12. GI-Fachtagung für Datenbanksysteme in
Business, Technologie und Web (BTW 2007)
5. bis 9. März 2007 - Aachen, Germany
http://www.btw2007.de/

Armada: a Reference Model for an Evolving Database System

Fabian Groffen Martin Kersten Stefan Manegold
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
{Fabian.Groffen,Martin.Kersten,Stefan.Manegold}@cwi.nl

Abstract: The data on the web, in digital libraries, in scientific repositories, etc. continues to grow at an increasing rate. Distribution is a key solution to overcome this data explosion. However, existing solutions are mostly based on architectures with a single point of failure.

In this paper, we present *Armada*, a model for a database architecture to handle large data volumes. Armada assumes autonomy of sites, allowing for a decentralised setup, where systems can largely work independently. Furthermore, a novel administration schema in Armada, based on *lineage trails*, allows for flexible adaptation to the (query) work load in highly dynamic environments. The lineage trails capture the metadata and its history. They form the basis to direct updates to the proper sites, to break queries into multi-stage plans, and to provide a reference point for site consistency. The lineage trails are managed in a purely distributed fashion, each Armada site is responsible for their persistency and long term availability. They provide a minimal, but sufficient basis to handle all distributed query processing tasks.

The analysis of the Armada reference architecture depicts a path for innovative research at many levels of a DBMS. Challenging many conventional database assumptions and theories, it will eventually allow large databases to continue to grow and stay flexible.

1 Introduction

Soon we face a common repository size scaling into petabytes, filled with data that needs to be archived and processed. The rapidly improving technology cannot keep up with the data growth rate, hence data processing becomes more and more an expensive and time-consuming task. This problem is of major concern, since data processing is a core process for many businesses and applications. Yet a real solution to the data growth problem has to be found.

Scaling into multiple machines to process the data is currently successfully applied in grids and distributed databases. However, the centralised scaling technique using a large number of machines, is *fragile* from an availability point of view. All systems depend on the availability of one. Moreover, this single point-of-failure can easily get overloaded, thereby forming the bottleneck in serving a high workload.

The problem does not limit itself to large scale machines such as datagrid environments

in scientific settings [LF04]. Also mobile and ambient settings deal with a data explosion problem. The ambient environment consists of a potentially large number of database-empowered sensory systems, which learn and exchange information to reach a common goal, e.g., increased experience without computers in sight [A^+03]. Mobile environments are characterised by a large number of clients sharing information through multiple data brokers. The data 'follows' the device, which may be offline for lengthy periods [D^+97].

Although in each environment the complexity can be controlled within the context of a single application and strict separation of roles, current distributed database offerings stem from an era where a limited number of always-on servers were prevalent. They lack functionality in a number of areas to provide a general solution. A novel reference architecture for a distributed database is urgently needed.

It should take site autonomy and volatility as both driving force and core feature of a system architecture. A sole central broker to guide all interactions is a *dead end* for the scalable solutions required. Instead, several sites may take such a role for a limited period and only for part of the data space. To co-ordinate their efforts, 'contractual' arrangements and economic models are needed. They should facilitate a 'data market' to ensure the desired system behaviour and be flexible to cope with temporal and evolutionary changes.

Of course, the setting is not completely new. It is the natural step forward in the area of highly distributed database technology. The underlying techniques are still based on *data fragmentation* and *data replication* to break the database into manageable portions, and query shipping versus data shipping for efficiency [OV99]. However, the volatile setting calls for better solutions to keep track of the data whereabouts, their status, and their lineage in the grand scheme. A portion of the database may be broken into pieces and migrated to autonomous sites with little control other than powerful re-conciliation algorithms when the pieces are fused in the future. The long liveness of such networks makes legacy of information, e.g., out of date schemas and queries, a ground rule rather than an exception. It calls for data management schemes optimised for incomplete and only partially consistent information.

Within the *Armada* project we study the building blocks for an organic database. The main contribution of this paper is a visionary description of the *Armada* model and its architecture, designed to facilitate evolutionary growth in a distributed environment. It uses data fragmentation, data replication and data fusion as the minimal basis for the lineage of data blocks, that allows maximal autonomy of the nodes co-operating in a distributed application.

Sites can easily join an Armada alliance by donating resources and taking responsibility for a portion of the data space prescribed in the database schema. They can also leave the alliance with minimal detrimental effect on its environment. The real size of the distributed system and data locality is largely hidden from individual nodes. The *Armada* administration allows for localisation of data without need for a central entity that becomes a bottleneck, single point-of-failure and hot-spot in busy systems.

The clients (applications) are put back into the loop to steer query processing and distributed transaction management. Common client policies can be captured in scenarios managed by a middleware software layer, but the autonomy of the Armada sites ultimately

relies on co-operative clients as well.

The remainder of the paper is structured as follows. Section 2 introduces the Armada model, its notation and operations. The realisation of the model in terms of an architectural overview is presented in Section 3. The effect of lineage trails on query processing is described in Section 4. A sample embedding of the Armada model in the trends in distributed database systems is indicated in Section 5. We conclude and give a short outlook in Section 6.

2 The Armada Model

The focus of this work is to create a reference model for a flexible, self-maintaining, efficient distributed database architecture. To achieve this goal, we try to avoid the classical bottlenecks that limit the efficiency of most existing and proposed architectures. These bottlenecks can be seen as the two extreme alternatives of storing and maintaining the metadata that is necessary to ensure correct and efficient handling of the actual data. Classical designs on the one end of the spectrum require a centralised server that holds all metadata, and hence forms a hotspot. The central server is accessed to lookup or update metadata for both operations that query/update the actual data and operations that change the structure of the system at large. The latter involves addition or removal of nodes and/or reorganisation of the data for load balancing purposes. It creates a bottleneck that limits the overall performance and scalability of such systems.

Designs on the opposite end of the spectrum avoid this hotspot by fully replicating all metadata. Such designs have to rely on the consistency of the replicated metadata, and hence, each structural change requires the (synchronised) update on all nodes in the system. Because all metadata is available locally, data operations are cheaper, but it significantly increases the price of structural operations, prohibiting efficient dynamic changes of the data distribution.

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, i.e., 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.

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, each captain does not only 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 [1].

In the remainder of this section, we will briefly formalise the key components of the Armada model. We start by introducing the basic notation, terms, and definitions that make up the 'static' part of the Armada model, i.e., the part that is used to describe how the metadata is represented. After that, we proceed with the 'dynamic' part that models operations to perform structural changes on the Armada.

The goal of this work is to establish the Armada model as a generic framework for distributed database architectures. Hence, discussion of actual instantiations of the model, like strategies as to when, why and how to perform structural changes, is beyond the scope of this paper.

2.1 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 will provide some constraints for such functions. Section 3.2 will discuss 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*

[1] The trail administration for each box is only valid at the time it is created. Afterwards, its references to successors may be outdated. For the site hosting the box this does mean, however, that it can reach the rest of the Armada through the sites it knows as stored in the trails, even though that might not be the most up-to-date state.

require that

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

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 application 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* $s_l = [\, g\ ,\ S\ ]{:}B$ *, 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'', $T' = T''.[\, g'\ ,\ S'\ ]{:}B'$ and $T = T'.[\, g\ ,\ S\ ]{:}B$, respectively. We call*

T''	*a* predecessor trail *of box* B',
$s' = [\, g'\ ,\ S'\ ]{:}B'$	*the* local step *of box* B',
$T' = T''.s'$	*a* local trail *of box* B',
$s = [\, g\ ,\ S\ ]{:}B$	*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 predecessor 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 will 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:

$$
\begin{array}{rr|l|l}
 & pre & loc & suc \\
\hline
T'' = & T''' \,. & [\,g'', S''\,]{:}B''\;; & [\,g', S'\,]{:}B' \\
T' = & T'' \,. & [\,g', S'\,]{:}B'\;; & [\,g\;, S\;]{:}B \\
T = & T' \,. & [\,g\;, S\;]{:}B\;; &
\end{array}
$$

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 B_o's local step:*

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

2.2 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:

$$
\begin{array}{rl}
T_o = & [\,\%\,, S_1\,]{:}B_o\;; \begin{cases} [\,r\;, S_1\,]{:}B_1 \\ [\,r\;, S_2\,]{:}B_2 \end{cases} \\
T_1 = & [\,\%\,, S_1\,]{:}B_o\,.\,[\,r\;, S_1\,]{:}B_1\;; \\
T_2 = & [\,\%\,, S_1\,]{:}B_o\,.\,[\,r\;, S_2\,]{:}B_2\;;
\end{array}
$$

In this example, the origin has two successors, B_1 and B_2, which themselves have no successors.

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:*

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

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:

$$\begin{aligned} T_o &= \quad [\,\%\,,S_1]{:}B_o\,;\begin{cases}[\,f\;,S_1]{:}B_1\\ [\,f',S_2]{:}B_2\end{cases}\\ T_1 &= [\,\%\,,S_1]{:}B_o\,.\,[\,f\;,S_1]{:}B_1\,;\\ T_2 &= [\,\%\,,S_1]{:}B_o\,.\,[\,f',S_2]{:}B_2\,;\end{aligned}$$

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

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., reconstructing 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 the a new box (B_9), creating a duplicate free combination of the inputs' data.

[2] We felt free to 'invent' this verb.

$$
\begin{array}{ll}
T_4 = & T_3 \,.\, [\,r\;, S_1\,]{:}B_4\,;\;\; [\,m\,, S_1\,]{:}B_9 \\
T_6 = & T_2 \,.\, [\,f'', S_2\,]{:}B_6\,;\;\; [\,m\,, S_1\,]{:}B_9 \\
T_9 = & \left.\begin{array}{l} T_4 \\ T_6 \end{array}\right\} .\, [\,m\,, S_1\,]{:}B_9\,;
\end{array}
$$

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

2.3 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.

$$
\begin{array}{lrl}
T_o = & [\,\%\,, S_1\,]{:}B_o\,; & \left\{\begin{array}{l} [\,f_1\,, S_1\,]{:}B_1 \\ [\,f_1'\,, S_2\,]{:}B_2 \end{array}\right. \\
T_1 = & T_o \,.\, [\,f_1', S_1\,]{:}B_1\,; & \left\{\begin{array}{l} [\,f_2\,, S_1\,]{:}B_3 \\ [\,f_2'\,, S_3\,]{:}B_4 \end{array}\right. \\
T_2 = & T_o \,.\, [\,f_1, S_2\,]{:}B_2\,; & \\
T_3 = & T_1 \,.\, [\,f_2, S_1\,]{:}B_3\,; & \\
T_4 = & T_1 \,.\, [\,f_2', S_3\,]{:}B_4\,; &
\end{array}
$$

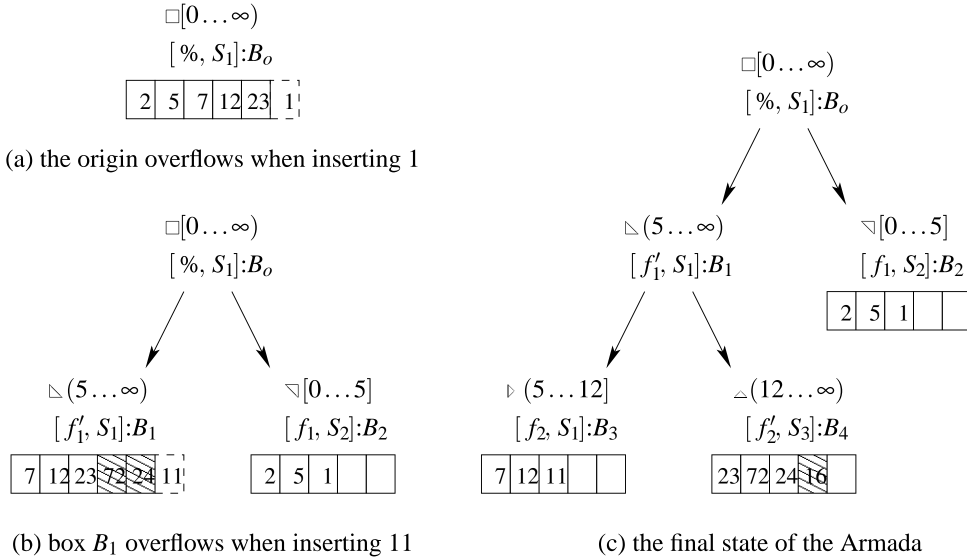

Figure 1: Sample Armada with 5 boxes.

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 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_o. The other symbols; '◹', '◺', '▹' and '▵' 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 1 is built over time by inserting data into the Armada. In the initial situation, depicted in Figure 1a, only B_o 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:

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

Since there only fit five integers in each box, the origin B_o consists of $D(B_o) = \{2, 5, 7, 12, 23\}$ 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 1b, the state of the Armada after the first chunk operation is depicted. As can be seen, the data from the origin box B_o 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_o is not active any more, and will redirect 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 [3]. 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 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.

2.4 Localisation

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 system. Using the previous example, we will briefly sketch that the lineage trails provide sufficient information

[3] A more detailed description of how this redirection is decided upon is given in Section 2.4.

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 1c describes 5 boxes that are in fact hosted on 3 sites, S_1, S_2 and S_3.

Suppose a client c has a query which is answered by $\triangle$, say 42. c can now contact any of the sites from the Armada. Any site that cannot handle the request by c, will redirect it 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 S_3 available, meaning S_3 is active. Trail T_4 tells that the query for $\triangle$ 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_o, 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 $\triangle$, 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.

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 $\triangle$ which describes a range $[10\ldots20]$. Like in the previous cases described, client c will end up at sites S_1 and S_3. Both sites will be 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 choose 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.

3 Architecture Overview

In this section, we illustrate the formation of an Armada, the role of the database schema, and decisions taken by the sites regarding responsibilities for data management.

3.1 The Alliance

A collection of sites $\mathcal{S}$ provides the context to distribute the database. At any time, only a subset of $\mathcal{S}$ is actually involved in the Armada, i.e., those sites containing boxes and associated lineage trails. The subset of participating sites is called the *Armada alliance*, or $\mathcal{A} \subseteq \mathcal{S}$. The starting point for an alliance is a single site $\mathcal{A} = \{S_o\}$, i.e., the origin.

The alliance $\mathcal{A}$ should be extended before a clone, chunk, or combine operation can deposit the result boxes at a new site. A site Y can only be invited to join the alliance if a member of the alliance is willing to co-operate with Y. For that, Y should adhere to the Armada's *Code of Conduct*:

DEF. 9 *A site Y becomes a member of the Armada alliance $\mathcal{A}$ iff*

- *it is nominated by an existing member,*
- *it donates resources to manage boxes,*
- *it keeps a permanent record of its lineage trails,*
- *it co-operates and faithfully answers queries,*
- *its existence may be published to other members.*

Admission of sites is broadcasted to all members asynchronously or on a need-to-know basis. Due to update propagation delays, at any time a member only knows a portion of the alliance $\mathcal{A}$ by inspection of the lineage trails it receives together with new boxes.

A technical issue is to publish the site identities S of possible new members. In line with the dynamic nature envisioned for Armada, every site X knows only a fraction $\mathcal{S}_X \subseteq \mathcal{A}$ and should be told about possibly new members explicitly by intervention from an outside authority [4].

Selection of candidate sites to join the Armada is initiated by a member when it can no longer fully co-operate due to resource constraints. Given the Code of Conduct, an open call is issued to sites $\mathcal{S}_X$ for bidding on solving a quantified resource problem, e.g., lack of storage or CPU units.

DEF. 10 *A bid*$(X,\text{CPU},\text{MEMORY})$ *is an operation executed by site $X \in \mathcal{S}$ and returns* $\{t \in \mathbb{R} | 0 \leq t < 1\}$*, a positive real number representing the value (eagerness) of X to participate with a minimum of* CPU *and* MEMORY *units.*

DEF. 11 *Let $X \in \mathcal{A}$ be a site with a wish to offload (*CPU, MEMORY*) units. It issues bid requests to $\mathcal{S}' \subset \mathcal{S}$ and grants the bid from site $Y \in \mathcal{S}'$ which supplies the most satisfying bid.*

Calling for additional help by a site $X \in \mathcal{A}$ may lead to a situation that no other site $Y \in \mathcal{S}$ is willing or able to make resources available, effectively passing back the problem to the site X. This failure cannot be resolved at X. Instead, the process of finding, negotiating

[4] New members can be searched using a client application or could be hardwired in the implementation, e.g., using an IP-range.

and bidding is then pushed back to the client by rejecting queries due to resource overload. The client could attempt alternative sites to receive the attention needed, or should contact an outside authority to increase the basis from which Armada members are recruited. In our analogy, any Armada is limited in the ships it can deploy. An extension requires a governmental approval (and new taxation).

3.2 Chunk Functions

Extending the Armada (or offloading work to others) is grounded in the ability to fragment and replicate portions of the database. The Armada model captures the offloaded work in the lineage trail as the clone, combine and chunk functions. The operations are precisely administered, such that at any time the lineage trail can be re-interpreted to assess the past decisions.

The chunk function should satisfy the correctness criteria for distributed relational systems [OV99]: the function f is lossless, i.e., each possible data element can be associated with either box B^f or $B^{f'}$, it should designate disjoint portions $B^f \cap B^{f'} = \emptyset$, and the original box can be reconstructed from its components.

Note that a chunk function can be generalised to partition a space into multiple disjoint components. This way it encompasses all known techniques from physical distributed database design. The function f could be a simple hash or range distribution function, which derives the destination boxes based on the key value. Scalable distributed data structures have been developed to support the evolutionary growth as well [L$^+$04]. Alternatively, the function f is a deterministic algorithm, solely based on time and location invariant data properties.

It should be stressed that the nature of the chunk function can be decided upon at each site autonomously and it may differ for each box being considered. The consequence is that chunking or cloning a box leads to a local datamanagement optima, ignoring the goal of an Armada at large to form a coherent and effective distributed system. Autonomy in this respect calls for a brokerage service, e.g., a client application, to mediate between sites to balance their tasks.

3.3 Box Updates

A client application c interacts with the Armada sites on a one-by-one basis. It is told the identity of at least one Armada site X using publicly known information or through an authoritative outsider. There is no a priori relationship; a client may pick any member from the alliance.

The client c can issue updates to the Armada using site X as a starting point. If X contains the boxes holding the data of interest, updates follow the traditional local database patterns. However, if the lineage trails carried at X denote existence of cloned versions or

the updates have effect on remote boxes, it tells the client. Unlike traditional distributed systems, it does not mediate directly in propagation of the update requests.

For example, for cloned boxes it returns a list of sites the client should contact to ensure global consistency in due time. To implement this policy any of the known database update replication schemes can be used. They can even be implemented with a few a priori defined Armada agents, who take over the role of the clients' responsibility.

Beware that this client-Armada relationship is built on mutual trust and persistency. If the client forfeits its duty to forward updates to related sites, the outcome could be an inconsistent database. It does not render the Armada useless, but may affect local decisions taken in the future. It mimics reality where decisions are mostly based on locally consistent information only.

3.4 Armada Heterogeneity

A real Armada consists of different types of ships. To name a few famous Dutch kinds, the *galjoten*, *hoekers* and *spiegelschepen* [5] all have different capabilities on storage, speed or defence. As such, different ships have different functionalities and responsibilities within the Armada. Similarly, the sites in our Armada, can be of different types. Not only their resources differ, but also their connection to the rest of the Armada, and the kind of data storage engine they run. Actual storage may be done in a flat (log) file, or in an SQL database such as PostgreSQL, IBM DB2, Oracle, MySQL, etc., which gives different properties to the sites.

The analogy goes further when it concerns the data boxes. Much like ships carry both barrels, boxes, and crates, an Armada site carries data boxes of different flavours. Some might be designed to store particular data items, e.g., blobs, multimedia images or structured documents, while others are organised around the physical boundary conditions, e.g., disk block sizes. In all situations, the amount of lineage trail information is considered small compared to the box itself. Thereby avoiding a bureaucracy.

4 Query Processing

The loose affiliation and strong autonomy of sites, combined with the pivotal role of the client calls for a fresh look on distributed query processing. In such a vision, any form of system induced centralised control over query execution is ideally removed.

In this section, we describe the mapping of the Armada reference model to a relational context and illustrate the challenges for query processing.

[5] See `http://www.holland.com/voc/gb/fleet/ships/index.html` for more information on these ships.

4.1 Relational Lineage Trails

The Armada reference model does not a priori prescribe the data model and query language. However, once we deploy it in the context of a real application setting, it has to be fixed to delimit the scope. The first refinement of the Armada model is geared towards relational systems, which calls for a redefinition of boxes and lineage trails.

DEF. 12 *A* relational box B_i *in a lineage trail* T_i *is a box whose content is covered by the relational schema DB attached to the origin site* $DB = schema(origin(T_i))$.

The data in the box should satisfy traditional key and domain constraints. However, the role of referential and table constraints should be reconsidered in the light of the Armada autonomy. A discussion on this topic is left out of this paper for space reasons.

DEF. 13 *A* relational lineage trail *is an Armada lineage trail* T_i *whose chunk, clone, and combine functions* f, r, m *are limited to relational algebra queries over the schema DB attached to the origin site,* $DB = schema(origin(T_i))$.

This definition emphasises the role of the origin site. Its schema determines the scope of the data space managed. All boxes managed by the Armada can be phrased as relational queries, but care should be taken to limit the expressiveness of the query language to also ensure a lossless Armada. The chunk function f is a simple SELECT-FROM-WHERE query, such that the key attributes are retained in the derived boxes.

A relational lineage trail can be seen as a small snapshot of a relational catalog. It describes the data retained in boxes in terms of a compound view over the origin schema. Furthermore, the relational trails contain descriptions of database views once managed at remote sites. It forms a roadmap for referral queries and decomposition into a distributed query.

4.2 Single Box Queries

Finding a box with data of interest in the Armada remains the most important query. However, unlike P2P schemes, the whereabouts of the relevant box are not administered centrally; its location may even frequently change. [6]

A client can send a query Q to any participating site where it can be validated against the database schema for correctness using any of the relational trails. Subsequently, the query is replaced by a union-query $Q = Q_0 \cup \cdots \cup Q_k$ such that term Q_i represents a sub-query to be solved by site S_i in the Armada. Splitting is based on all lineage trails known locally. Algorithmically it requires a search for the union query with all known sites holding boxes

[6] We consider the hash function in a Chord to be globally known, hence based on a central copy of this function.

of interest. Given the nature of the successor trails known at the boxes, a query might be broken up again when it turns out that the box it refers to was chunked afterwards.

Unlike traditional distributed databases, the subqueries $Q_i, \ldots, Q_n$ are not immediately forwarded to their destination site. Instead a query referral list $Ql = [(Q_i, S_i)]$ is built and sent back to the client for further consideration. To retrieve the answer the client should explicitly ask the sites for their result sets for the given query Q_i. It may iterate through the referral list, asking each site in turn to deliver it, or it may broadcast the complete list at once. This control also gives the opportunity to the client to abort query processing after each sub query issued. In all cases, the client is responsible to merge the results obtained and to deal with the interaction of the sites. Connections to sites may time out. Sites may appear to be unreachable, reject connections or tell they are too busy. In case of clones, this may even result in a redirect to one of the other clones. It is the client's task to prevent an endless loop to arise if both clones appear to be too busy to handle the request of the client.

As a remedy against unavailable or too busy sites, a client might inspect the lineage trails to see if there are any clones in the lineage. Finding a clone may result in getting the data from another 'branch' in the Armada, if available.

The autonomy of the Armada sites and its evolution complicate this scheme. Consider query Q_i arrived at site S_i for evaluation. Then a few cases should be considered.

1. The site S_i accepts the query and can handle it locally. A result set is prepared and shipped to the client.
2. The site S_i accepts the query, but produces a partial answer and an update for the query referral list.
3. The site S_i runs out of resources and is not able to respond to the query request. It returns the query to the client, which should decide on what to do. If the site knows about other sites that could possibly handle the query (partially), it also sends an alternative update for the referral list. Especially for clones this is a standard procedure.
4. The site S_i detects that the boxes of interest have been relocated. It sends a new query referral list back to the client.
5. The site S_i runs out of resources and decides to expand the Armada with new sites. The query is replaced by a new query referral list afterwards and sent back.
6. The site S_i breaks the connection after a partial result has been shipped. The client should re-submit the query.
7. The client breaks the connection with site S_i, which triggers a local transaction abort.

The scheme proposed shifts the burden of distributed query processing partly to the client. The rationale is that the client ultimately resolves conflicts, e.g., time to wait for an answer and 'money' to spend. An actual client implementation may be based on a library with a priori defined scenarios for dealing with the query referrals.

A tricky part is detection of duplicate results, for it may potentially call a very large local memory at the client side. The solution sought is based on keeping the lineage trails attached to the referral list. It can be used to identify boxes with duplicate information (clones). Judicious execution of the referral queries and early diversion of result sets that are known to hold duplicates are the tools for the client to deal with this problem.

4.3 Query Evaluation

Query optimisation within Armada takes on a different flavour as well. Known techniques for semantic and symbolic query optimisation still apply. However, reducing the amount of data shipped or minimising the response time cannot a priori be the prime target. When a task is taken from the referral list there is still no guarantee on the responsiveness of the site being addressed. Therefore, ruling out many query execution plans upfront, based on cost estimations, is not an option.

The approach taken is based on the rationale that subqueries can only be solved if the (partial) input data resides at a single site. It leads to the refinement of the query referral list into a dependency graph, which captures the processing dependencies. The client should obey these dependencies during query evaluation.

At each site, a subquery is evaluated and/or preparation steps are taken to bring boxes together for the next step. Preparation involves a decision on cloning, chunking and combining pieces at a site with ample resources available. It is a variation on our initial bidding process. For querying we are interested in temporary resources only. After the result has been produced and shipped to the client, the storage could become available for re-use.

The query bid request $ask(Q_i, T_i)$ involves the subquery Q_i and associated lineage trails T_i. It is sent to sites of interest for a quote on its fictive cost.

A site S can respond in different ways. It may accept the task and reserve resources for the duration of the query. Or, it may propose to initiate an Armada re-organisation first, e.g., cloning, chunking and combining the operands. And finally, it may simply opt out.

The effect at the client is that query evaluation becomes highly dynamic and unpredictable. Just-in-time decisions are taken on where data should be sent and what order of evaluation is most effective. The benefit is that the query plan can be stopped at any time to avoid spending resources on less interesting results.

Furthermore, conceptually the result sets remain at their site of origin until the client explicitly releases the resources. The same holds for all intermediate results. The global effect is that the Armada becomes polluted with temporary results. However, a site's autonomy will permit unilateral disposal, provided the box discarded can be reconstructed or a referral query can be issued to recover it from a dump site.

5 Related Research

Research on federated/distributed database architectures has a long history. All major DBMS suppliers provide technology to realise a distributed database. They are also optimised for a limited number of servers, e.g., running on a cluster computer composed of several tens of processors with a NAS service. Wide area distributed databases benefit from a plethora of publish/subscribe techniques, e.g., Oracle Streams [Tum04] and Microsoft's Message Queues [B+04b].

Close to Armada's objectives is Mariposa [S+94]. This system aims 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. Further on, location of fragments is not really specified, 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.

In recent years, two research trends in distributed databases have emerged: sensor network databases and P2P systems. 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 a limited memory, limited network capabilities, and limited energy, glued together to realise 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 [M+05, F+05, B+04a, A+05]. However, their underlying architectures ignore the autonomy target set for Armada. In essence, they are built from functionally scaled-down versions of relational database systems.

The focus of Peer-to-Peer systems is efficient query routing and localisation [P+04, MM02]. 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. P2P techniques assume the data is already in place and numerous, usually in the form of files, like in PIER [H+05]. Unlike P2P, Armada has functions that define how data is split over a number of boxes, which allow for concise localisation of data.

Scalable distributed data structures (SDDS), a predecessor of P2P systems, use globally known, but locally adaptive partitioning functions [L+04, KK00]. 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.

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 do

not provide any means to adapt to these changes automatically. Rather, increasing the systems 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 [OV99]. Additionally, client/server settings form the base of dealing with the work, thereby greatly reducing the autonomy of the participating servers.

The area of self-managing and self-tuning databases limits itself by only advising the DBA [R+02, Z+04] or only dealing with indices and materialised views [A+04] — 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.

6 Conclusions and Outlook

Emerging applications based on large numbers of autonomous systems challenge the assumptions of underlying traditional distributed database technology. The storage and processing requirements encountered are often modest compared to the servers on which commercial databases run. Instead, they stress the need for autonomy in managing a portion of the database in a co-operative or P2P setting. The volatility of devices joining and leaving the ensemble, calls for a fresh look on metadata management, query processing and transaction semantics.

In this paper, we introduced Armada, a reference model and system architecture for distributed datamanagement. The research methodology is purposely focused on the introduction of a concise model based on *lineage trails*. The exploratory description of the envisioned architecture charters a rich research landscape ahead. The analogy of a real-world Armada, a fleet of autonomous ships sailing under authoritative goal and charter, provides the necessary insight in alternative solutions herewith no-go areas for distributed database systems.

A simulator for the Armada model has been developed to experiment with large examples and study the effect of lineage trail management. Its next incarnation provides quantitative data on the robustness of the model against (deliberately) unavailability of physical sites.

Other priorities on the Armada research agenda include development of economic models to steer the interaction between clients and Armada sites and fleet formation reorganisation. Finally, a real Armada system implementation based on existing database technology should prove that the database community is ready to take its role in emerging domains.

References

[A+03] E. H. L. Aarts et al., editors. *First European Symposium on Ambient Intelligence (EUSAI)*, volume 2875 of *LNCS*, 2003.

[A+04] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, 2004.

[A+05] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[B+04a] H. Balakrishnan et al. Retrospective on Aurora. *VLDB Journal*, 13(4), 2004.

[B+04b] S. Boyd et al. *Pro MSMQ: Microsoft Message Queue Programming (Paperback)*. Apress, 2004.

[D+97] M. H. Dunham et al. A Mobile Transaction Model That Captures Both the Data and Movement Behavior. *Mobile Networks and Applications*, 2(2), 1997.

[F+05] M. J. Franklin et al. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR*, 2005.

[H+05] R. Huebsch et al. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR*, 2005.

[KK00] J. S Karlsson and M. L. Kersten. Omega-storage: A Self Organizing Muli-attribute Storage Technique for Large Main Memories. In *Australasian Database Conference*, 2000.

[L+04] W. Litwin et al. LH*RS: A Highly Available Distributed Data Storage. In *VLDB*, 2004.

[LF04] D. T. Liu and M. J. Franklin. The Design of GridDB: A Data-Centric Overlay for the Scientific Grid. In *VLDB*, 2004.

[M+05] S. Madden et al. TinyDB: an acquisitional query processing system for sensor networks. *ACM TODS*, 30(1), 2005.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*, 2002.

[OV99] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Prentice Hall, 1999.

[P+04] Y. Petrakis et al. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. In *DBISP2P*, 2004.

[R+02] J. Rao et al. Automating Physical Database Design in a Parallel Database. In *SIGMOD*, 2002.

[S+94] M. Stonebraker et al. Mariposa: A New Architecture for Distributed Data. In *IEEE 10th International Conference on Data Engineering*, 1994.

[Tum04] M. Tumma. *Oracle Streams: High Speed Replication and Data Sharing*. Oracle In-Focus Series, 2004.

[Z+04] D. C. Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.