 28.435</td>
<td>0.098 22.162</td>
<td>0.118 30.364</td>
</tr>
<tr>
<td>4</td>
<td>0.129 28.435</td>
<td>0.099 22.113</td>
<td>0.118 30.192</td>
</tr>
<tr>
<td>5</td>
<td>0.130 28.421</td>
<td>0.099 22.122</td>
<td>0.119 30.182</td>
</tr>
<tr>
<td>avg</td>
<td>0.134 28.421</td>
<td>0.099 22.135</td>
<td>0.123 30.512</td>
</tr>
</tbody>
</table>

Table 6.1: Wall-clock times in seconds for performing 1000 queries over a single connection (left) or over separate connections (right).
6.3 Metrics

Each measurement needs a way to describe the observed facts. Those descriptions are expressed using a given metric. Within our experiments on the Armada model, we are primarily interested in the performance of agents that navigate through the system. This performance depends on several factors, which we try to address given the following metrics.

**hopcount** The number of steps taken by the agent for a query from the starting site to the site holding the active box with the value being looked for. An agent that directly contacts a site which contains an active box responsible for the data value it looks for, has \( hopcount = 0 \) for that particular query. The bigger \( \text{avg}(hopcount) \) becomes, the worse the seek performance of the Armada.

**sitehits** The number of hops to a site by the agent. When an agent makes a hop to a site it also performs an action on it. \( sitehits \) expresses the importance of a site by means of how many times it is hit. A query with \( hopcount = 0 \), increases \( sitehits \) by one.

**sitequeries** The number of actual queries performed by a site for the agent. This value is an indication for the query workload. When data or queries are skewed, the distribution of sitequeries among the sites shows a clear peak. A query is an action performed on a site, i.e. an insert, update or select.

**sitetraffic** The number of hops made by an agent from a given site to another site in the Armada system. Per site this can result in multiple incoming as well as outgoing hops, referred to as traffic. Frequently used traffic paths are an indication for classic bottlenecks.

**sitefree** The percentage of available space to store tuples. This can give insight on how well equally loaded with data sites are, and hence if the data is spread equally.

Further, we have the site capacity and box capacity. Also, multiple data boxes can be placed on a site, which allows to reuse a site (sitefree) if no empty sites are left.

It is seducing to think of the sites in the association tree as an in logical rings around the origin ordered system. As such, each ring is another step away
from the origin. It cannot be said that the hop count to reach one site from the other equals the ring distance as the actual traversed path may take more hops given the tree structure of the association tree. More importantly, the ring distance only works as long as sites are used only once for storing a box. As soon as a site is “reused” the association tree becomes cyclic. With cycles in the graph it is no longer trivial to put sites on a ring. Apart from if or if not it would be possible to define a rule to still do the ring assignment, it becomes unclear how the resulting rings should be interpreted. Therefore any metrics that rely on ring distances are bound to become void once cycles appear. It is to be expected that cycles become a natural part of Armada to achieve a higher storage load, spreading throughout the available nodes and a way to gain access to trail information from other parts of the tree. The latter can help to reduce the hopcount.

6.4 Policies

To study the effects of various facets of a simulation, those facets need to be changed following a strategy to prove or falsify a given hypothesis. We identified four facets that are important for the performance of an Armada implementation in speed and space utilisation. For each facet, we identify a number of policies that define a certain action or behaviour for the given facet.

6.4.1 Chunk Policies

Chunking splits data from one box into two new ones. In our experiments, we use adaptive chunking. The chunk function is the divider function, that splits the original box in two. Further, for each chunk operation, one of the new boxes is placed on the site of the original box. As a result only one new site has to be found, and the amount of data that needs to be physically moved is limited.

With adaptive chunking, the chunking operations that are executed fully automatic are influenced by the environment as observed by the box. A simple idea is to have each box monitor the inserts being done. A simple “statistic” in this way can be to keep a boolean indicating whether the inserts done on the box have been “appends” only, considering a given (sort) order. If so, the divider can be set to retain very little slack space, which means for instance in a linear insertion case that the site load is high, comparing to the normal half min/max 50% slack space chunk function. If a box on a site chunked with no slack space has a new value inserted, this cannot be an append in sorted order
any more (due to the chunk with no slack space) and hence, the append only flag is not set, so a regular 50% slack space chunk can be performed.

### 6.4.2 Trail Policies

During simulation, it quickly becomes clear that there are extensions possible to the Armada model that allow for more precise redirections. Often these extensions can be a part of the standard amount of trails being exchanged during a chunk operation.

**Vanilla Armada** Follow the Armada model unconditionally. Each site contains boxes, which have a predecessor trail, the self step, and the successor steps. This bare-bone approach forces each agent to go around in the Armada using the absolute minimum of available data.

**Sibling Steps** Upon each chunk operation, add sibling information to the sites involved. This is a small optimisation on top of the vanilla Armada model, as this adds extra trails which allows successors of a node to redirect to each other without having to contact their parent box. In practice, this means that the newly created site gets an extra pointer to its sibling, because its sibling already has this pointer via its parent, which is hosted on the same site. However, for the new site this also holds (there is a pointer to the parent box/site) and hence adding the sibling trail does not help much here: the new site can send the agent to the original site, which knows what the specific box is, which is irrelevant for the navigational structure. Therefore, this policy only makes sense when both new boxes are stored on a new site.

**Agent Hinting** Agents that are being redirected to an inactive box (hence requiring another redirect), hint the site that offered an out of date redirect with the updated (more specific) trail. The site can use that trail the next time an agents visits to possibly direct more accurately. The possibilities here are numerous. Agents can hint only for successor steps (only optimising the current sub-tree) or for predecessor steps (which quite often trigger another redirect) as well. Agents can only perform one level e.g. A to B, B to C, C to D, only adding D to A, or D to B, or both/all levels. Rationale here needs to be found with respect to the association trees.
6.4.3 Agent Policies

The Agent in an Armada simulation is the entity that performs most of the work. It basically deals with the entire traversing through the system, by means of following redirects. On successive queries, the agent has a number of options to try and minimise the amount of hops taken for each query. The baseline approach for an Agent is the naive strategy of the Lazy Policy.

**Lazy** A Lazy agent starts each query at the same site, which is the only site it knows, the origin. Obviously this stresses the origin with a magnificent hit-count, but allows the origin to exercise in redirecting to the right site at once (e.g. in combination with Agent Hinting Trail policy).

**Random** This type of agent picks a random site from the cluster for each query. It uses no knowledge whatsoever, but truly randomises to spread the load over the cluster. This avoids the origin being a hot spot, via a technique that could be employed by e.g. DNS load balancing. This approach stresses each site in the cluster for its ability to send the agent in the right direction.

**Sticky** Sticky agents stick to the last site they have touched, for their next query. This strategy is in particular useful when doing a linear insertion, as it most of the time yields in a hopcount = 0. Sticky is a very cheap policy that tries to do slightly better by not disregarding the last known state of the Armada.

**Cache** Smart agents cache the lineage trails they see when traversing the Armada, and use that cache prior contacting a site to make an educated guess what would be the most appropriate site to contact, e.g. the site closest to the target. Obviously, out of date cached trails can be thrown away when being encountered to reduce the search space. A caching client can ultimately get a hopcount close to 0, as its cached trails represent the part of the Armada it is interested in. However, this is an ego-centric policy of which no other agents benefit. It is not realistic to have all trails for the entire Armada cached, as this may be a large amount. This large amount may not be an issue memory wise, but it will be a performance issue given that the search space increases. Hence, the agent needs to define a policy for itself that defines which trails need to be kept, with a limited number of cache buckets.
6.4.4 Site Policies

A site in the Armada hosts boxes. The model does not limit a site to host just a single box. While hosting multiple (active) boxes is not difficult, the capacity of a site is no longer subject to the utilisation of a single box. In particular the process of finding a new site to host a box as result of a chunk operation is affected by the decision whether sites with available capacity can be reused or not.

**Single Box** Each site contains at most one box. If there are no more available sites, the Armada cannot grow any further, even though some sites may be hardly full storage wise.

**Reuse Free** Sites are used based on their available storage space. From the pool of available sites, the site site with the most available storage space is used, assuming all sites have an equal storage space.

6.5 Data Sets

The shape of an Armada tree is influenced by the data it contains and the order in which it was inserted. To experiment with different tree shapes, and to see the effect of them on the various metrics previously defined, we used the following carefully crafted workloads. For each workload we used the same value range starting at 0, ending at some predefined positive number. By doing this, the sets can be used to query the other sets without getting an artificial skew because of a range mismatch. This avoids either having a high skew on the edge node because all high values are mapped onto it, or having a skew on a range of nodes because the other nodes cover values not in the value range of the query set.

**Linear** The linear set is a simple ascending counter with regular gaps to fill up to the desired value range. Since no duplicates are allowed, each value appears at most once in the Armada. The gaps between the values are equal, and hence do not affect chunking decisions due to the introduction of skew.

**Random** A randomised list of values. Duplicates are forced to occur during the generation process. 10% of duplicates are generated and gaps are likely to occur. The duplicate values are randomly spread throughout the set.
6.6 DATA LOADING

Uniform  Again a randomised list of values, but using a perfect even distribution. While gaps are still possible, duplicates are not allowed. The random order of the values, causes unlike the Linear set to have unordered insertion of values.

Zipf  Another random input based set, but with a value probability following a Zipf distribution. In this set, typically a small amount of values are very popular, and likely to occur more than once, while other values hardly occur, if at all. This distribution clearly is very skewed. In the generated set we chose to have the popular items to be those with a low value.

Real World  A set of integer values extracted from the entire MonetDB/SQL test set. Skew, duplicates and gaps are occurring here, as many tests use the same values, very close values and entirely different ranges. This set does not have the fixed range the other sets use, because the real world values simply are fixed due to being real. This set is hence only of limited use in comparisons.

6.6 Data Loading

The previously described sets generate Armada trees with characteristic shapes when being inserted. Starting with the Linear set, the simple ascending nature causes the values to be inserted into the most recently created box, which is chunked once its capacity is exceeded. The lineage tree of this set eventually is a deep tree, where each time a chunk is performed, the last added box is chunked. Because per the chosen chunk policy one box is retained on the original site, and the other containing the overflow values on a new site, the resulting association tree is a simple chain of successive sites being attached to each other. See Figure 6.1.

Random and Uniform  The Random and Uniform sets result in a balanced lineage tree, see Figures 6.2 and 6.3. As values are scattered, the tree is built by approximately filling all of its active boxes evenly for the value range they cover. The association tree on the other hand shows a pattern where the older the site, the more offspring it has. Since all sites represent for the time being at most one active box, each site can overflow similar to why the lineage tree grows in a balanced manner. This means any site can get a new site association. Hence, the longer a site is around, the more site associations it gets due to chunks.
Figure 6.1: Loading the Linear set.
Figure 6.2: Loading the Random set.
Figure 6.3: Loading the Uniform set.
Obviously, the origin site has the most associations, with its direct offspring following in association count. However, not only its generation defines the number of associations, as also the age of a site is related. So typically what can be seen here is that each site has less associations to offspring itself, than its parent has, and within a generation every site has less offspring associations than the sites in his generation that are older. In total this gives a diagonal shape to the association tree.

**Zipf** In a Zipf distribution typically a few values are very popular and rare values are almost never used. The Zipf set we generated uses a Zipf probability for each value to occur. As random values are chosen, their chance of being put in the set is based on the Zipf chance of the value. This way the lower the value, the higher the chance it ends up in the set. The resulting set is skewed towards low values. This shows up in the lineage and association trees, see Figure 6.4. The chunk operator places the overflow values on a new site. Because small values occur more often, in the Zipf set, this means that more often a box needs to be chunked that is the non-overflow part of a previous chunk operation. This typically leads to a slight opposite of the Linear set, where the overflow part is constantly chunked. In the Zipf lineage tree, the boxes on the older sites are re-chunked over and over again, resulting in a deep, not very wide tree. Because there are also higher values, the tree is not as narrow as the lineage tree. For the association tree the re-chunking means that old sites have many connections. Because of this property the resulting association tree is very wide, and not so deep.

**Real World** The Real World set is based on data values harvested from the MonetDB/SQL test set and therefore not following any special pattern. The lineage and association trees show however that the set contains parts that follow the Linear set and parts that appear to have a Zipf or other high-skewed distribution, see Figure 6.5. This observation matches the nature of the set, which has many duplicate values as a result of slightly modified copied and pasted tests, as well as some linear sequence to just insert a sufficient amount of tuples with distinct values. In particularly interesting is the constant alternation between high and low values that occurs at almost every place in the set, except from a piece with a linear nature. The tail of this set shows an alternating pattern for low and high values that linearly increase, the high value much more than the lower value. In particular the hop count graphs that we discuss later on visualise this pattern very well.
CHAPTER 6. PERFORMANCE ANALYSIS

Figure 6.4: Loading the Zipf set.
Figure 6.5: Loading the Real World set.
Figure 6.6: Loading the Linear set.

Figure 6.7: Loading the Random set.
Agent Policies While the shapes of the lineage and association trees depend on the data set being loaded, the actual agent policy used does not affect the shape at all. Though, it influences the performance of the agent during the loading. There is not just one “best” policy, as efficiency in terms of a minimal amount of hops depends on the set in use. The Linear set typically has a “moving hot-spot” as it appends to the last added box. The Lazy policy here gets an increasing hop-count for every query after a chunk operation has been applied, see Figure 6.6. The policy that is well suited for this set is the Sticky policy, as it nicely “follows” the moving hot-spot, and needs at most one hop right after a chunk operation has taken place to jump to the new site. The Random policy performs better than the Lazy policy simply because it has a chance of accidentally picking a site closer to the moving hot-spot. As the policy is based on pure randomness the performance in terms of hop counts is on average half of the depth of the association tree. Lastly, the Caching policy performs on any set very well, simply because it has all trail information locally as soon as it has visited each location once. Because we assume an unlimited cache, this policy is always very well performing. Because of this we skip discussion of this policy for the other sets.

Random and Uniform The Random and Uniform sets cause an agent to be jumping back and forth between sites in the Armada, see Figures 6.7 and 6.8. Since this cannot be predicted, without a cache each guess is as wrong positioned as any other. On average, the Sticky policy does not perform any better than the Random policy for this reason. Interestingly, the Lazy policy is the best for these sets when inserting the data. The reason behind this is that because it always starts at the origin, it never has to jump back to the origin, as the other policies have to when they start in a wrong branch of the Armada tree. This makes the Lazy policy on average around half a hop shorter. This can be explained by occasional “luck” of the Random and Sticky policies when they happen to start close to the target.

Zipf The Zipf set, even though it is skewed, shows the same pattern as the Random and Uniform sets, see Figure 6.9. This is not so surprising considering the Zipf set is also a random set, but with a Zipf probability. Because the tree is less deep, the advantage the Lazy policy has is bigger, resulting in almost one hop better performance than the Random and Sticky policies.
Figure 6.8: Loading the Uniform set.

Figure 6.9: Loading the Zipf set.
6.7 Querying

So far we have only observed the number of hops taken per insert during the loading phase. A constructed Armada tree can be queried again, as part of normal querying operations on an (existing) database. To further study the agent policies, for each loaded set, we query it using all sets per agent policy. Within the sets, not the same values have to be used. This means that when the loading and querying use different sets, values can happen not to be found. Though, a matching (responsible) site has to be found in any case.

**Lazy Policy** Querying the Linear set using a lazy policy in general yields in many hops. Obviously when querying with the Linear set itself the number of hops necessary per query continuously increases as the values are found further away, deeper in the tree. The Random and Linear sets have stable hop counts that on average are close to around half of the association tree depth. This nicely demonstrates that both sets are truly random given that the average is in the middle of the value spectrum. The Zipf set is as stable as the other random sets, but has much less hops. This can be explained by the nature of the set, where lower values are much more likely to occur than higher ones. As a result the average value is not in the middle of the value spectrum, but below it. Eventually the Real World set shows a pattern where first around the same values at on average 32 hops are retrieved for around 1000 queries, then some queries that do not require any hops, followed by a peak leading to the maximum depth of the association tree. After around 1000 more queries close to the origin, the remaining 3000 queries of the set are all around 15 hops.

**Random Policy** When we switch from a lazy policy to a random agent policy, the number of hops taken per query are reduced by about 50% of the hops taken with the lazy policy. This can be easily explained, as the random policy on the linear set jumps into the tree, always in the right direction. In the same way that the random query sets on average target the middle of the value spectrum, the random agent policy starts querying on average in the middle of the association tree, which obviously cuts the maximum hop count in half. The Linear set using this policy shows random behaviour, but on average shows an increasing hop count to only half of the hop count necessary when using the lazy policy. Where the line in aforementioned policy was straight, in the Random policy it is slightly curved in a quadratic shape. This can be explained when considering the values being queried for. Since the Random policy on average positions the
Figure 6.10: Querying the Linear set.

Figure 6.11: Querying the Random set.
agent in the middle of the association tree, lower values can be quickly found by jumping back immediately in the lineage trail as far as required, yielding in at most two hops. However, when values are requested that are deeper down in the tree than the start position, a one by one hopping down to the target has to be performed, with a higher hop count. The random effect makes this happen gradually.

**Sticky Policy** The sticky policy achieves the same hop counts for the Random and Uniform sets, because the positioning of the agent for those sets is like the random policy since the values queried are in a random order and hence end up at random sites. Since the Zipf set favours lower values, with the sticky policy, the agent more often ends up at a site close to the origin, resulting in a bigger chance than the random policy that the agent has to hop one by one away from the origin. The Real World set shows no peak using the sticky policy, as this peak consists of a linear sequence, which the sticky policy is able to efficiently follow. For the other parts of the set, the sticky policy results in higher hop counts, because the previous value is far way from the next.

**Cache Policy** The cache policy obviously has very low hop counts, simply because it can position the agent very well for every query already starting once it has learned a part of the tree.

**Random Set** The Random set has much smaller hop counts compared to the Linear set, see Figure 6.11. This is due to the association tree depth of the Random set being much smaller as a result of better tree balancing caused by the random value insertions. Like before, the Random, Uniform and Zipf sets show a steady average hop count over all queries for the lazy, random and sticky agent policies. However, unlike with the Linear set, the lazy policy is the most efficient in terms of hop counts here. This difference of on average half a hop can be explained by the random positioning of the agent no longer being a jump in the right direction. The association trees for the Random, Uniform and Zipf sets are wide, and hence a random jump has a high chance of ending up in a wrong branch of the tree. The latter requires a jump back to the origin, which is the position of the lazy policy. Therefore the latter policy is on average more efficient on these sets. The Linear set, when queried on the Random set, shows different behaviour though under different policies. The sticky policy, as expected, allows to reduce the hop count during querying quite dramatically. The association tree built has due to the fragmentation function the property
of sorted order, allowing the sticky policy to need at most one hop on a linear query sequence. Hence it again removes the peak from the Real World set.

**Uniform Set** When querying the Uniform set with the other sets using the four agent policies, a slight variation on the loaded Random set is the result, see Figure 6.12. Since the Random set only has some duplicates, which the Uniform set does not have, this is not surprising. There are no noteworthy differences to be found, and the same remarks as for the Random set hold.

**Zipf Set** Because the Zipf set has an association tree with a smaller depth than the Random and Uniform sets, there are less hops possible in total. As expected, the Uniform and Random sets are again steady, but close to the Zipf query hop counts for all policies, see Figure 6.13. Because the depth of the tree is smaller, the positioning error of the agent is less punishing, hence in total less hops need to be taken. The higher values that are not in the Zipf set, are simply not found, but the responsible site is found earlier because of the smaller depth. Because the Zipf set is a random set, the lazy policy performs slightly better as with the Random and Uniform load sets. The Real World set shows a different pattern, since the lower values are more fine grained fragmented, and successive queries need different sites now.

**Real World Set** The loaded Real World set has a very wide association tree with a small depth, except for one very deep branch. A random agent policy for this reason has an effect of on average one hop on top of the lazy policy because of the jump back to the origin, see Figure 6.14. Because the Real World set has a value range that is much smaller than the other sets, querying it with those quickly yields in out of scope value retrieval. The effect of this is that the one box that is responsible for the last range that reaches to theoretical infinity is queried for all these values. This can very well be observed through the Linear set using the lazy policy. After around 1700 queries, the hop count does not change any more, indicating the value range has been exceeded. The Linear set also shows that most of the lower values are found on the same depth in the association tree, most probably on the same site. In between a peak is found where the single deep branch is being followed. Obviously, the sticky agent policy effectively removes most of the hops for the Linear set here as the last visited site is in most cases the target for the next query. The Real World set when being queried shows how the values are being distributed over the set. The peak that the Linear set encounters is a result of the values at the end of
Figure 6.12: Querying the Uniform set.

Figure 6.13: Querying the Zipf set.
Figure 6.14: Querying the Real World set.
the Real World set. Note that the hop counts for the Real World set appear to be not as high as the peak of the Linear set. This is an effect of the moving averages being compared, where the Real World set apparently has alternating values causing high and low hop counts. The sticky agent policy helps to reduce the hop counts in the small linear part of the Real World set.

**Cache Policy** From the loading and querying simulations we can conclude that there is no single agent policy that suits best for all cases. Without any help from the Armada cluster, the lazy policy achieves the lowest hop counts for any random based set, ignoring the cache policy. The sticky policy only performs well on a set that makes the agent often visit the same site in succession. This typically happens in a linear sequence or stable value case. The sticky policy can be considered to be a limited cache policy. It stores at most one location, but does not use the information present therein and always unconditionally returns to this location stored in the “cache”.

The cache policy which we have mostly ignored before, outperforms any other policy by far. Its superior low hop counts are mainly due to the unlimited amount of cache slots which eventually allow to collect all trails available in the entire Armada. Mainly because of this unrealistically high (and theoretically unbounded) storage capacity, this policy in its current form is considered to be artificial and only feasible in a hypothetical world. The more trails are stored, the longer the time it takes to search through these trails. Since trails are only appended, this just makes the cache lookup slower and slower over time. The problem is made worse given that each trail has to be searched step by step to find a possible best match from the cache. However, its supreme performance win cannot be ignored. To be able to understand this performance and possibly approach it with a much more realistic policy, we have to look in more detail into the association tree and in particular where most of our hops go.

The Random sets are a good starting point for this performance quest. They show a very stable average of hop counts for all policies, where the lazy policy performs slightly better. The reason for this, is the price one has to pay for an association tree branch mis-prediction, which leads to a jump back – in the worst case to the origin. Such jump back to the origin, the lazy policy never has to make, since it always starts there. The cache policy performs so well on the same set, simply because it hardly mis-predicts. Because it considers its own cache, it always knows the origin, resulting in an equal to lazy performance in the worst case. However, if there is a cache item for the right branch, the cache policy can use it, jumping ahead in the right direction. The more trails cached,
the more precise the cache policy becomes, which eventually means that the chosen site for a query is immediately the right one.

6.8 Cache Policies

Our random agent policy simply does not take any branches into account. If it ends up in the right branch, then this is pure luck. The sticky policy only ends up in the right branch if the workload allows for this, as mentioned before. The cache policy gets most of its performance from jumping in the right branch of the tree. Hence, an agent would greatly benefit from having a cache which contains a number of trails for separate branches, which are taken as starting point. Experiments are necessary to show the trade-off of storing those trails against the gained performance. A brute force policy to just store the last $x$ different trails would help, but probably needlessly store a lot of duplicate data. As each trail includes the full parent trail most of the information may overlap. It would be interesting to try and detect this overlap, and to store the trail that has the most depth. Problem here is to decide when it is or is not paying off to discard a previous trail in favour of a newer one. The common part of both trails may be small, and hence resulting in loss of branch information.

**Cache Metric** A metric that we can use here is the length of the trails after their common part starting from the origin. Consider Figure 6.15 depicting three situations where two trails intersect. In the figure, only the sites referenced in the trails are depicted. This equals the association tree, and hence can have a situation as in Figure 6.15(a) where trail $A \in B$. Obviously, for
this situation, trail \( B \) can be chosen without losing any information, as we can reach the same sites as before. As a metric, for this situation we can define that by replacing \( B \) with \( A \), we reduce the possible hops we have to take for any query at maximum by 2 hops. At the same time, we do not add any additional hops in the worst case scenario, as all sites from \( A \) are contained in \( B \). Figure 6.15(b) on the other hand shows trail \( B \) which is much more specific than \( A \), but does not fully contain \( A \). In the depicted case, it may be evident that the loss of discarding trail \( A \) does not outweigh the win of storing trail \( B \). The to be discarded site from \( A \) can be reached via \( B \) by stepping from the last site in the common part of both trails. In terms of hops, this case reduces the number of hops at maximum by 4, while it increases them at maximum by one.

Lastly Figure 6.15(c) shows trail \( A \) and \( B \) where the overlap is partial and the benefit of either over the other is not obviously clear. Applying our metric, the maximum number of hops is decreased by 4, increased with 3. Though the loss of either branch is substantial. It may be clear that when the cache slots are all filled, an algorithm to find which trail should be dropped — if any — needs to be run. From the metric used before, we can define the benefit ratio as the maximum number of reduced hops divided by the maximum number of added hops. This ratio has a value greater than 1 for trail \( A \) against \( B \) if \( B \) reduces more hops, than those lost by removing \( A \). When the ratio is smaller than 1, trail \( A \) is favourable for the system as a whole. When there is no loss such as in Figure 6.15(a), the ratio cannot be computed. This is not a problem, as in such case \( A \) can always be replaced by \( B \). A cache insertion algorithm that makes use of this ratio is depicted in Algorithms 6.1 and 6.2.

Figure 6.16 depicts the final state of the cache trails after a query run when the cache allows for 5 trails. While 5 trails are insufficient for each tree to reach every leaf node, the figure clearly points out that the available trails are not positioned in the most effective locations. In particular, a lot of redundancy is contained in the used trails.

**Benefit Ratio** From Figure 6.17 the average number of hops taken per query for various cache sizes can be read. From the Random set in Figure 6.17(b) it immediately shows up that the performance for 1, 2 and 3 cache trails is roughly the same. The next performance wise jump is made by 4 and 5 cache trails. This behaviour can be explained by the graph from Figure 6.16(b). Obviously the leftmost (and longest) trail is always in the cache, as it is the most beneficial trail according to the benefit ratio. The algorithm adds the leftmost two next trails to the cache first, resulting in an almost equal performance. The trails have a very
Algorithm 6.1 Cache insertion algorithm.

\[
\text{candidate} \leftarrow \emptyset \\
\text{maxratio} \leftarrow 1 \\
\text{for each cache } t \text{ do} \\
\quad \text{ratio} \leftarrow \text{benefit(newtrail}, t) \\
\quad \text{if not ratio then} \\
\quad \quad \text{replace}(t, \text{newtrail}) \\
\quad \quad \text{break} \\
\quad \text{else if ratio} > \text{maxratio then} \\
\quad \quad \text{maxratio} \leftarrow \text{ratio} \\
\quad \quad \text{candidate} \leftarrow t \\
\quad \end{if} \\
\text{end for} \\
\text{if candidate} \neq \emptyset \text{ then} \\
\quad \text{replace(cache, candidate, newtrail)} \\
\text{end if}
\]

Algorithm 6.2 Implementation of the benefit function.

\[
i \leftarrow 1 \\
\text{while } A_i \neq \emptyset \text{ do} \\
\quad \text{if } A_i \neq B_i \text{ then} \\
\quad \quad \text{break} \\
\quad i \leftarrow i + 1 \\
\text{end while} \\
\text{if not } A_i \text{ and not } B_i \text{ then} \\
\quad \text{return 0} \\
\text{end if} \\
\text{if not } A_i \text{ then} \\
\quad \text{return } \emptyset \\
\text{end if} \\
\text{len}A = \text{len}(A) - i \\
\text{len}B = \text{len}(B) - i \\
\text{return len}B - \text{len}A
6.8. CACHE POLICIES

Figure 6.16: Final state of first generation cache trails after querying.
CHAPTER 6. PERFORMANCE ANALYSIS

Figure 6.17: First generation cache with hops for different cache sizes.

(a) Uniform
(b) Random
(c) Zipf
(d) Real World
large common part, but are selected by the algorithm because they have a larger benefit ratio than other (shorter) trails. The next added trail is the rightmost trail from the figure, which explains the performance gap between 3 and 4 trails in the cache. Adding the fourth trail from the left in case of 5 trails in the cache results in the little performance win between 4 and 5 trails. It is obvious that the chosen trails to cache are quite inefficient for the total picture. Instead a cache utilisation such as chosen for the Zipf set in Figure 6.16(c) is much more efficient. The hops diagram for this set in Figure 6.17(c) shows an improving performance per added cache trail. The algorithm chooses better in this set because it operates on such a wide association tree, where the deepest leaves are all in a separate branch of the tree. Since the depth of most of these are equal, the algorithm does not consider a trail which is almost contained in another already cached one as better than one from another branch as happened in Figure 6.16(b).

It is obvious that the algorithm in its current form is not choosing the ideal trails for its cache. This is most prominently shown by Figure 6.16(d) where three trails are used for the deepest branch. The two trails used for the single site splits off of the main branch actually add very little (one hop) extra knowledge given the longest trail, considering other branches that are not in the cache, but could have been cached instead. We can conclude that with the current algorithm, the amount of overlap with other trails in the cache is ignored. This results in trails that are very close to other cached trails to be added in favour of other cached trails which have a smaller benefit ratio. The trail that is added to the cache as a result simply is a loss in the total picture of the cache coverage. The benefit ratio algorithm needs to be refined to take the overall benefit for the cache as a whole into account when replacing a cached trail for another.

Looking at the cache trail trees from Figure 6.16, it is persuading to think that siblings are directly reachable from the trails themselves. However, this is not true, as the vanilla Armada model defines not to include sibling information in the sibling trails themselves. They only have the predecessor trail, hence this information is not available. Also, because the trails are depicted on an association tree, it is hard to see that there is a temporal relation between all direct successors of the same step. This means that even if sibling information was passed onto the successors, this still would not include the full set of siblings, but only the siblings that were involved in the same chunk operation. For this reason, the cache trails as depicted in the figures, describe the full “span” of
the trails. Any optimisations to the cache replacement policy need to take into account that only this information is contained in each cache trail.

**Algorithm 6.3** The `cacheappend` function.

**Require:** \( new \neq \emptyset \)

\[
\text{cacheaddifnotcontained}(new) \quad \text{if } \text{size}(\text{cache}) > \text{MAXCACHESIZE} \quad \text{then}
\]

\[
\quad \text{removeleastfromcache()} \quad \text{end if}
\]

**Algorithm 6.4** The `cacheaddifnotcontained` function.

\[
\text{for each } \text{cache } t \quad \text{do}
\]

\[
\quad \text{if } t \subseteq new \quad \text{then}
\]

\[
\quad \quad \text{cachereplace}(t, new) \quad \text{return}
\]

\[
\quad \text{else if } t = new \quad \text{then}
\]

\[
\quad \quad \text{return}
\]

\[
\quad \text{else if } new \subseteq t \quad \text{then}
\]

\[
\quad \quad \text{return}
\]

\[
\quad \text{end if}
\]

\[
\quad \text{cacheadd}(new)
\]

\[
\text{end for}
\]

**Improved Cache**  Algorithms 6.3, 6.4, 6.5 and 6.6 depict an improved version of the cache replacement algorithm. Instead of comparing a new trail to each of the trails in the cache separately, the new trail is compared to the other trails in the cache as if it were part of the cache. This leads to removal of the trail in the cache that results in the least loss in terms of benefit. The essential difference between the first cache replacement algorithm and this algorithm is that the benefit is no longer calculated based solely on the trail itself. The benefit is now calculated as the number of hops that are reduced considering all other cache trails. As a result, those sites (hops) that are in common with other trails do not count for the benefit any more. For this, the longest part in common with the other trails in the cache has to be determined, to calculate how many sites are uniquely added to the list of known sites by the trail.
Algorithm 6.5 The removeleastfromcache function.

\[ \min_b \leftarrow \infty \]
\[
\text{for each cache } t \text{ do}
\]
\[
b \leftarrow \text{length}(t) - \text{commonlength}(t)
\]
\[
\text{if } b < \min_b \text{ then}
\]
\[
\min_b \leftarrow b
\]
\[
cand \leftarrow t
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
cacheremove(cand)
\]

Algorithm 6.6 The commonlength function.

Require: \( t_a \neq \emptyset \)

\[ \max_s \leftarrow 0 \]
\[
\text{for each cache } t_c \text{ do}
\]
\[
\text{if } t_a = t_c \text{ then}
\]
\[
\text{continue}
\]
\[
\text{end if}
\]
\[
s \leftarrow \text{commonpart}(t_c, t_a)
\]
\[
\text{if } s > \max_s \text{ then}
\]
\[
\max_s \leftarrow s
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{return } \max_s
\]
CHAPTER 6. PERFORMANCE ANALYSIS

The cache replacement algorithm works by requiring one extra slot in the cache to store a new trail. To ease the algorithm, a new trail is only added if it is not already in the cache, or superseded by a trail from the cache. Also, when a trail is found that supersedes a trail from the cache, it is used as replacement for the superseded cache trail immediately. This way, trails added to the cache are always trails that address a site which is not addressed by all others.

If the number of trails in the cache exceeds the maximum number of allowed trails, the cache replacement algorithm is run to evict one trail from the cache to be removed. The trail to be removed is chosen based on the afore described benefit function. For each trail in the cache, the benefit is calculated, and the trail with the smallest benefit is chosen to be removed. Figure 6.18 depicts a situation of three trails. On the right of the picture the benefit calculation for each of the trails is shown by taking the total length subtracted by the length of the part of the trail in common. It is obvious that the trail with benefit 1 would be evicted in favour of the other two with both a benefit of 2. Note that after removing this trail, the benefits of the other two trails have to be recalculated because the common parts may have changed, as is the case for the longest trail in the figure.

It is to be expected that there is not always a single trail that has the lowest benefit. There may very well be multiple trails matching. The algorithm removes the oldest trail in such case, as it is based on a cache that is implemented as a linked list, where new trails are appended to the tail. Hence, the first trail found when traversing the list is the oldest. The rationale for doing this is that the more recently added trails may better reflect the current query behaviour.

**Gradual Improvement** From Figure 6.19 it can be deduced that the second generation cache trails replacement policy reaches a better final state of cached trails. Compared to Figure 6.16 more branches are represented in the cache, and trails with large overlapping parts are no longer present. This is entirely conform the objectives of the improved replacement algorithm. The performance wise results are depicted in Figure 6.20. When compared to Figure 6.17 we observe that with 10 cache trails the second generation has an average per-
Figure 6.19: Final state of second generation cache trails after querying.
Figure 6.20: Second generation cache with hops for different cache sizes.
formance around half a hop per query, whereas the first generation delivered an average performance of around 1 hop per query for the random sets. Unlike the first generation, the second generation shows a more gradual performance improvement for the Random and Uniform sets per added trail to the cache size. The Real World set in general has an improved performance with the second generation cache replacement policy. However, for the cache size of one trail, a peak can be seen for the second generation that is 1 hop higher than in the first generation. The peak is caused by a linear query pattern, which values are found in the long trail of the Real World association tree. The hops are caused by the values that are stored in the start of the long trail. Once the values address a site that has a longer trail than those stored in the cache, it is being added, and hence the hops immediately drop to a minimum, as successive values at need one hop every time the site capacity has exceeded, like in the linear case is happening. The single hop is explained by the benefit function disregarding the common part of trails in the second generation. For this reason it takes one site longer before the benefit is higher since the root node is always shared with other trails.

Wide Coverage When observing the trail transitions in the cache using the second generation cache replacement policy, situations similar to Figure 6.21 appeared to be common practice. What happens is that the dashed trail is added to the cache, while the dotted trail is the oldest. All trails have an equal benefit according to the second generation cache replacement policy, and hence its default method of removing the oldest trail results in the two longest trail to be kept, and the dotted trail to be removed. While in absolute terms all three trails add the same to the system as whole, the dotted trail can be more useful to the system considering heuristics. First sites up in the tree have a higher chance of having many (direct and indirect) successors over sites lower in the tree. For this reason, sites up in the tree possibly help for more queries, due to their potential coverage. Second, sites up in the tree have a higher chance of addressing a not yet addressed branch of the tree. As we have discussed previously, getting distinguishing branches in the cache increases the theoretical
coverage on the tree. Because of these heuristic hypothesises we refined the second algorithm for the cases where an eviction decision is made for trails with an equal benefit. Instead of choosing the oldest in the cache, an evaluation is made aimed at retaining the trails targeting sites higher in the tree.

A few variants are possible to achieve the aforementioned goal. First the algorithm could consider the common part of trails and prefer trails with a lower common part as this means they distinguish themselves better from other trails, than those with a large common part. Naturally, this favours short trails, as they have a small common part given that the benefit of the trails being compared to is the same. Short trails obviously address sites higher up in the tree. Secondly a relative benefit based on the depth of the trail could be calculated. In the given example from Figure 6.21 the dotted trail would have a benefit of 1 weighted over a common trail of length 1. The dashed and normal trail have a benefit of 1 over 4, resulting in one fourth. Alternatively, the benefit is made relative to the depth in the tree where the non common part of the trail starts. Effectively, the same node is selected to be retained by both variants. Tests have indicated no performance improvements compared to the previous generation. Since this approach is a micro optimisation that only affects a few cases, this is not a surprising outcome.

6.9 LRU Cache

Until now we have only considered caches based on the Armada lineage trail structure. As a result our caches were focused on keeping the optimal trails for the tree structure. The behaviour of the clients per their queries has been ignored. Patterns of interest in only a specific part of the Armada are not recognised by the till now implemented cache strategies. From observations of the cache transitions, strategies that adapt to the current workload may improve performance in cases where sibling nodes are alternatively selected to be included in the cache. Those cases clearly indicate a localised query interest, but cache trails are assigned for those sibling nodes, since other (longer) trails are considered to be more valuable for the entire tree and any possible query.

We implemented a simple LRU cache strategy for Armada trails. Without assuming any knowledge of trails, the LRU simply stores the trails that point to the box for each query. Trails that already exist in the cache are removed and reinserted. When the cache overflows, the trail that is least recently inserted is removed.
Figure 6.22: Final state of first generation LRU cache trails after querying.
Figure 6.23: First generation LRU cache with hops for different cache sizes.
Figure 6.23 shows the average hopcount graphs for this LRU cache replacement policy. Again the graph for the Linear set is omitted, as it is a straight line close to zero hops on average. The LRU is obviously very well capable of “following” the linear case. The random sets show a steady and slightly worse performance compared to the first non-LRU cache generation. In all cases adding a cache trail results in a reduction of the average hop counts. More interesting is the Real World set. It shows for more than one cache trail that the entire tail starting from around the 1900th query can be reached in close to zero hops. This is very well explained because that entire tail consists of an alternating low/high value sequence. Starting from two trails, the LRU can keep the low and the high value sites in its cache and hence serve the query load very well. Starting from 3 trails, the entire Real World set can be done with on average less than 1 hop, which is quite effective.

**Overlap Awareness** Because the LRU cache does not take anything into account from the Armada model, but just stores trails, the chosen trails are not optimal considering the entire path up to the root is used by the agent when using the cache. From Figure 6.22 this particular problem can be observed, as only 4 trails can be seen, while there are 5 in the cache. In all four cases there is a trail that is fully contained in another in the cache, hence invisible in the figures. Like we do in the non-LRU cache policies, the LRU could be extended to recognise when trails are contained in each other and then move the largest trail. When a site is found in the predecessors of a trail that matches, the trail for that site is added to the LRU. Hence making the LRU aware of this, can improve the effectiveness of the cache trails.

Experiments show that applying above strategy indeed improves the effectiveness of the cache trails. For all sets, a cache size of 50 is now sufficient to reach a near zero hops performance where the previous generation was not able to achieve that. Compared to the third generation non-LRU cache, the second generation LRU cache is roughly half a hop worse in performance for the random sets. The performance on the Real World set has not improved over the previous generation, but still outperforms non-LRU caches.

**Cache Performance Comparisons** We have considered the agent’s query behaviour using a least recently used scheme, where the last used trails are kept in the cache. This cache differs from the non-LRU caches in that it is fully driven by the query load, possibly adapting to the query points of interest. However, this strategy on average performs worse than non-LRU, trail logic based caches
for all but one set. The cache replacement based on LRU may be too sensitive to outliers and hence discard trails too often. An alternative here to adapt to the query workload, is to use the usage rate of trails in the cache. This usage could be either defined as the number of times the trail is used, or as the number of times the sites contained in the trail are used. However, for such a strategy it is necessary to keep the usage count for every trail or site, since otherwise trails will never be added to the cache as they are never more used than those in the cache. Keeping usage information about all possible trails or sites is as roughly the same as having an unlimited cache, and hence not a viable solution.

Summary

The Armada agents have to locate data in the system. They do so by following lineage trail information, available on every site. An Armada that has grown large involves many sites, which all potentially can contain the data an agent is looking for. While network connections are expensive, time wise, the more an agent needs to hop around, the worse the performance.

Four agent policies have been studied to see the effect of them on five data sets. While different sets result in trees of different depths, the hops taken by an agent are affected by this depth. While some policies work reasonably well on some sets, only the policy where the agent caches trails for later reuse reaches a very good performance for all sets.

Since an unlimited cache is a rather unlimited resource claim, we conducted several experiments with limited cache sizes. By revising our cache algorithms, based on characteristics of Armada lineage trails, we reached an acceptable amount of hops per query for a limited amount of cache. This result indicates that the active Armada client is viable in terms of costs with respect to the autonomy and distribution it allows.