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

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.
4 Automated Operations

4.1 Introduction

The Armada model tracks the evolutionary path of data within a cluster of machines. Where data is placed is precisely administered through lineage trails. Important in those trails is the use of functions that specify what part of the entire data is affected by the operation. A large part of the flexibility of Armada is due to the freedom of choice in what function to use for each operation. A part of Armada is the autonomy of sites to initiate operations to take place by themselves. This yields in the question how the functions can be applied without human intervention, such that the Armada can evolve itself where necessary. This kind of evolution where the system adapts itself is a form of self-adaptation from the self-managing area.

Already back in 1989, Self-Something aware databases were proposed, such as Cactis [36], a self-adaptive, concurrent implementation of an object-oriented database management system. The self-adaptiveness of Cactis is in the dynamic change of the physical organisation and order of update algorithms to reduce disk access. Many more self-* were to follow, all focused on optimising a local condition automatically without human help. Nowadays, self-healing refers to detecting and restarting failed or hung services [17]. SQL Anywhere [10] does database self-management, which means that the buffer pool size of the DBMS is increased and decreased based on feedback from the OS and paging faults. Commercial database management systems take the term self-management as the ability to avoid a performance nightmare. Instead of seeing spikes in the performance of the system, a graceful degradation is achieved by e.g. applying control theory, such as in [67].

Recently virtual machines have gained a lot popularity. They impose different behaviour characteristics due to the shared physical hardware. In [15] the inference caused by the virtual machines in a replica setting is discussed. The
self-managing nature here is to figure out what queries cause a lot of interference and deal with them either via complete relocation on another replica, or by limiting the resource quotas.

The work done in the self-* area mainly focuses on local tuning operations. Where other sites are involved, those sites are used to solve a local performance issue. In an Armada cluster, the self-adapting nature we are after goes beyond a single system and instead focuses on the entire cluster. What we’re looking for is more of the self-organising nature of multi-agent (MA) systems [73] in this respect. MA systems consist of a number of agents that independently try to work on the same target. Here, a maximisation of the utility of the whole system is preferred over the utility of an individual agent. In other words, one may have to suffer for all others, or the system at large to become better. Each site in an Armada can be seen as an agent, and as such the Armada itself as a MA system. The sites in an Armada work together on serving the work and storage loads of a DBMS. Maximising the efficiency of the whole Armada is achieved by both spreading data for performance, dividing data because of capacity, and replicating data for redundancy.

The first sections in this chapter deal with the operations available in Armada and the functions that can be used for them and their characteristics. The last sections dive into the self-managing aspect of the Armada and how the functions should be operated in such setting.

4.2 Functions and Operations

Distributed databases come either as replicated or fragmented variants. Proposals for combinations of the two are rare and most systems focus on either one of the two. Since replication is more an effort of coordinating machines, usually through an hierarchy [60, 19], not much can be said about the data operations for it. Fragmentation on the other hand is more challenging. Two types of fragmentation are distinguished, horizontal and vertical fragmentation. They refer to how the data relations are cut into pieces. In the case of horizontal fragmentation this means a row-wise cut, resulting in fragments containing full tuple rows. Taking the union of all horizontal fragments yields in the original relation. Vertical fragmentation cuts column-wise, with as result separate sets of columns as fragments. Obviously reconstruction here is done using a join between the column sets.

At the base of horizontal fragmentation are predicates that describe which tuples belong to which fragment. Traditional literature considers these predic-
ates to be a prerequisite for fragmentation [12]. Predicates are simple attribute conditions, and the process of determining how to fragment is driven by identified predicates based on application usage. The classical example uses the geographical location attribute with a condition that separates two regions of interest. This condition is taken from major application queries, such that the fragmentation is supportive to the system. The idea behind this is that when a fragmentation is chosen that results in consulting all fragments for each query, the fragmentation does not help the system, since networks are slow. This has improved, and possible gains from parallelism contrast this way of thinking.

When more attribute conditions are identified, a minterm predicate is used to identify the conjunction of conditions that are relevant. The minterm predicate takes simple predicates either in their normal or negated form, but only in conjunction. This means no combinations can be made which extend each other as in the boolean or logic. Thus:

\[ y = \bigwedge_{p_i \in \Phi} p_i^* \]

with \( p_i^* \) being either the normal or negated form of \( p_i \). Further the conditions may not contradict, hence \( y \neq \text{false} \). Resulting are the relevant predicates considering the identified important application queries.

In vertical fragmentation, columns are grouped in sets, which are like in the horizontal case identified by important applications’ queries. If an application needs attributes from more than one column set that application does not benefit from the fragmentation as the columns need to be joined again to reconstruct the original relation, again with slow networks in mind. To aid in this a tuple identifier like a primary key or row id needs to be present in each column set. This implicitly adds duplication of data when using vertical fragmentation. However, in today’s technological state, the benefits of performing e.g. selections on columns in parallel can be quite substantial.

Dropping Boxes All sites in an Armada are autonomous in their actions, but required to abide one simple rule. \textit{Data may never get lost by dropping it on purpose.} This restriction forbids any site to remove data without actually checking if it would make that data unavailable. Hence, within this contract, data \textit{can} be dropped, if and only if a clone of that data can be found, which can handle the same queries. This effectively keeps the data available in the Armada. When a site wants to drop a box, it has to find out if the data it drops can be found somewhere else in the Armada. The best way of doing so, is to examine the
local lineage trail. Clone operations in the parent trail indicate a replication of a superset of the data contained in the local box. Following such clone operation is necessary to determine if the clone is still available in the runtime state of the Armada, as it may have been combined with another box again, or simply be dropped or unavailable as well.

**Functions** In contrast to chunking functions, cloning functions are defined to have “overlap”. More specifically, the clone function puts no restriction on tuples, instead accepts every tuple, regardless which box it is on. Of course this is only from the “fragmentation”-like point of view. Additional actions need to be taken to get the behaviour of duplicating the data. The clone function is the opposite of being pair-wise disjoint: on updates it always also redirects to its partners in the clone contract. The clone function implementation includes the knowledge that a full replica of the data is available elsewhere. Due to this definition the implementation space for clone functions is limited to a single implementation of cloning behaviour.

Unlike chunk and clone functions, the combine functions are designed to reduce number of active boxes in the Armada. Where chunking and cloning facilitate growth, combining facilitates reduction by merging two or more boxes into one. By definition, this is done in such a way that duplicate data is reduced to a single copy. The resulting data is the merger of the data that is combined. The combine function takes the union of two or more other functions, and filters out duplicates from the data. Since the implementation of this function is a simple boolean $\text{OR}$ relation between the functions it combines, there is just one implementation, like with the clone function.

Opposite to the clone and combine operations, the functions to use with the chunk operation are numerous and diverse. All but one of those functions are horizontal fragmentation functions, simply because only horizontally the actual data is involved. Vertical fragmentation is based on schema rearrangement into separate sets of columns. In fully vertically fragmented database systems, such as MonetDB [7], further vertical fragmentation is impossible given that all relations are already fragmented to the column level. Hence, we focus primarily on horizontal fragmentation, when we deal with chunk functions in the rest of this chapter.
4.3 Chunk Functions

It is not uncommon for databases to have pieces of the data that are accessed more often than others. Such frequently accessed pieces of the data are referred to as the hot sets in the data. The ratio in which the hot sets are accessed more often than the rest can be very large, resulting in a typical hot spot in the dataset. This skewed access may simply stress a server and/or make it sluggish to do queries on data outside the hot spot. In a read-oriented setting cloning the hot spot would improve the performance. However, in a setting with many writes splitting up the hot spot over two or more servers has the benefit of not having to synchronise the two servers, while it still improves the general performance of the system. This goes under the assumption that both machines are at least equally fast and network costs are equal. The effectiveness is dependent on the chunk function chosen to effectuate the split.

When a server simply runs out of (disk) space, further storage of data is impossible and service degrades. Actions have to be taken to release such server by chunking a part of its data to another server. While for the problem at hand (capacity) no performance issues are of relevance, splitting can occur based on pure space-wise grounds.

4.3.1 Function Types

With capacity and performance as driving forces behind fragmentation in mind, we identified the following chunk functions.

**range function** The range type of functions span a consecutive region of values. It requires that the values can be ordered, such that ranges can be defined. A unique and dense sequence suits the best with this function, to have ultimate control over the volume of the ranges. Without this characteristic, ranges can have a hard to predict volume. The function has obviously no means to control volumes with duplicate values. However, to a certain extent the range function can deal with duplicates, for as long as the volume of a range can still be controlled to be small enough to fit. The idea of the range function can also be extended into $n$-dimensional worlds by using cubes or more dimensional “ranges”. A special form of the range function is the divider function, where the range starts or ends at the borders of the encompassing scope, thereby just dividing into two consecutive regions.
hash function  Hash functions take key values and produce new values that ideally are equally spread over the space of values produced. Typically the space of produced values is smaller than the possible space of input key values. Due to this characteristic, hash functions are not reversible: they are so called “one-way” functions that are meant to project one space into another. The actual chunk boundaries are defined on the result of the used hash function in the projected space. As a result, no relations other than the hash function between the input data and their fragmentation exist. Advantage of hash functions is that they do not require a sequential order such as the range function. However, it is hard to tell what the scope of input values belongs to a chunk. A good hash function does not produce a skewed result, however, with duplicate values, the result contains at least the same number of duplicates.

omega function  The omega-storage structure is defined to divide based on bit-level of the attributes involved [41]. Given the bit representation of the involved values, it tries to find the bit position that distinguishes the values the most when applying the split. However, because it operates on bit values, the domain of a key needs to be known upfront, since all values need to map to an equally sized bit string. Typically, such bit string is chosen to be as small as possible. In [41] the author does not elaborate on how non-numeric values, such as strings, should be used with the omega-storage structure. A possibility for strings would be to use some mapping from the string value to an integer value that can be turned into a bit value, such as a simple hash function. Because the division is based on the bit position, the split always produces two boxes. This could be extended into $2^n$ resulting boxes by using the $n$ most distinguishing bit positions.

round-robin  The round-robin fragmentation function is, unlike the previously mentioned functions, a function that does not operate based on the actual data. In the round-robin case, there is typically some external entity that row wise distributes the data over a number of servers, such as a load balancer. Because the actual condition on when to send what data to what server is external (and might even be completely random), there is no function that describes what data goes where. As a result, the best one can do with the round-robin function is to administer that there is more data in the siblings. This comes close the clone function, with the difference that the full data is not available on every box. Note that round-robin is only feasible for an append-only environment which is known to
be unique by some key, for each insertion would otherwise have to be checked against the other descendants in order to guarantee the disjointness constraint. This constraint is just explicitly guaranteed to hold in the append-only case, but the Armada system itself can only get control over this by paying very high costs defeating the purpose of round-robin. Without this check, the function is a threat to the Armada strategy. Because no guided redirection can be made, the risk of a client dangling between two or more boxes exists.

**projection function** For completeness, we include the chunk function that does vertical fragmentation. Unlike the previous functions, the projection function does not operate on the values. This means fragmentation performed by this function is not based on the data in the schema at all. There is no relation to the data, and hence volume control through this function is difficult. Instead, the projection function fragments the schema, resulting in a vertical split of the columns in the schema over a number of boxes.

**compound function** The compound function is a pseudo function consisting of multiple functions. By definition of this function, it is only a mere convenience function that can be simulated by applying all the functions it contains in order. However, we like to include it here, to discuss the different possibilities that exist when combining functions. By combining functions, more complex spaces can be described, for instance multiple ranges. When the ranges are very small (a tuple) this might also be used for addressing “per tuple” spaces. Given this granularity possibility, a fragmentation based on a regular `SELECT A, B, C FROM X WHERE Y` SQL query could also be modeled in a compound function as a combination of the projection function with some of the other functions, given that the key is suitable for the selection. As a last note, the round-robin function can not be combined with any other, as it does no selection on the data, while still fragmenting horizontally.

Most fragmentation functions have some requirements in order to operate optimally. The range function can only be applied when there is some knowledge on the space it is being applied to. It makes no sense to divide a space in two if the separation is done in such a way that it does not divide the actual data, and hence leaves one box empty. It is necessary to have a clear idea on the “minimum” and “maximum” values in the space and what is between those.
The hash function is less selective than the range function, but depending on the hash function itself, its output can be skewed. Having the right function for the key values is a complicated task, and belongs to the standard hash research problems. In order to make a good hashing function, it is necessary to know what the input space is, and whether it is skewed already.

4.3.2 Classifications

The aforementioned implementations of chunk functions can be categorised according to several classification schemes, based on some properties of the functions.

An obvious first categorisation is based on the fragmentation being made by the functions. Here basically two categories are to be distinguished: horizontally and vertically. An extra category for a combination of the previous two is also included for convenience. The horizontal fragmentation functions are the round-robin, hash and range functions. They typically make a row-wise fragmentation of the data, where tuples are either within the selection or outside of it. In the category of vertical fragmentation functions, only the projection function fits. It is the only function that fragments by means of splitting the columns in the table. Of course a combination between the projection and a horizontal fragmentation function can easily be made, resulting in both horizontal and vertical fragmentation. This is covered in the combination category.

Another classification considers how the chunk functions operate. Three types of chunk functions can be distinguished: functional, predicate based and simple/compound functions. The functional class uses some algorithm to calculate a new value (apply a function) from the given key value, which is the base for the decision whether the function covers the key value, or not. The only function that exhibits this characteristic is the hash function. Predicate based functions use a certain predicate value to directly decide upon the key value whether the function covers the key value or not. Functions in this category are the range and projection functions. While the projection does not deal with the actual data, it still can be considered a predicate based function if the columns are considered the key value for that function. The last category of functions in this classification is for the simple or compound functions. Functions of this type are either not doing anything at all, or a combination of other functions from other categories. The compound function is such typical function.

A more abstract classification is based on two axes of the different functions. The one axis considers whether the function is based on the data in the Armada
relation or not. Not depending on the data results in decisions that cannot be 
reproduced from within the Armada, as some external decision resulted in the 
function. The second axis is made for data dependent functions only. Data 
dependent functions can either depend directly or indirectly on the data. Dir-
ect functions apply some condition on the data directly \((f(v))\), while indirect 
functions first apply a (number of) functions before applying some condition 
\((f(h(v)))\). This classification is depicted in Table 4.1. The categories are in the 
top half of the table, examples in the bottom part.

### 4.4 Function Trails

A number of functions applied in a sequence is called a \textit{chain}. In Armada a trail 
is a typical chain of functions applied to the data. To determine the possible 
contents of a box, it may be necessary to examine the full trail. We refer to
the possible contents as the “coverage” of a box, or function, i.e. that what it selects. In practice, the tuples that match the function.

A trail consists of steps, which each have a function associated to them. This function need not to be a chunking function, also cloning and combining functions are administered in the steps. Since chaining is about determining the coverage, the clone function can be left out of consideration in such trail, since it doesn’t change the coverage in any way. The coverage before and after the clone operation is the same.

An idea behind Armada is growth over time. This means that trails also grow over time. New steps get added to previous trails, and so a history of steps is being retained. As a result, also the functions in the trails are used on top of each other. In the chain of functions, the functions’ coverage can be based on the previous one(s), or fully self containing. The first one we refer to as relative functions, the latter one as absolute functions.

Relative Domain Functions When dealing with relative functions, the coverage of the function is based on the coverage of the previous function in the chain. One could compare relative functions to percentages. Every new function works on top of the old function coverage as if it were 100%. By this definition, each new relative function is most of the time more “narrow” than the previous in the chain, but at maximum equal to it. This avoids redundancy like might be the case with absolute functions to get the right subset, for instance as a combination of different functions. The functions can be simple as they do not have to take the coverage of the previously applied functions into account. The right-hand side of Figure 4.1 depicts this relative coverage, where the dashed box is based on the outer box it is contained in.

Figure 4.1: A graphical representation of absolute and relative function coverage.
Absolute Domain Functions  The class of absolute functions have a fixed coverage based on the entire value domain, as observed by the origin. This makes the resulting coverage independent on previous functions, although it might change the actual coverage depending on them due to occlusion. In the most extreme case the function before the absolute function in the chain selects a part of the entire space that is not in the coverage of the absolute function. An example of a partial overlap of this kind is depicted on the left of Figure 4.1. Even though the dashed function coverage is within the outer box, it also falls partially outside of it. The result may be undesirable since the dashed function may expect values in its entire coverage, while obviously only the overlapping part receives values. This does, however, indicate the independence of the function with respect to its environment. In practice, it is most useful for absolute functions to be contained into the predecessor function, and so be more “narrow”, like relative functions.

The advantage of an absolute function is that when functions in front of it in the chain are removed, its coverage stays the same, even though the received values need not to stay the same. Also when it is moved to another location, regardless whether the whole chain (trail) is copied or not, the coverage of the function remains the same. However, in the Armada model, this is unlikely to happen.

Function Composition  A chain of functions, either relative or absolute, is a concatenation of multiple functions which need not to be of the same type. However, chaining arbitrary functions might not be possible or a sensible thing to do based on previous functions in the chain.

Absolute functions can be combined with relative functions, and vice-versa. Because absolute functions in principle do not take into account the coverage of the predecessor function, they run the risk of being ineffective due to non-coverage, as mentioned previously. Two range functions can easily be positioned in such a way that there is even no overlap between the two. However, range functions are an easy example, since they allow to easily see their overlap. Hash functions, on the other hand, are much more difficult. Not only is it harder to visualise their coverage, but also is the way in which any other function can be applied on top of them more difficult. Due to the characteristics of the hash function, it is not known what the relation between the tuples selected by the hash function is. Hence, it is hard to determine what the coverage of the function is, which makes it in turn hard to apply for instance a range function to tackle a skew problem.
4.5 Self-Management

The ultimate self-maintaining Armada system applies the clone, chunk and combine operations automatically upon need. The problem here is not how to apply, but when. The Armada model specifies how to effectuate an operation, but not at what conditions (when) using which function.

**Cloning** Cloning is used to add redundancy to the system. While redundancy in general is a safeguard for the system not to loose data, it can also serve as performance win due to load spreading. However, cloning adds the need to keep all clones synchronised. Therefore, cloning in itself is an operation that remains expensive after the operation has been applied. In some cases, a performance problem can be solved using chunking as well. Consider a hot-spot created by point queries, splitting it up in two distributes the load as well. The same hot-spot, but created by range queries would not reduce the load in terms of query hits, but perhaps it does in terms of the amount of data each of the boxes have to use for answering. Eventually, the entire hot-spot may also be chunked to a new and faster machine that can handle the load more easily.

While chunking causes much less overhead after the operation has been applied to maintain the chunk operation, it is preferable for an Armada system to use chunking over cloning, where the consistency of both clones needs to be maintained in some way or another for it to be effective. With this insight, the decision to use cloning unavoidably has to come from outside the system, e.g. as a (human) preference. Only active boxes are eligible for such operation, hence the further the Armada is chunked into pieces, the more clone operations need to be applied to clone the entire Armada system.

**Properties** A more complex solution would be more geared towards ease of use relying on the flexibility of the Armada system. Here, the tree based structure is used to define an inheritance based property system. The properties define boundaries for the Armada system to operate within. The more properties defined, the more restricted the systems in the Armada are regarding their autonomy. Two properties for cloning would suffice to balance the requirements from outside the system with the autonomy inside the system. These properties are the minimum and maximum levels of replication that the Armada system should maintain. The maximum level implies that the system has a freedom to clone more than the minimum level dictates. This implication means that the Armada system is able to find situations where cloning has benefits over
chunking, even considering the burden of the continued maintenance to keep both clones in sync.

In this complex solution, at any time, on any box in the Armada, the set of properties can be changed. Such change triggers addition or removal of clones, to be automatically performed by the Armada system. However, the complex structure that the lineage tree can become makes this rather difficult to achieve. First, in a solution like this one, the inactive boxes, such as for instance the origin, are also valid boxes to control the properties of. This results in for instance creating a clone of the origin box, representing the entire Armada. Second, due to the inheritance of the tree, it becomes less obvious what actions are performed if at various levels in the tree properties are changed. For instance, a box may have inherited the maximum replication level, but its minimum replication level explicitly set. There are more solutions for what should happen if the inherited maximum replication level is changed to a value lower than the minimum replication level. Lastly, it is difficult for an Armada to know that the properties set on the boxes are satisfied. On the one hand, the sites hosting the boxes aim to remain autonomous and not to get involved in contracts with others. On the other hand, a box that inherits a minimum replication level needs to know whether it should clone itself, or whether the predecessor has already taken care of this. Eventually only the active boxes can perform the clone operation as requested, which means that in an existing tree it can only be determined how many clones of a sub tree exist, by traversing it down and building an image of which parts are cloned and how often. Combine operations in this scheme make it even more complicated since arbitrary boxes may be combined, and hence it may be impossible to deduce what part of the tree is still redundant and what part not due to the nature of some chunk functions.

**Chunking**  A self-managing objective that an Armada system can take, is to fragment automatically when capacity problems occur. This is a natural objective, as without fragmentation, the system is limited to the size of a single system. Hence, the target of the objective is to allow the system to grow beyond the limitations of a single system.

Using pre-allocated small chunks, fragmentation does not apply any more like before. Since fragmentation is applied in advance to create the small chunks, a new chunk is allocated on another system when it does not fit on the original one. Here the slack space profile setting is aligned with the size of the chunks. A chunk is not assigned if there is not enough free space left. Key problem with automatic chunking is what function to take to achieve this effect.
A solution that avoids creating special functions for the problem at hand, is to chunk the data upfront in small chunks. Many of these chunks fit on the same system. However, once the system overflows it is obvious what chunk should be placed somewhere else. With these small chunks it is also easy to define the “load” for each chunk, to possibly detect a hotspot this way. With this knowledge, such chunk could be cloned, or chunked such that the box is essentially moved to another system. The drawback of using many small chunks is that the Armada itself grows artificially fast because of all the mini chunks allocated on the same system. This stresses the lineage trails, and their effectiveness on large scale.

To create less lineage “overhead”, a chunk operation is only performed once capacity problems demand so. Chunking in this case means that a follow up has to be found for the current box, such that additions of data can continue. Without any knowledge of the data inserted in the box, the best way to chunk that box is by splitting it in two equal pieces. Equal here can be seen as in “data coverage” or as in the number of tuples the two resulting boxes contain. While the latter achieves a better distribution of the data over the two new boxes, the former is easier to implement using a range function, typically by taking halfway between the minimum and maximum values. By design, some hash functions are well suited for making an equal distribution of the original box. However, since the contents of the box differs per case, it is hard to tell if the chosen hash function also distributes the box contents evenly.

Statistics Assume a range-based scenario, where range functions can be applied. Ideally, when a box is chunked purely because of performance problems, the hot-spot in the box is detected. This can be based on statistics kept by the database kernel of frequently accessed ranges derived through e.g. a simple histogram. Depending on the histogram, one or multiple hot-spots may be found, or no hot-spot at all. Instead a close to even distribution may be found. In case of a single hot-spot, the chunk function can be designed in such a way that it splits the hot-spot, or that it extracts the entire hot-spot to relocate it to another machine. The latter option may be feasible if the Armada system has reasonable beliefs that the other machine is able to handle requests to that data more powerfully. This can be e.g. due to other hosted boxes, or physical hardware configuration matters. When multiple hot-spots are found, each can be assigned to a separate machine, or perhaps multiple combined into one and moved together.

In case of an even spreading of load, e.g. when the histogram shows no
data that is obviously accessed more often, the entire box can be simply seen as hot-spot and hence split in two even pieces. However, in such case also a hash function could be beneficial due to a more even distribution. Though, since the result of a hash function is not a consecutive sub-range of the original space, it is possibly less effective as a range function if there are many range queries performed on the data. When a range query fits in one of the sub-ranges of a range function, it can be executed on a single box, whereas with a hash function this is not possible. However, once the range selection spans both boxes, there is no clear benefit, since both boxes need to be consulted, as is the case for a hash-function.

**Storage Capacity** When a box runs out of free space, said box needs to be chunked to resolve a capacity shortage problem. Detecting such shortage usually is fairly simple given that there is a notion of available “free” space. After this has been detected, a new box has to be added to extend the system. One option for that is to chunk the existing box in such a way that all data remains where it is, while new data ends up in the newly allocated box. This only works in certain data scenarios. First, the current contents of the box has to be described. While minimum and maximum values can be used for that, this may not be sufficient to achieve a split where new data goes into the new box. Randomly inserted data, for instance, is hard to describe. This highlights the second required point for this method, which is the need for consequently added data, as typically found in log-like data that is append only and fairly suitable for range partitioning.

Many other scenarios are not suitable for range partitioning this way. Instead they may have gaps in their keys which can be filled in later. For these scenarios it is better to free up some space on the original box, such that insertion of data on the original box is still possible. The free space allocated, the slack space, depends on the chunk function chosen. The amount of slack space available, influences whether the original box has to be chunked again later on when it reaches its capacity limits again. The easiest way is to simply split the entire box in the middle, to effectuate a slack space of 50%. This amount is the best one can do in a scenario where it is unknown how and when the data is inserted. If there is some information available on how the data is inserted, the slack space level can be adjusted to suit, such as shown before with log-like data insertion. Advanced statistics may record the amount of “appends” versus “inserts” given a range ordering, as to define an efficient slack space percentage. If slack space is allocated and never used, the result is unused space in the Armada system.
Summary

An important aspect of the Armada model, is the function as part of an operation that defines how data is spread. The amount in which this function can be devised by a site, defines how well such site can operate on its own, and hence how well incremental scalability within the cluster is effectuated.

Functions are not always independent, the chain of functions of which they are part in a lineage trail, may put certain restrictions on them. A limited amount of function types can be identified to be used, with their own properties, affecting the usefulness of other functions to be applied in a chain.

Thus far, the decision on a certain function is best done based on simple heuristics, until further research shows better ways. This way it is easiest to avoid clones in a system, and to apply chunk functions of a range type, based on histogram information of data usage frequencies.