Optimising Client Accesses within Armada

Fabian Groffen Martin Kersten Stefan Manegold
Centrum Wiskunde & Informatica
Science Park 123
1098 XG Amsterdam, The Netherlands
{fabian,mk,manegold}@cwi.nl

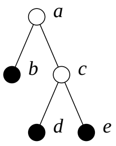
ABSTRACT

The Armada model describes how a distributed database system evolves, using multiple nodes that together form the database. In such system, posing a query involves continuously locating the right node until sufficient data to answer the query has been found.

Locating a node involves making a connection to such node. Since making a connection is expensive in time, avoiding to do so where possible, pays off in the total query execution time. In this short paper, we give an extended summary of our work on cutting down the number of made connections per query in an Armada system.

1. INTRODUCTION

In previous work, we presented Armada, a reference model for an evolving database system [1]. Due to space reasons we have to refer the reader to our paper for details. In short, the Armada model builds a "lineage" tree representing the growth in history of a distributed database that is divided into two new pieces on every step. The divisions contain enough information to tell what data went where, allowing a client in the system to convergence to its target from any node in the system. However, instead of reaching the data directly after a single catalog lookup, multiple steps can be necessary to reach the data being looked for.



The small tree on the right depicts a little Armada lineage tree. Clients typically arrive at any one of the leafs of the tree, and start their query from there. The structure of the Armada model allows for each node in the tree to tell how to get nearer to a given value, if the value is not supposed to be stored locally. A node can redirect a client if it does not hold the requested value itself, either one step down in the tree, or to any node up on the path from itself towards the root. Hence, a client that sends a query to node b in the small tree, asking for a value stored on node e, gets a redirect from b to a. a in turn, redirects to c, which eventually redirects the client to node e, which can answer the query. The other way around, if the client starts at e and asks for a value from b, e redirects the client to a, and then reaches via another redirect b. Obviously, depending on where the client starts its query, the number of redirects necessary to find the answer can be large or small.

PostgreSQL		MySQL		MonetDB	
0.134	21.021	0.099	14.735	0.123	23.112

Table 1: Wall-clock times in seconds for performing 1000 queries over a single or multiple connections.

Armada is a model designed to facilitate the use of both replication and fragmentation. It supports administration of operations for both retrieval and evolution of data with a self-tuning flavour: due to the flexibility of the model, new systems can participate when necessary, old ones can leave, and the actual number of systems or location of data is hidden from users of the system. The Armada administration allows for localisation of data without need for a central entity that becomes a bottleneck and hot-spot in busy systems.

In this paper we study the process of locating the right node by following directions in 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 additional redirects. Hence we look for strategies to minimise the required redirects in the Armada model.

That redirects are indeed expensive, is shown by Table 1. It shows a small experiment we conducted on an OpenSolaris AMD Athlon64 3800+ system running three Open Source database systems. For each database system the left value represents 1000 simple queries performed over a single connection, while the right value represents the number of seconds necessary to perform the same 1000 queries but each over a new connection to the database system. From the table it can be concluded that separate connections 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 one has to make during the query process, since this takes a substantial amount of time.

Throughout this paper we frequent the terms *agent*, *site* and *box*. When we refer to an *agent*, we refer to the entity in the system that interacts with the data nodes in the Armada system. A *site* is a data node, capable of storing boxes in the Armada system. A *box* is a logical block of data hosted on a site, a product of dividing the database.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WDDDM '09, March 31, 2009, Nuremberg, Germany
Copyright 2009 ACM 978-1-60558-462-1/09/03 ...$5.00.

2. EXPERIMENTAL ENVIRONMENT

In this extended abstract, we look at a limited set of environmental settings and their effect on the number of redirects that need to be followed. The key measurement we use is *hopcount*. It represents the number of steps taken by the agent for a query from the starting site to the site holding the active box that is responsible for the value being looked for. An agent that directly contacts a site which contains an active box that is responsible for the data value, has $hopcount = 0$ for that particular query. The bigger $avg(hopcount)$ becomes, the worse the seek performance of the Armada.

For simplicity, we assume that 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 able to store more data using a box. The *agent* in an Armada 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.

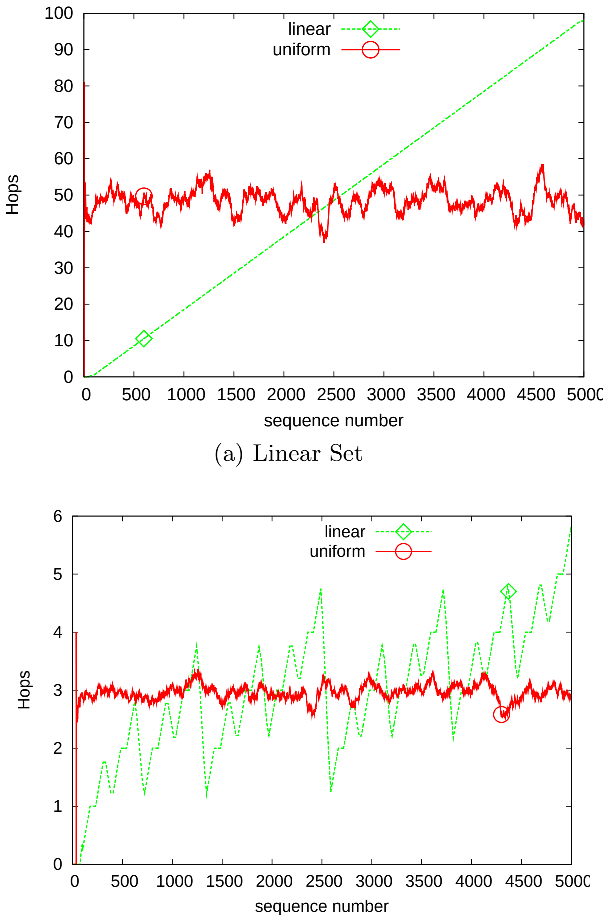
When using the *Lazy Policy*, an agent starts each query at the same site, which is the only site it knows, the origin. Obviously this stresses the origin with a magnificent hitcount, and takes the agent as many hops as there are on the shortest path from the root to the target box. An alternative to this naive approach is the Cache Policy.

Smart agents use the *Cache Policy* and 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 agent can ultimately get a hopcount close to 0, as its cached trails represent the part of the Armada it is interested in. However, it is not realistic to have all trails for the entire Armada cached, as this may be a continuously growing 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.

The shape of an Armada tree is influenced by the data it contains. To experiment with different tree shapes, and to see the effect of them, we used two 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.

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.

A randomised list of values is the *Uniform* set. It uses 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.



(a) Linear Set

(b) Uniform Set

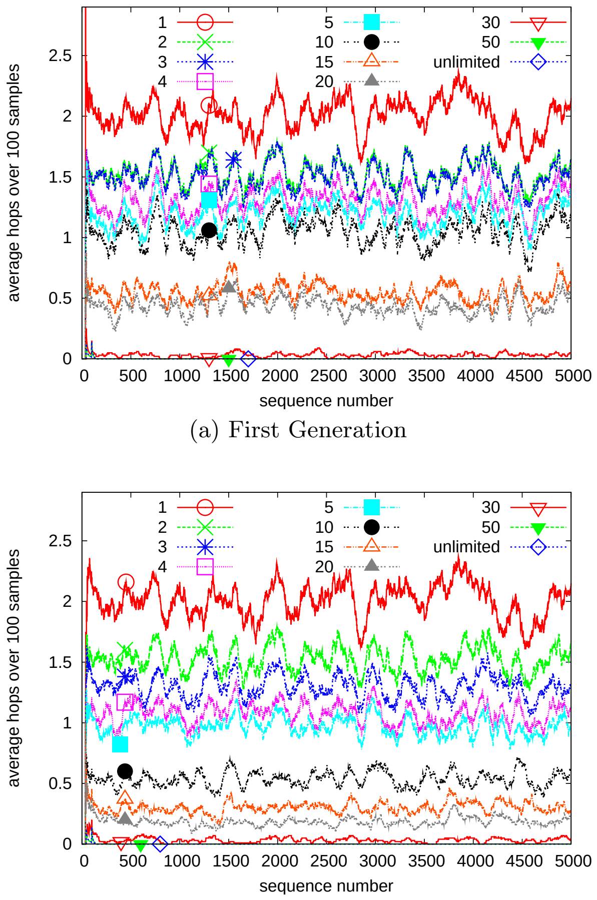
Figure 1: Querying using the Lazy policy.

3. QUERYING

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, as can be observed from Figure 1a. Querying with the uniform set on the other hand, nicely requests values which are located scattered over the entire tree. This results in on average a hopcount close to half of the tree depth.

The Uniform set has much smaller hop counts when queried with any set compared to the Linear set, as can be seen in Figure 1b. This is due to the depth of the tree generated by the Uniform set being much smaller as a result of better tree balancing caused by the random value insertions. Still, when querying with the Linear set, a slightly increasing hopcount is seen. This is due to the Linear set getting a larger reach all the time, while the Uniform set has the maximum reach from the start.

The cache policy on the other hand, has very low hop counts when using an unlimited cache, simply because it can position the agent very well for every query already starting once it has learnt a part of the tree.



(a) First Generation

(b) Second Generation

Figure 2: Querying the Uniform set using the Cache policy.

Cache Policy.

The cache policy, depicted in Figure 2a, 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 unlimited 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 Uniform set is a good starting point for this. The cache policy performs so well on this set, simply because it hardly mispredicts. 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.

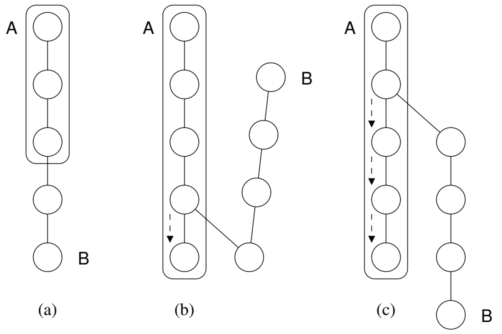


Figure 3: Association trial intersections.

Limited Cache.

When the cache size is reduced, no longer all trails can be kept, and trails have to be thrown out. The performance of the next query depends on which trail is removed, as it might affect how much the agent can start close to the actual target. A metric that we can use here is the length of the trails after their common part starting from the origin. Consider Figure 3 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 3(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 3(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 3(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 3(a), the ratio cannot be computed. This is not a problem, as in such case A can always be replaced by B.

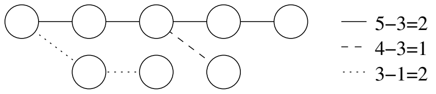


Figure 4: Benefit calculations for three trails.

First Replacement Algorithm.

From Figure 2(a) the average number of hops taken per query for various cache sizes can be read. It immediately shows up that the performance only slightly increases, with the obvious exception for a single trail cache. This behaviour can be explained by the longest trail always remaining in the cache, as it is the most beneficial trail according to the benefit ratio. The algorithm adds the longest trails to the cache first, often resulting in little to no performance improvement. The trails have a very large common part, but are selected by the algorithm because they have a larger benefit ratio than other (shorter) trails, while those could possibly be more useful when they address an entirely different branch. It is obvious that the chosen trails to cache are quite inefficient for the total picture.

It is obvious that the algorithm in its current form is not ideal. In particular the performance for 2 and 3 trails in the cache is equal, and indicates inefficient trail caching. In-depth analysis revealed that the current algorithm favours longer trails over shorter ones, even though the shorter ones address completely different branches. 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.

Second Replacement Algorithm.

In the replacement algorithm as depicted in Figure 2(b), 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 on solely 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.

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 4 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.

The effect of this cache replacement algorithm is clear given the two graphs of Figure 2. The second generation graph shows a better improvement per added cache trail, and eventually a lower hopcount for a number of trails. For example, while the first generation would require 15 to 20 cache trails to reach a half a hop performance, the second generation reaches that performance with 10 trails for the uniform set.

4. CONCLUSIONS

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.

Two agent policies have been studied to see the effect of them on two data sets. While different sets result in trees of different depths, the hops taken by an agent are affected by this depth. From the two policies we discussed, only the policy where the agent caches trails for later reuse reaches a 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 agent is viable in terms of costs with respect to the autonomy and distribution it allows.

5. REFERENCES

[1] F. Groffen, M. L. Kersten, and S. Manegold. Armada: a Reference Model for an Evolving Database System. In *Proceedings of Datenbanksysteme in Business, Technologie und Web*, Aachen, Germany, Mar. 2007.