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.
3 The Armada Model

3.1 Introduction

The Armada Invencible is the well known Spanish “invincible” fleet of late 16th century. Full of glory, this grand armed fleet wrote impressive history, and yet today its reputation is its invincibility. However, these promising words are not meant to drive up the expectations for the work we present here. In fact, the invincible Armada faced some bitter moments of reality, when it was defeated by Dutch and British ships in the battle of Gravelines. Again, we do not intend to anticipate on any expectations here. Instead, the model that we present in this chapter is named after a fleet of ships. The name Armada is used because it is internationally well understood to be such fleet. The discovery of several parallels between Dutch maritime history and our modern database research has also contributed to the title of this thesis. The contents of this chapter is based on the paper “Armada: a Reference Model for an Evolving Database System” [30].

Autonomy The foundation of Armada, is the power of autonomy. Like in a fleet, where each ship has its own autonomous captain. However, even though each captain is independent, efforts are made such that all ships together act as a group, with the same target. The Armada model defines a way to allow autonomous sites to work together as a group towards the same target: serving a distributed database.

Decentralisation Traditional distributed approaches [68] are designed with a strong emphasis on data availability and maintainability. However, these approaches mostly rely on vulnerable centralised techniques. Whenever the central server becomes unavailable (or worse, demonstrates performance prob-
lems), the system at large may come to a halt. Also, scaling is often limited by the capacity of the central server, which eventually becomes the bottleneck of the whole system.

This problem has been recognised and targeted by P2P systems. Many distributed hash table (DHT) approaches have been proposed [62, 59, 51]. They form the base for structured overlay networks on top of which P2P database systems are built, such as [37, 29, 40]. DHTs provide associative key-based lookups of data, in the form of a single value, row or block of (cached) data depending on the system’s purpose. To find the data, the corresponding key for that data has to be somehow known.

In P2P systems, nodes frequently join and leave the network. This property inherently stems from the environment they are situated in. File sharing between millions of (unknown) people on the Internet simply introduces different time zones, unreliable connections and people unwilling to share (any further). Also, the files available in the network depend on what individual people like to share with others. These — mostly social — aspects of availability and location of data are reflected in the general structure of many P2P systems. They provide an efficient search for data in the network, if it is currently available. Data that is currently not in the network, does not exist. Only through stale pointers, data that once existed can be found, but in general P2P networks aim at quickly removing such stale pointers. Hence, there is no notion of a data space in these networks that allows determining whether an answer can exist, or does not exist.

**Evolution** The schema based approach towards distribution as explored in the Armada model delivers a solution for both problems. The model starts from a complete relation and breaks it up into pieces. In this process, it keeps a lineage-based administration of the actions taken. The data within each piece is characterised by a decision function about its permissible content. Moreover, each piece can be recursively broken up further using a refinement relation of the decision function. The pieces naturally map on the data units used in distribution, and the functions are the navigational handles through the ensemble of autonomous nodes.

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.

3.2 Innovations

The Armada model innovates on three key areas, which characterise the capabilities of the model and form the key components towards autonomy, decentralisation and evolution.

**Function-based Distribution Control**  Armada uses general functions to describe the content of a piece. These functions are arbitrary expressions and can be freely chosen on a piece-by-piece basis. They enable an Armada instance to adapt easily to the (local) data distribution or workload characteristics, as part of its autonomy. This is in contrast to DHT systems where a single global decision function, e.g. a hash function, is used to control data locality. The Armada model delivers more flexibility by this freedom of choice for functions.

Traditional systems can, in theory, deliver the same flexibility using substantial human efforts, due to the burden imposed by static configurations. The dynamic aspect of Armada aims at adapting on demand by creating pieces with carefully crafted functions to cater for the situation at hand.

**Incremental Query Evaluation**  The Armada query evaluation scheme brings the pieces back into the original relation incrementally, like reconstruction of a jigsaw puzzle. This means that the data is available in the system as ordinary relation fragments. The decision functions allow precise localisation on their logical whereabouts. Even if the piece itself is currently not available in the system, its position in the query result sequence can be derived from its function. This contrasts with DHT systems where this is not known and the only conclusion to be made is that a value that cannot be found is currently not available. Functions in Armada describe the data space of the relation. If a value being addressed is not covered by any function, the value cannot possibly exist in any of the pieces. Like in traditional systems, for a query in Armada, it is a priori known what pieces should be inspected before the complete answer has been retrieved. Even if those pieces are temporarily not available. Hence, data that is currently not available, remains to be known to the system via the function administration of the Armada model.
Active Client Participation One can consider traditional centralised approaches to fail to distribute the metadata required for localisation over the system. The other end of the spectrum is where no central server exists, and instead all individual systems contain the full metadata regarding the data whereabouts. In both situations, the entire system remains a monolithic cluster as long as traditional approaches to transactions and query execution are retained.

For a distributed system to benefit from its infrastructure, a more relaxed transactional setting is required. In such setting for instance locks do not cross a system boundary, hence excluding distributed locks. Armada assumes, like other distributed systems, that the system is used in a loosely coupled setting. Local consistency in the data pieces is required, but global consistency in the system as a whole is left to the client to solve. However, in this thesis we do not focus on the implementation of transactions by clients. With partially distributed metadata and a relaxed transactional setting, the Armada model enables full decentralised query execution.

This puts an emphasis on the client's role in the Armada model. In particular, Armada assumes that a client becomes an active participant. Active implies that the client plays an important role in steering query resolutions. Instead of relying on the server to construct a complete query result, the client expects a server to answer in portions whenever possible. Not only are partial results returned to the client, but also directions on where and how to get the remaining parts. Here the client is offered some opportunities to influence the execution of large queries, as well as responsibilities to e.g. maintain global consistency.

3.3 The Armada Model

Classical designs for distributed databases, require a centralised server that holds all metadata describing the whereabouts of the available data. Due to the centrality of such server, it can easily become the bottleneck of the entire system. The central server is accessed to lookup or update metadata for both operations that query or update the actual data. More importantly the metadata has to reflect changes in the structure of the system, such as addition or removal of nodes, or a reorganisation of the data for load balancing purposes. It creates a bottleneck that limits the overall performance and scalability of such systems.

Bottleneck An obvious solution to this bottleneck problem is full replication of the metadata over all participating systems. Such designs have to rely on
the consistency of the replicated metadata, and hence, each structural change requires a synchronised update on all nodes in the system. However, because all metadata is available locally, data operations are cheaper, at the expense of significantly higher prices for structural changes. While the latter are often infrequent these high costs may be acceptable, but they imply all participating systems to be available and willing to perform the metadata update, with no other update running at the same time. These additional constraints prohibit efficient dynamic changes of the data distribution.

**Metadata** With the Armada model, we aim at finding a balance between these two extremes. On the one hand, Armada does not come with a centralised server, and thus avoids the bottleneck of metadata lookups. On the other hand, Armada does not require to replicate all metadata on all nodes. Instead, Armada finds a compromise by replicating metadata partially only, and being able to cope with incomplete or stale metadata. Obviously, each node holds its own local metadata, e.g. schema information about the portion of the database stored, and keeps it up-to-date. In addition, it holds some remote metadata, such as information from nodes in its vicinity. To limit maintenance overhead, the idea is to limit remote updates of metadata to those nodes that exchange data due to structural updates. Thus, remote metadata is not necessarily kept up-to-date at all times. Rather, an Armada-node assumes that its remote metadata is an approximation or a past snapshot of the situation of a remote node.

**Armada** The inspiration for our novel reference model comes from the Armada analogy. An Armada is a fleet of ships, that forms a unity although each ship has a captain who is sovereign. The Armada model reflects this property in a minimal set of relations between the captains of the ships. Each ship has cargo (data) stored in barrels (boxes) that are addressed by cargo documents (trails) kept by the captain. A captain can repackage the cargo on his ship, and/or hand over (parts of) his cargo to one or more other ships in the Armada (cloning, chunking). Repackaging may also occur if barrels are empty or only partially used, such that multiple barrels are put in one (combining). The cargo documents describe the content of each barrel as well as the lineage of the respective cargo. A captain keeps one cargo document for each barrel he has aboard his ship. When handing over cargo to other ships, the respective cargo documents are duplicated; the original copy stays with the captain on the old ship and the other one is attached to the barrel on the new ship. Thus,
not only does each captain know what cargo his ship currently carries, but also where he sent the cargo that he once had aboard, and where any cargo he ever transported came from. In fact, the cargo documents kept on each ship provide sufficient information to allow the captain to locate any cargo item in the whole Armada.

When barrels are transferred to other ships, the captain administrates to who the barrels were transferred in the now obsolete cargo documents. To be able to track down a barrel, copies of the old cargo documents are attached to the cargo documents of the new barrels, and vice-versa. We use the analogy of a real Armada in our world of database servers (ships) and apply some of their properties to them.

### 3.3.1 Notation, Terms and Definitions

We informally introduce the term (data) box to refer to the portion of the data that is hosted at a site. We assume that the content of a box can be described by an arbitrary function $g$. The actual specification of such function is left to the instantiation of a specific Armada system. In the course of this section, we provide some constraints for such functions. Chapter 4 discusses these functions in more detail and give some simple examples.

Further, we use the term structural operations to refer to operations that create and modify the data distribution across sites, i.e., operations that replicate, (re-)fragment or merge portions of the data. Data boxes form the entities that these structural operations operate on.

**DEF. 1** Be $B'_i, B'_{i+1}, \ldots, B'_{i+n}$ existing boxes in an Armada system with functions $g'_i, g'_{i+1}, \ldots, g'_{i+n}$ describing the content of each box. A structural operation $o$ operates on one or more boxes $B'_i, B'_{i+1}, \ldots, B'_{i+n}$ and produces one or more new boxes $B'_j, B'_{j+1}, \ldots, B'_{j+m}$ with functions $g'_j, g'_{j+1}, \ldots, g'_{j+m}$ describing the content of these new boxes. A structural operation cannot generate new data, but must not “loose” any data, either. Hence, we require that

$$g'_j \cup g'_{j+1} \cup \cdots \cup g'_{j+m} = g'_i \cup g'_{i+1} \cup \cdots \cup g'_{i+n}.$$

Inspired by the cargo documents of the Armada analogy, we introduce lineage steps and lineage trails to store and administer metadata. A lineage step
captures the logistic information of applying a structural operation to a box:
g, the function that is applied (and hence describes the content of the
new box),
S, the site that the new box is shipped to, and
B, the identifier of the new box (for the convenience of later reference).

DEF. 2 A lineage step \( s = [g, S]:B \) is a composition that identifies the appli-
cation of a structural operation, resulting in a new box B on site S with function g
describing the content of the new box. The box B' that s is applied to is identified
by the lineage trail \( T' \) that s is appended to (see below).

Each box in the Armada is uniquely identified by a lineage trail that captures
the whole history of its data.

DEF. 3 A lineage trail, or trail for short, \( T = s_1.s_2. \cdots .s_l \) is a sequence of \( l \in \mathbb{N} \)
lineage steps. With \( s_i = [g, S]:B \), T identifies box B on site S.

DEF. 4 Be B'', B', and B boxes on sites S'', S', S with their content described by
functions g'', g', g, respectively. Further be B'', B', and B identified by the trails
\( T'', T' = T''.s' \), \( s' = [g', S']:B' \) and \( T = T'.[g, S]:B \), respectively. We call
\[
T'' \quad \text{a predecessor trail of box B',}
\]
\[
s' = [g', S']:B' \quad \text{the local step of box B',}
\]
\[
T' = T''.s' \quad \text{a local trail of box B',}
\]
\[
s = [g, S]:B \quad \text{a successor step of box B',}
\]
and analogously for boxes B'' and B.

The metadata maintained and stored for each box consists of a set of prede-
cessor trails, exactly one local step, and a (possibly empty) set of successor steps.
The predecessor trails represent the box' heritage. The local step describes the
box itself, and the successor steps point to the box' offspring. The predecessor
trails and local step are set upon creation of a box, while the successor steps are
only set once a box participates in a structural operation.

We assume that a structural operation (logically) removes all the data from
its input boxes (transferring it to the newly created boxes), and destroys the
input boxes. Only the respective metadata (lineage) is kept. This assumption
relieves us from the need to consider different versions of each box, and thus
helps to simplify the model. The assumption does not limit the generality of
the model. In a practical implementation, this does not necessarily require
a (physical) copy of all data with each structural operation. Instead, simply renaming the box can be sufficient.

To simplify the presentation, we omit the set notation whenever a set of trails is empty or contains only one trail. In the first case, we simply omit the empty trails set; in the latter case, we depict the only element as singleton. Thus, the metadata for boxes $B''$, $B'$ and $B$ of Definition 4 is depicted as follows:

<table>
<thead>
<tr>
<th>pre</th>
<th>loc</th>
<th>suc</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T'' = T'' \cdot [g'', S''] : B''$;</td>
<td>$[g', S'] : B'$;</td>
<td>$g', S'] : B'$;</td>
</tr>
<tr>
<td>$T' = T'' \cdot [g', S'] : B'$;</td>
<td>$[g', S'] : B'$;</td>
<td>$[g', S'] : B'$;</td>
</tr>
<tr>
<td>$T = T' \cdot [g', S'] : B$;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The set of successor steps is empty for all boxes to which no structural operation has been applied yet, i.e., all boxes that physically exist and store data. The set of predecessor trails is empty for one box in an Armada, the origin.

**DEF. 5** An Armada instance is born as a single initial box $B_o$. We call $B_o$ the origin of the Armada instance. Obviously, the origin has no predecessor trails. Further, since no structural operation is applied to create the origin, there is no function that describes (restricts) $B_o$’s content. We indicate this by $\%$ in the local step of $B_o$:

$$T_o = [\%, S_o] : B_o.$$  

### 3.3.2 Structural Operations

To let an Armada evolve from the origin, we consider the following three structural operations.

**Replication: the clone operation**

**DEF. 6** The clone operation operates on one box $B'$ with function $g'$ and generates one or more new boxes $B_j, \ldots, B_{j+m}$ that all contain a copy of $B'$’s data. Hence, their functions $r_j, \ldots, r_{j+m}$ are all identical to $g'$.

Replicating a data box is the action of copying its content to a new location. We call it the *clone* operation, denoted by function $r$. Consider the following example of cloning the origin box $B_o$:

$$T_o = [\%, S_1] : B_o; \quad \{ [r, S_1] : B_1 \}
\quad [r, S_2] : B_2$$
$$T_1 = [\%, S_1] : B_o \cdot [r, S_1] : B_1$$
$$T_2 = [\%, S_1] : B_o \cdot [r, S_2] : B_2$$
In this example, the origin has two successors, $B_1$ and $B_2$, which themselves have no successors.

From the origin meta-data we can now observe two trails by reading from left to right: each of the successor steps, followed by the local step and the predecessor trail. The full local trail for the two new boxes (successors of $B_o$) is also visible.

Following Definition 6 the number of new boxes produced can also be a single one. Strictly, this is no cloning operation any more: since the original box is (logically) destroyed after the cloning, its data is not replicated, but rather moved to a single new location. However, there is no reason to prohibit this in the model.

Although we use different site identifiers for the two new boxes in the above example, it is perfectly sound with the model to produce two (or more) clones of a box on the same site. The question, whether this is reasonable in practice, is not relevant in the context of a reference model.

**Fragmentation: the chunk operation**

**Def. 7** The chunk\(^2\) operation operates on one box $B'$ with function $g'$ and generates one or more new boxes $B_j, \ldots, B_{j+m}$ that all contain a fraction of $B'$'s data. We require that all fractions are disjunct, but no data is lost, i.e., the following must hold for new boxes' functions:

$$f_j \cup \cdots \cup f_{j+m} = g' \quad \text{and} \quad \forall k, l \in \{j, \ldots, j+m\}, k \neq l : f_k \cap f_l = \emptyset .$$

Fragmenting data means it gets spread out over multiple boxes. We call this the chunk operation, denoted by functions $f, f', f'', \ldots$. Consider the following example of chunking the origin box $B_o$:

$$T_0 = \begin{cases} \%: B_1 \end{cases}$$

$$T_1 = \begin{cases} \%: B_0 ; \\ [f, S_1]: B_1 \end{cases}$$

$$T_2 = \begin{cases} \%: B_0 ; \\ [f', S_2]: B_2 \end{cases}$$

The origin has been chunked into two parts, using chunk functions $f$ and $f'$. Like with cloning, in case there is only one result box, a move operation is effectively being executed.

\(^2\)We felt free to ‘invent’ this verb.
Merging: the combine operation

DEF. 8 The combine operation operates on one or more boxes \( B_i', B_{i+1}', \ldots, B_{i+n}' \) with functions \( g_i', g_{i+1}', \ldots, g_{i+n}' \), and produces a single new box \( B \) that combines all the data of the input boxes. The produced box' function \( m \) spans the domain of \( g_i' \cup g_{i+1}' \cup \cdots \cup g_{i+n}' \).

While cloning and chunking are growing operators, the combine operation is a shrink operation. Applying it to a number of boxes merges them into one. However, this operation is not restricted to acting as an inverse-operation to the clone and chunk operations, i.e., re-constructing a previously cloned or chunked box. Our model allows to apply the combine operation to an arbitrary set of boxes. This is depicted in the following example, where a clone \((B_4)\) and a chunk \((B_6)\) are combined into a new box \((B_9)\), creating a duplicate free combination of the inputs' data.

A note on the notation is necessary: for convenience, clarity and space reasons we do not write down the predecessor trails. From now on, we use a reference to them in the form of \( T_x \) where possible instead.

\[
T_4 = T_3 \cdot [r, S_1]:B_4; \quad [m, S_1]:B_9
\]
\[
T_6 = T_2 \cdot [f'', S_2]:B_6; \quad [m, S_1]:B_9
\]
\[
T_9 = T_4 \cdot [m, S_1]:B_9;
\]

Again, if there is just one box merged, the result is a semantic move of data.

3.3.3 Lossless Principle

The clone, chunk, and combine functions permit an arbitrary Armada constellation to be constructed. It would even allow for destructive functions, i.e., creating rubbish.

An important class are the lossless constellations. That is, at any point in time it remains possible to combine the boxes on a single site with infinite resources without loss of any box content.

This property is fulfilled for clone operations by definition. For the chunk operations it limits their definition. It precludes aggregations and general (schema-based) data transformations.
### 3.3.4 An Armada Database

In practice, databases based on the Armada model evolve over time quickly. For many reasons, e.g., resource limits, boxes are the target of chunk and clone operations. An illustrative example of a database with 5 boxes is shown below.

\[
T_0 = \{ [\%, S_1]:B_o; \} \\
T_1 = T_0 \cdot [f_1', S_1]:B_1; \\
T_2 = T_0 \cdot [f_2, S_1]:B_2; \\
T_3 = T_1 \cdot [f_2, S_1]:B_3; \\
T_4 = T_1 \cdot [f_2', S_3]:B_4;
\]

(a) the origin overflows when inserting 1

(b) box \( B_1 \) overflows when inserting 11

(c) the final state of the Armada

Figure 3.1: Sample Armada with 5 boxes.

In this example, we only use fragmentation functions to spread the data in the Armada over 5 boxes. Each box is hosted on a separate site for ease of presentation. The origin box \( B_o \) was first chunked into boxes \( B_1 \) and \( B_2 \). The
first of these two children, $B_1$ is chunked again, resulting in boxes $B_3$ and $B_4$. The evolutionary steps are graphically shown in Figure 3.1 using symbols which indicate the coverage of the functions applied in the operations on the boxes. The symbol ‘□’ is used to represent the data at the origin of the Armada, in box $B_0$. The other symbols; ‘\text{\textdagger}’, ‘\text{\textless}’, ‘\downarrow’ and ‘\triangle’ represent pieces of the origin box. Note that the symbols equally divide the original square symbol. This is of course only a drawing issue, which is not necessarily true for the fragmentation functions being used.

For this example, we describe how the tree from Figure 3.1 is built over time by inserting data into the Armada. In the initial situation, depicted in Figure 3.1a, only $B_0$ exists on site $S_1$. For the sake of the example, the boxes store simple integer values. Each box has a fixed capacity of 5 of such integers. Normally this capacity is determined by the site that hosts the boxes and the size of the data items, but for the sake of clarity we use these fixed sizes. The data to be inserted in the Armada, in order, is for the example:

$$D = \{2, 5, 7, 12, 23, 1, 72, 24, 11, 16\}$$

Since there only fit five integers in each box, the origin $B_0$ consists of $D(B_0) = \{2, 5, 7, 12, 23\}$ when the next integer, 1, is attempted to be inserted. Since it does not fit, a chunk operation is performed. In our example, we split equally, which results in $D(B_1) = \{2, 5, 1\}$ and $D(B_2) = \{7, 12, 23\}$. The fragmentation function $f_1$ used here selects the range $[0 \ldots 5]$. The function $f_1'$ selects the complement of $f_1$: $(5 \ldots \infty)$. Beware, this decision is taken at site $S_1$ in ‘full autonomy’, it is not inherent to the algorithm.

In Figure 3.1b, the state of the Armada after the first chunk operation is depicted. As can be seen, the data from the origin box $B_0$ has been moved to boxes $B_1$ and $B_2$. Note that the order of the items in the example is maintained, but this is not a restriction of the Armada model. The only restriction on the boxes is that each box only holds data that matches its respective local trail description.

Continuing the insertion of values, now the right box has to be searched. Inserting the values 72 and 24 ends up in box $B_1$. The origin box $B_0$ is not active any more, and redirects if being consulted. Since it knows the functions of its successors, it can easily tell that both values fit in the $(5 \ldots \infty)$ range of $B_1$ \footnote{A more detailed description of how this redirection is decided upon is given in Section 3.3.5.}. Also the next integer, 11, fits in $B_1$’s range, but since the box is full, a chunk operation has to be performed again. The result of this chunk operation is depicted in Figure 3.1c. Again the data values have been equally split over
the two new boxes $B_3$ and $B_4$. The last integer to insert, 16, ends up in box $B_4$
guided by the ranges associated with the active boxes $B_2$, $B_3$ and $B_4$.

From the example it can be easily seen that the different functions $r$, $f$ and $m$
end up in the trails for the various boxes. For each box lineage can be seen in the
predecessor trails, which grows every step by extending the lineage information
of the box being operated on.

### 3.3.5 Localisation

A client can query the Armada by sending its query to one of the Armada’s sites
($S_1$, $S_2$ or $S_3$). Multiple boxes can be hosted on a single site, hence sites have
access to all of the trails that belong to the boxes they host. A query directed
to a site, can then be evaluated by the site to see if it can (partially) handle the
query. Based on the functions present in the trails, data coverage and query
span can be evaluated. As a result of the administration of predecessors and
successors in the meta-data, a hint can be given into the right direction if (parts
of) the query cannot be handled.

Successful and efficient localisation of the box(es) that potentially hold the
requested data is a vital prerequisite to allow query execution on an Armada sys-
tem. Using the previous example, we now briefly sketch that the lineage trails
provide sufficient information to find the responsible box(es) for the requested
data.

Note that when clients contact the Armada, they are contacting one (or
more) of its sites that host boxes, not the boxes themselves. The example from
Figure 3.1c describes 5 boxes that are in fact hosted on 3 sites, $S_1$, $S_2$ and $S_3$.

**Point Query** Suppose a client $c$ has a query which is answered by $\_\_$, say
42. $c$ can now contact any of the sites from the Armada. Any site that cannot
handle the request by $c$, redirects $c$ to the site that it knows has more specific
information. The simplest case is when $c$ connects directly to $S_3$. On $S_3$, only
trail $T_4$ is available. This trail defines the box responsible for the data fragment
$(12 \ldots \infty)$. There are no successors for $B_4$ available, meaning $B_4$ is active. Trail
$T_4$ tells that the query for $\_\_$ can be answered. In our example this means that
$S_3$ can tell $c$ that there is no 42 in the Armada.

In case $c$ connects to $S_1$, $S_1$ has three trails at its disposal: $T_0$, $T_1$ and $T_3$,
where $T_3$ is the most “specific” trail. Evaluating from that trail, $c$’s query cannot
be answered, hence a redirect to the predecessor box has to be made. (There are
no successors to consider for $T_3$.) Since the predecessor box $T_1$ is on the same
site, the redirection can be done internally, resulting in no client redirection. Evaluating $T_1$, $c$’s query can be answered, but since box $B_1$ is no longer active, it must be answered by one of its successors. In this case by successor $T_4$, which is located on site $S_3$. Hence, a redirect to $c$ for site $S_3$ is sent. As obvious from the previous case, at $S_3$, $c$ retrieves the answer to its query.

Finally, $c$ can decide to connect to $S_2$. At $S_2$, the trail $T_2$ is available. This trail does not cover the query $\_\_\_\_\_$, so neither would its successors do, if any. Hence, a redirect to the predecessor box is sent. This box, the origin $B_o$, is located on $S_1$. Since $S_1$ does not (have to) know that $c$ was redirected for box $B_o$, it just evaluates $c$’s query like it did in the case above, with the same result.

**Range Query** So far we only considered a query which was fully contained in a single box: the lookup of the value 42. Instead of this point query, a range query could be issued by $c$, that possibly spans multiple boxes. Consider query $\_\_\_\_\_$ which describes a range $[10 \ldots 20]$. Like in the previous cases described, client $c$ ends up at sites $S_1$ and $S_3$. Both sites are able to return a partial answer to the query and an additional redirect in order to get the remainder of the answer. Here, the client has to deal with the data being spread over two sites.

It must be noted that for this example we chose to have three different physical sites. This is merely for explanatory purposes. It is very well possible for every box to be on its own site, or for all boxes to be on the same site. There are no inherent restrictions in the Armada model as to where boxes are hosted.
Relations  The lineage trails from the Armada model, resemble the relations between the sites and the function coverages. Every step from the predecessor and successor trails at a site contains a site and a function. The example Armada from Figure 3.1c, can be depicted as in Figure 3.2. In the latter figure, only sites are shown, and their relations to other sites, represented by arrows. Next to the relations between the sites, Figure 3.2 also depicts (chunk) functions as a range in a finite space, for ease of understanding. The shaded area at $S_1$ represents the range covered by functions of no longer active boxes, the black area the current range covered. The outward arrows from $S_1$ show the function coverage at the head of the two arrows. As can be seen from the figure, the black areas in the ranges of $S_1$ and the outward arrows together cover the full range. This is due to each operation being performed with $S_1$ involved. That this is not the case for $S_2$ and $S_3$ can be seen from their arrows back to $S_1$. From the trail available at $S_2$, $T_2$, it is only known what the local function is, $f_1$, and what the predecessor function is, %. As a result, $S_2$ does not know about $f'_1$, hence when a request falls outside of its own function coverage, all it knows is that its predecessor (the origin) contains “all” data. Of course it never redirects for the data that is locally covered, as indicated by the striped area in the figure. The same holds for $S_3$, where $T_4$ is available. It has two steps in the predecessor trail up to the origin. Next to the origin itself, like for $S_2$ the step from $T_1$ contains the function $f'_1$. In a redirect, the site can use both to determine which site comes closest to what data is requested, which in the example happens to be the same site, but not necessarily has to be in larger Armadas. In Chapter 6 redirection strategies based on the functions available are discussed in more detail.

For every site, the whole Armada can be constructed by combining the functions from all outgoing arrows with the local function. This is easily deduced from the range representation in the figure, since combining all the ranges result in full coverage of the entire space. This combination is used in query resolving and composed of the local and remote functions as known from the local, predecessor and successor trails. When a query is executed at a site, it is executed using this combination. Since each site has its own combination of functions which represents the same — the whole Armada — actual execution may differ from site to site, but the final outcome is the same.

3.4 Lineage Wrecks

Eventually, each Armada that has a form of continuous growth becomes large. In principle, such large tree is no problem, however while data can be moved
Figure 3.3: The same Armada tree, in (a) represented as lineage tree and in (b) represented as association tree. (c) represents the association tree after $S_1$ has been removed, (d) after $S_4$ has been removed.

\[
T_0 = \{ \% , S_1 \} : B_0 ; \{ f^1 , S_1 \} : B_1 ; \{ f^2 , S_2 \} : B_2 ; \{ f^3 , S_3 \} : B_3 ; \{ g^1 , S_4 \} : B_4 ; \{ g^2 , S_5 \} : B_5 ; \{ h^1 , S_6 \} : B_6 ; \{ h^2 , S_7 \} : B_7 ;
\]

Figure 3.4: The Lineage Trails for the Armada tree depicted in Figure 3.3a.

to release sites from their data load, the sites themselves cannot be removed. The lineage trails in the tree in this respect “claim” each site to avoid a gap in the routing scheme, thereby preventing their removal.

The Armada model defines lineage trails to be immutable, hence simply “updating” their contents is impossible. Instead of reconsidering this immutability, we simply define additions to the lineage administration, which are in line with the original model. These additions take the form of an extra trail, which we refer to as jump trail. Jump trails are trails not pointing to direct predecessors or successors, but to any other box in the system. By carefully adding such jump trails, boxes in the lineage can be bypassed, thereby rendering them obsolete. Note that trails are never removed, hence references to sites remain to exist for the life-time of the Armada.
In Figure 3.3a and 3.4 an example Armada is depicted by its lineage tree and trails respectively. In addition to the lineage tree, an association tree is shown for the same Armada in Figure 3.3b. The example describes a chunk-only situation, where $S_1$ and $S_4$ became empty after they were chunked. This is in contrast to previous examples where one part of the chunk operation remains on the original site. Because $S_1$ and $S_4$ are empty now, they only send redirects to their offspring when consulted. In the situation of the example, both sites have no other function than redirecting, as they store no data.

The sites $S_1$ and $S_4$ are due to their limited use good candidates to be removed from the Armada. In fact, a planned removal could be the reason for chunking to two new sites. While a site cannot be “removed” from the Armada, it can become unavailable with no intentions to return. While the site remains to be referenced in the trails of the Armada, the information stored at the site remains necessary for the Armada as a whole. Hence, a ship that becomes a wreck — a site that purposely is removed from the Armada — hands over its lineage information to another site in the Armada. The most obvious candidates for this are the predecessor sites for all hosted boxes. In the example, sites are not reused, hence there is only one predecessor site for each site, except the origin.

**Predecessor Propagation** For site $S_4$ to become permanently unavailable, it has to transfer its unique knowledge to its predecessor. The unique knowledge is the case of $S_4$ the successor steps to $S_6$ and $S_7$, as specifically can be seen in Figure 3.3b. Without trails pointing to those sites in the Armada, others cannot reach them, while $S_6$ and $S_7$ themselves could still reach the rest through $S_1$, which they inherited in their predecessor trails. The jump trails to add to $S_1$ resemble $T_4$. With those specific trails, $S_1$ needs not to redirect to $S_4$ any more, since with the jump trails added it has access to more specific trails that supersede the original one for $S_4$, as depicted in Figure 3.3d.

**Successor Propagation** In case of $S_1$, making the site unavailable is not as simple as described before, since there is no predecessor to hand over the unique knowledge to. A conceptually simple solution would be to assign the trails of $S_1$ to another node in the system. This node would become the origin, and the whole system could continue to work as before, using the new origin. However, even though this sounds trivial and simple, it means all lineage trails in the entire Armada need to be changed in order to reflect this change, since each trail contains the full lineage up to the origin. Apart from being a very
expensive operation, this can only be done by adding extra trails, which in this case are not more specific. For both reasons, this solution is deemed not to be a viable solution. It may be evident that it cannot be overcome that lineage trails point to a no longer existing node, e.g. $S_1$.

Instead, in this special case, the unique knowledge has to be pushed down to the direct successors of $S_1$. Again, the idea here is to make $S_1$ obsolete by having more specific information available. Figure 3.3c depicts the situation where the trails from $S_1$ have been pushed down to all successors. If $S_6$ would send a redirect to $S_1$, this redirect points to a wreck. When this is encountered, a search for surrounding sites using the lineage trails is done, in this case ending up at $S_4$. The latter site is then able to redirect to the appropriate site to continue the search.

3.5 Related Research

Close to Armada’s objectives is Mariposa [65]. This system which we briefly mentioned in Chapter 2 aims next to its economical decisions for a distributed setting based on fragments of data among autonomous systems. Unlike the envisaged client interaction in Armada, Mariposa passes queries or data on to other sites it knows on behalf of the actual client, resulting in a chain of dependent systems representing the economical broker structure. Further on, location of fragments is not really specified and forms part of the bidding process, whereas Armada has this embedded in its lineage trails. The lineage information in Mariposa is used mainly for merging back previously split fragments. Armada on the other hand, allows merging of any two or more boxes. While Armada does not explicitly deal with data mobility, heterogeneous host capabilities and a simple language that controls actions done on the data, it does not outlaw their use. In fact, the Mariposa rule system defines action routines that map on the clone, chunk and combine operations, and the related fragmentation functions.

Stream Databases In recent years, two research trends in distributed databases have emerged. The first are sensor network databases are characterised by a large number of resource limited receptors at the edge of a network to collect mission critical data. Prototypical building blocks are small ‘Motes’, a single-board-computer (SBC) equipped with limited memory, limited network capabilities and limited energy, glued together to form a distributed information system to feed the upstream applications. On each site, we find one or
more sensors and an embedded SQL database engine for storage management and query processing [48, 21, 5, 2]. However, their underlying architectures ignore autonomy as we aim for with Armada. In essence, they are built from functionally scaled-down versions of relational database systems.

**P2P Systems** Second, Peer-to-Peer (P2P) systems have gained a lot of interest. A comparison of Armada with such P2P systems is inevitable, since both systems are decentralised using highly autonomous participating nodes. The focus of P2P systems is efficient query routing and localisation [56, 52], yielding in a routing-centric view. Armada differentiates from this approach in having a data-centric view: the data, in terms of boxes, filled with relations are aimed at evolutionary growth starting from a single node. This different point of view yields in some elementary differences between Armada and P2P systems. These differences are all related to the way data is distributed over the system. P2P techniques assume the data is already in place and numerous, usually in the form of files. In general the placement is not bound to any rule, and usually simply on the machine that provides the data on the network. Replication of that data is a side-effect of other machines that copy the data and make it available afterwards. P2P systems in general do not make any efforts to manage the data that is in the system. Instead, they focus on how to find this data in the network, using a key-based approach, where each data item is assigned a key used for lookup. Here, efforts are made to have this lookup structure being fast and resistant to failing nodes.

Unlike P2P systems, Armada has functions that define how data is split over a number of boxes, which allow for concise localisation of data. With Armada, the focus is on the data being stored. Using a schema-based approach, instead of dividing the key-space Armada divides the data-space over multiple machines. Here, the data is split into parts based on a machine local function, that suits best for the scenario at hand. Load, capacity or redundancy problems on a local machine are the trigger within Armada to split the data to another node. Here it must be stressed that the split function, is local to the machine unlike in P2P systems which use e.g. a predefined hash-function for the whole system. The latter of course, only deals with the key-values, not the data itself. As a result, Armada nodes are able to resolve local overflow situations by taking an autonomous decision to split their data and move a part to another node. The index to find the data, hence is not based on some key but the value itself, the data. Because keys in a DHT system are generated by a hash-function, data properties are lost in the key representation. This is not a problem for point
queries which look for a certain (upfront known) key, but it does complicate efficient range queries, that target the data values. Querying key ranges, which is not comparable to data value ranges, needs help to simulate range support such as in the Distributed Segment Tree [74].

A P2P system eventually never searches in the data value space. Instead, it solely looks at key values. In this regard, Armada is more suitable for traditional database use, as those databases work with the data values as well. P2P database systems, such as PIER [37] and PeerDB [55], hence only concatenate database systems or data sources. They simply lookup the appropriate sources and apply the queries on those. A fragmentation of a single data source, in a distributed manner such as in Armada, is not being dealt with. P2P systems that would assign a key to each data item, e.g. a database tuple, assume that a query in such system exactly knows what it wants to retrieve — which defeats purpose of the query.

In more detail, the objectives of PIER [37], a P2P database system, differ from Armada in what is provided by the convergence — eventual consistency as also found in epidemic-based systems [70] — of the Armada model. PIER focuses on massive distribution to validate scalability and distribution. Traditional ACID properties are relaxed because that is a necessity [26], but unlike PIER, Armada maintains the global schema requirement for the data. Another P2P related structure, BATON [38], is a tree shaped P2P overlay network. Whereas BATON is a balanced binary tree, Armada uses generic (heterogeneous) functions and needs not necessarily to be balanced. In fact, node relocations in the tree are not supported in Armada, because the tree is built out of the lineage relation between the nodes, that cannot possibly change in the same way that history is never rewritten.

**SDDS** Scalable distributed data structures (SDDS), a predecessor of P2P systems, use globally known, but locally adaptive partitioning functions [45, 42]. Also the client behaviour in SDDS implementations bears some similarity with the Armada approach. They manage a cache with metadata to direct data lookups. The main difference with the Armada vision is its level of abstraction. SDDS solutions are focused on single key-based retrieval. In our model, we extend the scope to the complete functionality of a database system. Furthermore, the lineage trails capture the complete history of a box, something not considered in an SDDS. It maintains the latest, locally consistent distribution status.
Self-management Over their life span, database systems experience a continuous change (usually growth) of the amount of data stored. Likewise, usage patterns and workloads keep on changing. For example, more recent data is often accessed more frequently than older data, creating a “continuously moving access hotspot”. Classical distributed database architectures hardly provide any means to adapt to these changes automatically. Evolution techniques are mainly based on local conditions. Rather, increasing the system’s capacity (by adding additional nodes) and re-distributing the data to balance the load are measures that have to be initiated and executed by some human DBA [68]. Additionally, client/server settings form the base of dealing with the work, where servers perform the entire job of query execution as “service” for the client. The result is a reduced autonomy of servers from an implied work point of view. Servers have to go through the full execution, instead of only the part they are responsible for.

The area of self-managing and self-tuning databases limits itself by only advising the DBA [57, 75] or only dealing with indices and materialised views [3] — the metadata. Combinations of replication and fragmentation are not supported, and only on the whole table data, where fragmentation is only horizontally applied. Armada, on the other hand, can be considered a self-adaptive model to meet the environment requirements and reconfigure when they change. A compatible vision can be found in distributed systems, where decentralisation is the key as well [70].

Summary

The Armada model is a schema based solution to distribution. The control over distribution parameters is set by functions that divide the data into (smaller) pieces. The function can be freely chosen by the site that performs the operation, leading to ultimate autonomy for that site. Not only can it decide when, but it can also decide how to perform the operation, thereby supporting incremental scalability.

Each function that is applied is recorded in trails that track how pieces of data in the Armada have evolved. These trails are stored decentralised in such a way that localisation of the pieces is possible from any site in the Armada. Sites that are unavailable are not “forgotten”, but instead remain present in the trails, leading to a consistent image; data does not suddenly appear or disappear, it is or was known to be missing instead.

To retain the autonomy of the sites in an Armada, sites do not perform work
for others. Instead they send redirects or refuse to work. This requires an interaction shift from passive to active, where the client is expecting to follow redirects and deal with the structure of the Armada to resolve a query.

A big contrast to other systems is that Armada uses a data centric view on distribution and the arbitrary functions that follow out of that. The high value for autonomy in the system, to make it self managing, is not to be found in most other systems. The self management for the cluster as a whole that Armada aims for, goes beyond the single (sub-)system level and enables a continuously evolving system.