Centrum voor Wiskunde en Informatica

**REPORT***RAPPORT*

*INS*

Information Systems

***INformation Systems***

# Armada: a reference model for an evolving database system

F.E. Groffen, M.L. Kersten, S. Manegold

**REPORT INS-E0603 MAY 2006**

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)


P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

# Armada: a reference model for an evolving database system

ABSTRACT
The current database deployment palette ranges from networked sensor-based devices to large data/compute Grids. Both extremes present common challenges for distributed DBMS technology. The local storage per device/node/site is severely limited compared to the total data volume being managed and the local processing power is too limited to handle a high query load. In this paper, we propose Armada: a novel reference model for a distributed database architecture to facilitate evolutionary growth. Participating systems can autonomously decide to take responsibility in the distributed data management task. The system adapts to varying workloads and supports dynamic system re-sizing, e.g. growing and shrinking of the system at large. Armada uses lineage trails to capture the metadata and history. Lineage trails from the basis to direct updates to the proper sites, break queries into multi-stage plans, and provides a reference point for site consistency. The lineage trails are managed in a purely distributed way, 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.

*1998 ACM Computing Classification System:* E.1; H.1.0; H.2.1; H.2.4; H.2.7; H.3.4
*Keywords and Phrases:* Distributed Database Systems; Distributed data structures; Data models; Data directory
*Note*: Work carried out under project Bricks IS2 ``Petabyte Data Mining Challenge''

# Armada: a Reference Model for an Evolving Database System

Fabian Groffen Martin Kersten Stefan Manegold
Fabian.Groffen@cwi.nl Martin.Kersten@cwi.nl Stefan.Manegold@cwi.nl

*CWI*
*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

ABSTRACT

The current database deployment palette ranges from networked sensor-based devices to large data/compute Grids. Both extremes present common challenges for distributed DBMS technology. The local storage per device/node/site is severely limited compared to the total data volume being managed and the local processing power is too limited to handle a high query load.

In this paper, we propose *Armada*: a novel reference model for a distributed database architecture to facilitate evolutionary growth. Participating systems can autonomously decide to take responsibility in the distributed data management task. The system adapts to varying workloads and supports dynamic system re-sizing, e.g. growing and shrinking of the system at large.

Armada uses *lineage trails* to capture the metadata and history. Lineage trails from the basis to direct updates to the proper sites, break queries into multi-stage plans, and provides a reference point for site consistency. The lineage trails are managed in a purely distributed way, 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.

*1998 ACM Computing Classification System:* E.1, H.1.0, H.2.1, H.2.4, H.2.7, H.3.4
*Keywords and Phrases:* Distributed Database Systems, Distributed data structures, Data models, Data directory
*Note:* Work carried out under project Bricks IS2 "Petabyte Data Mining Challenge".

## 1. INTRODUCTION

Ambient, mobile and datagrid environments call for major changes in the design and implementation of distributed database systems. 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 [ACvLdR03]. Mobile environments are characterised by a large number of clients sharing information through multiple data brokers to realise e.g. a digitised community. The data 'follows' the device, which may be offline for lengthy periods [DHB97]. Datagrid environments emerge in scientific settings as heterogeneous platforms providing compute and storage services to the science community. Data and processing is distributed and managed using database technology [LF04].

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.

First and foremost, the sites are not always connected to form a coherent distributed system. In mobile settings the prevalent mode is to engage in short sessions to exchange and synchronise information until the battery runs dry. In a grid environment sites may be offline for maintenance, or simply refuse providing services due to local resource constraints. In the ambient environment sites can get broken or components with new functionality are added over the years.

A consequence is that datamanagement should take site autonomy and volatility as a driving force and a core feature of a database system. A central broker to guide all interactions is a *dead end* for the scalable solutions required. Several sites may take such a role for a limited period and only for part of the data space. Instead, 'contractual' arrangements and economic models are needed. They should facilitate a 'device market' to ensure the desired system behaviour and be flexible to cope with temporal and evolutionary changes.

The role of the user also drastically changes. In an ambient setting his appearance is 'sensed' and indirectly activates data management tasks. An avalanche of database queries and remote updates may emerge. In the mobile and grid settings the real world monetary economics call for an active interaction to maximise the return on investment. A better synergy between the tasks of the user and the database system is called for. The users should again become drivers of the database interaction and they should be aware of cost.

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 [TOV99]. 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 then an exception. It calls for data management schemes optimised for incomplete and only partially consistent information.

The database community missed the birth of the world-wide-web as the largest heterogeneous database. Do we miss the emerging distributed database challenge, because we are too late to challenge our assumptions underlying existing database architectures? The fulfilment of this dream has been put forward as the development of organic database systems as a generic, self-managing and situation aware database system [KWF+03, KS99].

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 cooperating in a distributed application.

Sites can easily join an Armada alliance by donation of 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 bottle-neck, single point-of-failure and hot-spot in busy systems.

The user is back into the loop to steer query processing and distributed transaction management. Common user policies can be captured in scenarios managed by a middleware software layer, but the autonomy of the Armada sites ultimately relies on a cooperative user as well.

The remainder of the paper is structured as follows. Section 2 introduces the Armada model using analogy reasoning, 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 and transaction management are described in Sections 4 and 5, respectively. A sample embedding of the Armada model in the trend in distributed database systems is indicated in Section 6. We conclude and give a short outlook in Section 7.

## 2. The Armada Model

The main idea 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 form a hotspot. Both operations that query or update the actual data and operations that change the structure of the system at large, e.g. addition or removal of nodes and/or reorganising the data for load balancing purposes need to access the central server to lookup or update the metadata. 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. However, 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. While making data operations cheaper since all metadata is available locally at each site, it increases the price tag of structural operations significantly, 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 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 new 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 accompanies the barrel to 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 (or whoever has access to them) 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, any discussion of actual instantiations of the model and related issues, like strategies as to when, why and how to perform structural changes, is beyond the scope of this paper, and left for future work.

### *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 a 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 the purpose and potential of 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*

[1] The trail adminstration 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.

*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, i.e., 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: the function $g$ that is applied (and hence describes the content of the new box), the site $S$ that the new box is shipped to and, for the convenience of later reference, the identifier of the new box $B$.

DEF. 2 *A* lineage step $s = [\, g \,, S \,]{:}B$ *is a composition that identifies the application of structural operation $o$, 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 the data in the box.

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 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:

$$\begin{array}{rl} T'' = T'''.[\, g'', S'' \,]{:}B'' \,; & [\, g' \,, S' \,]{:}B' \\ T' = T''.[\, g' \,, S' \,]{:}B' \,; & [\, g \,, S \,]{:}B \\ T = T'.[\, g \,, S \,]{:}B \,; & \end{array}$$

The set of successor steps is empty for all boxes to which no structural operation has been applied, yet. The set of predecessor trails is empty only for one box in an Armada, its *origin*.

DEF. 5 *An Armada instance is born as a single initial box $B_o$ on site $S_o$. 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:*

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

*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 $r_i,\ldots,r_{j+m}$ 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 $B_o$:

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

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

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 $B_o$:

$$\begin{array}{lll} T_o = & [\,\%, S_1\,]{:}B_o\,; & \left\{\begin{array}{l}[\,f\ , S_1\,]{:}B_1\\ [\,f', S_2\,]{:}B_2\end{array}\right.\\ T_1 = [\,\%, S_1\,]{:}B_o\,.\,[\,f\ , S_1\,]{:}B_1\,; & & \\ T_2 = [\,\%, S_1\,]{:}B_o\,.\,[\,f', S_2\,]{:}B_2\,; & & \end{array}$$

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. Although this operation can act as an inverse-operation to the clone and chunk operations, i.e., re-constructing a previously cloned or chunked box, its use is not restricted to this. Our model allows to apply the combine operation to an arbitrary set of (independent) 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.

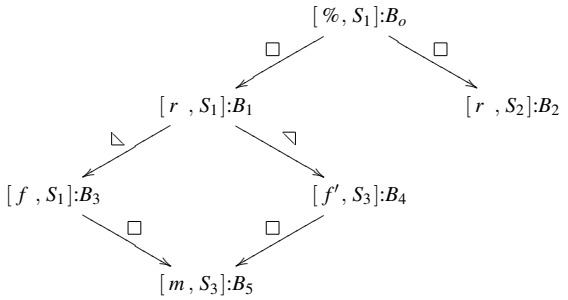
$$
\begin{array}{rl}
T_4 = & T_3\,.\,[\,r\ ,S_1\,]{:}B_4\,;\ \ [\,m\,,S_1\,]{:}B_9 \\
T_6 = & T_2\,.\,[\,f'',S_2\,]{:}B_6\,;\ \ [\,m\,,S_1\,]{:}B_9 \\
T_9 = & \left.\begin{array}{l} T_4 \\ T_6 \end{array}\right\}.\,[\,m\,,S_1\,]{:}B_9\,;
\end{array}
$$

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

*2.3 An Armada Database*

In practise, databases that follow the Armada model, will evolve over time. For many reasons, boxes will participate in operations, such as, chunking due to resource limitations, or cloning for redundancy reasons. We present an example of a database consisting of 5 boxes, and discuss it shortly.

$$
\begin{array}{rl}
T_o = & [\,\%\,,S_1\,]{:}B_o\,;\ \left\{\begin{array}{l} [\,r\ ,S_1\,]{:}B_1 \\ [\,r\ ,S_2\,]{:}B_2 \\ [\,f\ ,S_1\,]{:}B_3 \\ [\,f',S_3\,]{:}B_4 \end{array}\right. \\
T_1 = & T_o\,.\,[\,r\ ,S_1\,]{:}B_1\,; \\
T_2 = & T_o\,.\,[\,r\ ,S_2\,]{:}B_2\,; \\
T_3 = & T_1\,.\,[\,f\ ,S_1\,]{:}B_3\,;\ \ [\,m\,,S_3\,]{:}B_5 \\
T_4 = & T_1\,.\,[\,f',S_3\,]{:}B_4\,;\ \ [\,m\,,S_3\,]{:}B_5 \\
T_5 = & \left.\begin{array}{l} T_3 \\ T_4 \end{array}\right\}.\,[\,m\,,S_3\,]{:}B_5\,;
\end{array}
$$




In the above example, the origin box $B_o$ was first cloned into boxes $B_1$ and $B_2$, of which the first one was then chunked into $B_3$ and $B_4$. Finally, both chunks were re-combined into $B_5$. In the above figure, we use the symbols $\square$, $\triangledown$ and $\triangleleft$ to indicate the coverage of the functions applied in the operations on the boxes. The $\square$ symbol represents the full space as covered by the origin box, which is not changed by the clone operation. The chunking operation in the example equally divides the full space in two parts, $\triangleleft$ and $\triangledown$, which are combined again, resulting in the original coverage $\square$.

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 block a *lineage* can be seen in the predecessor trails, which grow every step by extending the lineage information of the box being operated on.

*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 above example, we will briefly sketch that the lineage trails provide sufficient information to find the required box(es).

Suppose a client contacts site $S_2$ with a query that requires data from values covered by $\triangle$. There is only one trail available on site $S_2$, namely trail $T_2$. From this trail, the site can deduce that it has a full replica of the whole data ($\square$) in the Armada. The function $r$ is the generic replica function that does not impose any limitation to the data coverage. Since the predecessor of $T_2$ is the origin, this means all possible data is available on the site and hence it can return a full (definite) query result to the client.

Now suppose a client contacts site $S_1$ with a query that requires data from $\triangleleft$. The trails available on site $S_1$ — $T_o$, $T_1$ and $T_3$ — form the basis to what answer the client gets. *Before* the merge operation to box $B_5$ had taken place, site $S_1$ could deduce from trail $T_3$ that it can answer the query as box $B_3$ contains the data. But once the merge operation *has* taken place, $B_3$ does not exist any more, and hence $S_1$ lacks the data to answer the query. However, since it still holds the metadata of $B_3$ with the respective successor step now indicating that the data of box $B_3$ has been merged into box $B_5$ on site $S_3$, $S_1$ can advise the client to contact site $S_3$, instead. In the given scenario, site $S_3$ can successfully answer the client's query, since it holds the requested data, as indicated by trail $T_5$.

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 currently no 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 phase for an alliance is a single site $\mathcal{A} = \{S_o\}$, i.e. the origin site.

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 cooperate with $Y$. For that, $Y$ should follow the Armada charter or adhere to the 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 cooperates 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 $\mathcal{S}_X$ and should be told about possibly new members explicitly by intervention from an outside authority [3].

Selection of candidate sites to join the Armada is initiated by a member when it can no longer fully cooperate due to resource constraints. Given the Armada charter, 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 $X \in A$ be a site with a wish to offload (*CPU, MEMORY*) units. It issues bid requests to $S' \subset \mathcal{S}$ and grants the bid from site Y 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 $Y \in S \cup \mathcal{A}$ is willing or able to make resources available, effectively passing back the problem to the site $X$. This failure can not be resolved at $X$. Instead, the process of finding, negotiating 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] New members can be searched using a client application or could be hardwired in the implementation e.g. using an IP-range.

*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 [TOV99]: the function $f$ is lossless, i.e. each possible data element can be associated with either box $B^f$ or $B^{\bar{f}}$, it should designate disjoint portions $B^f \cap B^{\bar{f}} = \emptyset$, 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 [LMS04]. 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. To make it work, 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. See Section 5 for a broader perspective on transaction management.

*3.4 Armada Heterogeneity*

A real Armada is formed out of different types of ships. To name a few, the *galjoten*, *hoekers* and *spiegelschepen* 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 carried 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 or multimedia images, 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 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. In the remainder of this section we assume the Armada to support a relational database, whose schema is made known upon creation of the original site. For simplicity of presentation, we assume boxes to hold tuples of a single relation.

### *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 postponed until we look at transaction semantics in Section 5.

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 both 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, the whereabouts of the relevant box are not administered centrally; its location may even frequently change.

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 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 $Q_i, \ldots$ 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 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 the 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 user. 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 (cloned). 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 can not a priori be guaranteed using database statistics. 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 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.

DEF. 14 *A query referral graph* $QRG = (V,E,O)$ *where* $V \subset S$, $E \subset V \times V$ *and* $O \subset V \times L$ *trail operations issued at individual sites.*

The elements $V$ represent the Armada sites involved in the query. 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 reserves 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.

For example, a distributed join involves chunking/cloning the operands, solicitation of sites to donate sufficient resources to receive a part of all operands involved and to execute the partial join. The boxes produced carry their own lineage trails. Although conceptually the Armada deals with static boxes, an actual implementation could opt for a more traditional stream/pipelining execution model.

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

Furthermore, 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.

### *4.4 Armada User Interface*

The novelty to put the client in the role of query coordinator calls for an easy to use administrative infrastructure and sufficient local resources to perform these tasks. Query formulation and result set inspection can be supported with a traditional tabular interface.

Additionally, inspection of the site's state requires different visual aids. A client roaming the Armada sites can collect data to show the per site boxes, their usage, and how those boxes relate to each other. Provided the sites are able and willing to share such information. The client could then construct a 'map' of the Armada to give insight in its current state and supports tactical decisions of the client. The map becomes the focal point for the Armada's admiral to manage it globally.

## 5. TRANSACTIONS

The choice for autonomous site behaviour puts traditional database consistency at stake. A general applicable transaction management scheme for an Armada system seems to lie beyond the horizon still. In Section 5.1 the prime approaches are reviewed in the light of the Armada vision. Section 5.2 reviews the consequences for each site. Novel directions based on the intricate setting of an Armada system are pointed out in Section 5.3.

### *5.1 Transaction Policies*

Consider a cloned chunk at two sites $X$ and $Y$, and an update request to either one. Should the sites be forced to cooperate to arrive at a consistent state? When is the update reflected in both instances? Or, is anarchy unavoidable, where both versions diverge as time progresses?

The traditional approach in a distributed/federated DBMS is to put the update receiving site in control to ensure that a transaction can only be committed when all related data has been updated as well. The drawback is that sites get locked into a protocol with other sites that might temporarily or even permanently become inaccessible. In this case there are many foreseeable policies, e.g. time-out and abort, or time-out and accept inconsistency.

For Armada we consider the following policies:

- *Anarchy Policy*. Each site handles update requests and does not pay attention to the state of Armada at large.
- *Delegation Policy*. Each site handles the update request and immediately propagates it to all sites to which the update (may) apply.
- *Mediator Policy*. Updates are always sent to a special agent within the Armada that implements a scenario to maintain global consistency.
- *Resolution Policy*. A dedicated site (group) is in charge to resolve global consistency using a master/replica strategy.

Each policy leads to a quite different client perspective. They range from a situation where global consistency of the Armada is unknown to the client, to a situation where global consistency can be guaranteed before a client is informed on the status of his request.

In an anarchic situation the clients never know if the information accessed or stored is consistent with concurrent actions or even past behaviour. Such a situation seems only warranted if the Armada is used in a situation

where data values stored locally do not affect the interpretation elsewhere. A prototypical case is a situation with read-only data, such as transaction logging of production servers in for instance a telecom environment.

The delegation policy models the traditional distributed database view, where the ACID properties are enforced under all circumstances. It is known for quite some time that this rigidity hinders responsiveness of business critical applications. Often a lower consistency level is deployed at the risk of mis-interpretation of data accessed. From an implementation perspective, this policy also leads to a strong reliance of all participants of the Armada, thereby greatly reducing the autonomy of sites aimed for.

The mediator policy suggests a solution that alleviates the sites of taking care of consistency. Instead, a middleware solution of trusted agents takes care of reaching a consistent state. The strategy of the agent — which can be located on one of the sites of the Armada — can be adjusted to the situation. An agent equipped with a set of common policies could become part of the standard Armada infrastructure.

The resolution policy exploits the serialisation offered of pure master/replica techniques. It requires consensus amongst all Armada sites of their role in consistency enforcement. For distributed constraints it creates a dependency that challenges the autonomy of each site. However, the client is assured of consistency when it interacts with a master site only.

### *5.2 Site Consistency*

A straightforward realisation of an Armada site is to encapsulate an existing database system with the required new functionality. Each site should then manage the data in the boxes hosted in a way traditional DBMS systems would do. Thereby providing full ACID properties.

This requirement also holds for the metadata captured in the lineage trails. They form a catalog table in the database subject to ordinary transaction rules.

From a client perspective, the autonomous Armada site is in no way different from an ordinary database system. Except for the ability of the site to autonomously decide to cease operation. However, this is not distinct from experiencing a crashing server. Multi-statement transactions follow the same approach.

### *5.3 Armada Consistency Model*

Forcing ACID properties on the global state of the Armada, forces the autonomy of all the sites to a bare minimum, as they may have to start a global transaction for every action they take. A heterogeneous pool of sites will not support this, hence a different consistency vision is required.

Regardless of the autonomy attributed to all sites, it remains of crucial importance to assess the system state. Applying an update to a site may lead to a database inconsistency that requires timely detection or a scheme to reconcile the differences at some point in the future, preferably without client interaction. In all circumstances it is the client's interest to see the outcome of his transaction.

Two different routes for the Armada are conceivable: *control consistency by design* or *authoritative guidance*. Control consistency by design is a scheme to avoid any distributed consistency problems upfront. Distributed transactions are forbidden from the outset. Instead, such a transaction first calls for bringing the relevant boxes together on a single site, where it can be dealt with as a pure local transaction.

Of course, the drawback of this scheme is that global aggregate constraints are disallowed from the outset. They would require a site with unlimited resources to hold the complete database. However, we conceive the restriction on consistency rules as natural in a setting where sites do not have (or can, or will) provide the information needed in the first place.

In Armada this puts constraints on the lineage trails that can be constructed by the autonomous sites. *Chunk* operations should be chosen such that all consistency related information stays within a single box. The combine operation is immune to the consistency rules under this scheme. The *clone* operation is problematic, because conceptually the cloned boxes should be brought together first using the *combine* operation. When in a single place, the update takes place on the combined box, whereafter a series of *clone* operations could move the boxes back into their original sites again.

The alternative, an *authoritative guidance* approach exploits the information stored in the lineage trails. For, each distributed transaction received by a site can be analysed against its locally maintained trails. The referral query list or query graph models the subqueries and sites involved. This can be sent back to the client as a

recipe to execute the global transaction. The prime disadvantage is that a two-phase, client initiated commit is needed to finalise the transaction. During this process boxes kept at participating sites are locked or an optimistic concurrency scheme should be implemented.

For the client it provides an easy decision. Given an update request ask any site for the query plan to execute. It will be limited to sites that can locally check constraints, possibly after receiving complementary information from subqueries. Global consistency has been achieved if the complete plan can be successfully executed.

Following the discussion on the various policies in Section 5, global consistency is either strictly enforced or left open for an external entity to solve. Depending on how the Armada is used, a different policy might be appropriate. Mainly query-intensive areas, with less or no updates, are served fine by an anarchy policy.

## 6. Related Research

Research on federated/distributed database architectures has a long history. All major DBMS suppliers provide technology to realise a distributed database. However, 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 [BCR+04].

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 [MFHH05, FJK+05, BBC+04, AAB+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 [PKP04, 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 [HCH+05]. Unlike P2P, Armada has functions that define how data is split over a number of boxes, which allow for concise localisation of data.

Research of scalable data structures has overcome the limitation in P2P systems using a globally known, but locally adaptive partitioning function [LMS04, KK00]. Also the client behaviour 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 [TOV99]. Additionally, client/server settings form the base of dealing with the work, thereby greatly reducing the autonomy of the participating servers, as in [MA05].

The area of self managing and self tuning databases limits itself by only advising the DBA [RZLM02, ZRL+04] or only dealing with indices and materialised views [ACK+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.

## 7. 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 cooperative setting. The system volatility, devices join and leave the ensemble, calls for a fresh look on metadata, query processing and transaction management.

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 into several dimensions charters a rich research landscape ahead. The issues identified may stimulate a larger audience to join our quest to renovate the foundations of distributed datamanagement. 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 proof that the database community is ready to take its role in emerging domains.

# References


[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[ACK+04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005. In *Proc. VLDB Conf.*, 2004.

[ACvLdR03] Emile H. L. Aarts, René Collier, Evert van Loenen, and Boris E. R. de Ruyter, editors. *Ambient Intelligence, First European Symposium, EUSAI 2003, Veldhoven, The Netherlands, November 3.-4, 2003, Proceedings*, volume 2875 of *Lecture Notes in Computer Science*. Springer, 2003.

[BBC+04] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uri Centintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[BCR+04] Scot Boyd, Richard Costall, Ken Rabold, Arohi Redkar, and Carlos Walzer Tejaswi Redkar. *Pro MSMQ: Microsoft Message Queue Programming (Paperback)*. Apress, 2004.

[DHB97] Margaret H. Dunham, Abdelsalam Helal, and Santosh Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *Mobile Networks and Applications*, 2(2):149–162, 1997.

[FJK+05] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, Eugene Wu 0002, Owen Cooper, Anil Edakkunni, and Wei Hong. Design considerations for high fan-in systems: The hifi approach. In *CIDR*, pages 290–304, 2005.

[HCH+05] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR January 2005*, 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*, pages 57–64, Canberra, Australia, January 2000. IEEE Computer Society Press.

[KS99] M. L. Kersten and A. P. J. M. Siebes. An Organic Database System. Technical Report INS-R9905, CWI, Amsterdam, The Netherlands, May 1999.

[KWF+03] Martin L. Kersten, Gerhard Weikum, Michael J. Franklin, Daniel A. Keim, Alejandro P. Buchmann, and Surajit Chaudhuri. A database striptease or how to manage your personal databases. In *VLDB*, pages 1043–1044, 2003.

[LF04] David T. Liu and Michael J. Franklin. The design of griddb: A data-centric overlay for the scientific grid. In *VLDB*, pages 600–611, 2004.

[LMS04] Witold Litwin, Rim Moussa, and Thomas J. E. Schwarz. Lh*rs: A highly available distributed data storage. In *VLDB*, pages 1289–1292, 2004.

[MA05] Kaloian Manassiev and Cristiana Amza. Scalable In-Memory Database Replication through Distributed Multiversion Concurrency Control. In *to appear: http://www.cs.toronto.edu/ kaloianm/publications.html*, 2005.

[MFHH05] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

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

[PKP04] Yannis Petrakis, Georgia Koloniari, and Evaggelia Pitoura. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. In *Databases, Information Systems, and Peer-to-Peer Computing*, 2004.

[RZLM02] Jun Rao, Chun Zhang, Guy Lohman, and Nimrod Megiddo. Automating Physical Database Design in a Parallel Database. In *Proc. SIGMOD Conf.*, 2002.

[TOV99] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice Hall, 2nd edition, 1999.

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

[ZRL+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Physical Database Design. In *Proc. VLDB Conf.*, 2004.