Armada: an evolving database system

Groffen, F.E.

Citation for published version (APA):
2 Overview

2.1 History

It is the nature of humans to organise. Admittedly some more than others, but deep inside we have a force that makes us form groups, build houses in orderly streets, define time tables for public transportation, etcetera. A place for everything and everything in its place. Most of our organisational efforts lead to more efficiency afterwards. Many streets are clustered around a given theme, in a city district, of a certain city, in a province, state, country and so on. But most of all, house numbers in a street are ordered ascending walking away from the city centre, one side the odd numbers, on the other the even ones.

Perhaps the most obvious example of our organisation drive is the library. On many shelves, per genre, all books are ordered first on author, then on title, just to achieve an easy search process afterwards. But, libraries are old fashioned and books are made of paper — a material that can only be used once and worst of all, tends to decay over time. No matter how well the library is organised, going to the library takes time. Walking around to find a book takes a few precious seconds, and do not forget the browsing through the book itself. Over the years, driven by another nature, man has found his way into the digital era. Even though this era is mainly virtual, the organisational drive has found its way deep inside it. As a world solely created by man, organisation is the key component it is made out of. Everything is nicely addressable, no exceptions for which no proper rule is defined, no chaos that eventually does not appear to be a well organised structure with logical rules. Among other things, organised storage of data in this organised digital world is nothing more than a logical thing to do, and hence the digital databases were born.

In the grand scheme of things, databases have a long history, even going back centuries if one considers the non-digital libraries and records as a data base. Nowadays’ digital libraries certainly show much resemblance in the
storage and search process with those non-digital libraries. Starting from the 1960’s, computers became cost effective, and database technologies were developed. The new databases were used to store administrative data, such as flight reservations. These were not books, but these records were searched for, like books. For this a similar structure as found in libraries was necessary, including a method to conveniently search for those records. Codd laid the foundations of modern databases with a proposal for a relational model [16] around 1970. In this model, the logical organisation of the database is disconnected from the physical organisation. Data comes in relations that can be considered to be sets, inheriting set operations. This yielded in relational queries over the data. The proposal by Codd resulted in two proofs of concept: UCB’s Ingres [66] and System R [4] from IBM. The latter used a query language SEQUEL, whose descendant SQL became the de-facto database query language in the mid 1980’s.

2.1.1 Distributed Databases

Soon these databases were extended to run on multiple machines instead of on a single monolith [33]. The relational model suits fine in this distributed setting which goes one step further in decoupling the logical organisation of a database from the physical storage, by hiding the location of the data from the user. As such, the distributed variant is a drop-in replacement for the single database system. Three architectures for distributed databases exist [68] with either a component shared or nothing in common with the other systems. Shared disk and shared memory systems are architectures where multiple CPUs work with a single disk or memory subsystem. In the shared memory architecture, all processors and disks share a common, often relatively large, memory. Communication between processors can hence go through reads and writes in the shared memory. In the cluster approach of shared disks, each processor has its own memory, but access to a single large disk. The shared nothing variant has no hardware components shared. Instead, separate systems send each other messages through a network between them. Finally a hybrid combination of these architectures can be formed into a hierarchy.

Network The shared nothing case has become popular, due to its cost effectiveness [63]. However, in the early days, this architecture was hindered by network instability, seriously affecting the process of shipping several megabytes of data per day. Hence, only once the network connections became fast and reliable, this architecture became feasible and needed. Distributed shared
nothing databases are scalable systems, consisting of multiple standard computer systems which are more powerful and cheaper than a big mainframe computer [18]. This trend has been stimulated by the rise of the Internet, which is mainly built of network-connected shared nothing systems.

**Parallelism** The distribution, which was originally intended for a shared database between geographically spread offices, allows for parallelism [18, 33] as well. When exploited well, this parallelism can result in considerable performance improvements. Hence, the query processing on this architecture tries to exploit this. In the Volcano model [27] the query processing design is modelled in such a way that it can be easily switched to parallel execution. The “brackets” of this model are arbitrary operations based on one or more input channels with as result one output channel. These brackets can be placed on other machines without changing the execution plan. Since shipping large amounts of data over the network is inefficient, this should be taken into account when doing distributed query processing. It is represented by the query shipping versus data shipping approaches to the data processing problem. Factors which influence the decision for either shipping the full data to the processor or first doing the processing where the data is and shipping that result include network characteristics such as speed and latency, the processing power of the involved systems and whether the system doing the processing has all the other data available to do the processing [46].

**Heterogeneous and Federated Databases** For various reasons, companies and institutions end up with several types of databases, often from different vendors. These heterogeneous databases are usually incompatible with each other and need additional measures to still perform joined work [24, 43]. This is the area of federated and heterogeneous database systems. In those systems, a mediator delegates full or partial queries to the underlying database systems via a wrapper that encapsulates the database specific behaviour into a generic request and response [32]. Because the capabilities of each database are generalised this way, the possibilities are limited [43].

**Data Placement** Where the data resides partially defines where it can best be queried. Static data placement, assumes the data to be positioned as defined by a database administrator [53]. Data location then depends on what kind of queries take place from what locations in the system [12]. Tools to examine the query workload function as input on where to place what data. However, the
real query workload, is hard to predict, as well as tends to change over time. Relocation of the data by the administrator is a tedious job, which triggered a focus towards dynamic data placement approaches [35]. The ideal here is to keep statistics and automatically move and make copies of data to adapt to the current workload [53, 11]. Work has been done on replication and caching of data to dynamically match the workload. Here two types can be distinguished. First the algorithms that try to reduce the communication costs by placing the data close to the clients that are likely to use the data e.g. by static prediction of a workload, and second the algorithms that do load balancing by replicating popular hot data dynamically [61, 9]. Data moving is an extension to replication, where a final migration stage abandons the old copy, such that the new copy on its new location becomes the primary one.

**Replication** With multiple copies of the original data, updates become less trivial to apply. Depending on the level in which all replicas need to be consistent with each other, architectures were designed [68]. An approach to keep consistency throughout all replicas is to force all copies to apply the same update at the same time. This is an expensive method as it requires all replicas to make the update disallowing any other operations on that data at the same time. Techniques to only use a majority of the replicas, a quorum [25, 69, 23], were devised to relax this requirement. A hierarchical architecture typically has a primary copy, on which all updates are performed. All other replicas receive the updates from the primary copy. The other copies may be out of date, but still consistent for what they store. Since updates are only done on the primary copy, no conflicts can arise. However the system may fail completely when the primary copy fails, hence techniques to choose a new master copy dynamically aim to resolve those cases. Other approaches use asynchronous updates to replay update statements when necessary to defer doing the work until really necessary.

**Economic Clients** In a system where clients can choose between multiple servers to execute their query on, the complexity of computing the right decision becomes too large for a single system to control. Systems relying on “economic models” eliminate this complexity via bidding for resources in an auction. Each server offers certain services for a given “price” as a response to a client’s query. The first proposed distributed database system based on this economic drive, was Mariposa [65]. In this system, clients assign a budget for each query. Brokers in the system start auctions to provide the necessary data
and to perform the required operations on this data. Servers that are willing
can place bids, which the broker will try to optimise in the cheapest possible
way to benefit itself by performing the query using less money than the offered
budget. If a budget is insufficient, a broker refuses to execute the query.

Decentralisation The client-server model assumed till now, causes a concentra-
tion on one, or a small number of servers targeted by the clients. To retain
service levels, load balancing schemes and fault tolerance algorithms have been
developed. However, this concentration problem stimulated research on ap-
proaches that try and spread this load over a large number of participants in
the network. The class of systems that were devised in this area are typed Peer-
to-Peer (P2P) systems, where each participant acts both as a client as well as a
server. It trades its own resources for those of others. At the heart of each P2P
system is an indexing structure to find keys in the distributed network of parti-
cipants. Examples are [51, 56, 58, 59, 62] where distributed hash tables (DHTs)
or other distributed hashing schemes are used to efficiently map data items onto
a participant of the system. The expectations of these systems to solve the con-
centration bottleneck has led to research for database technologies applied to
these systems, such as [29, 37, 40]. By the nature of P2P systems, the result-
ing database systems typically deal with efficiently finding data sources which
together are assumed to be the “database”. The data sources need not to be rel-
ational, or files, but can also be streams or XML documents. Here also mobile
and ambient settings get in the picture. Typically, the scene of large numbers of
database-empowered sensory systems which learn and exchange information to
reach a common goal are the topic [1]. In mobile environments, information is
shared through multiple data brokers. Data ‘follows’ the mobile device, which
may be offline for lengthy periods [20].

Stream Systems From a batch-like processing system, recent shift has been
towards continuous emission of data via streams. Examples can be seen in
the stock market “tick” data and various logging applications that generate in-
formation about changes or conditions measured by sensors. With the latter
becoming economical attractive, new floods of continuous data needs to be pro-
cessed. While these sensory streams need a fair bit of routing and aggregation
in an energy preserving manner, database systems have been adapted to per-
form those aggregations, or to help execution of formulated queries in an SQL
dialect, such as in TinyDB [49]. Monitoring applications that handle streams,
may get overloaded by a peak of observation data. In such case it is acceptable
for the monitoring application to drop or join observations, reducing the precision in favour of remaining a responsive system. Such operations are in general against the design principles of traditional database systems that try to handle everything, fully correct and in the same order.

2.2 Related Literature

Autonomy, evolution and decentralisation are well known terms in database literature. We briefly discuss the known work on each of these topics.

**Autonomy** In the research area of federated databases, the notion of autonomous components is rather common [34]. The components that manage the data are often willing to share, as long as they retain to be in control. The autonomy in federated systems stems mainly from the heterogeneity of the participating components. Because of their intrinsic differences it is hard, if not impossible, to leverage precise control on their behaviour. But also the sociological aspect of ownership of the database components plays a role in the setting of federated databases. Even though for instance universities share their databases with others, they prefer to remain in control of their own.

In [71] three types of autonomy are distinguished: design autonomy (D-autonomy), communication autonomy (C-autonomy) and execution autonomy (E-autonomy). **Design autonomy** refers to the ability of a component in a system to choose its own design. Types mentioned for D-autonomy are design selections of the data being managed (universe of discourse), how to represent this data e.g. using which schema, in what way transaction semantics are defined for the local constraints and on what hardware or with which software the database runs. Of course D-autonomy also allows to adopt new approaches on the aforementioned areas whenever that is considered to be beneficial from the component’s point of view. This type of autonomy is typically seen in the federated databases case, and used in e.g. [22]. **Communication autonomy** focusses on the ability to decide with who, when and what the component communicates. C-autonomy involves the freedom to decide at any moment whether to communicate, and if so, to some or all other components. This also includes the freedom to be selective on requests itself. A component may for example refuse to execute a query for a given host, but accept the same query when issued from another host. C-autonomy could be a requirement for weakly connected systems, such as mobile or ambient systems [8] where components are expected to be connected to the network for undefined periods at undefined times. For
reliable communication in these systems between multiple C-autonomy based components, a non-C-autonomy component has to be used as mediator for their communication. Lastly, execution autonomy deals with the freedom of a component to accept or refuse the execution of a request. This is based on the amount of efforts it takes to execute the request. E-autonomy allows a component to refuse to execute queries that exceed for instance a certain time limit. But also the effect of the request on the system, such as congestion or deadlock situations, may be reasons for an E-autonomous system to reject a request. E-autonomy can even result in requirements for other components to be involved in the execution of a request, if the executing component considers this to be necessary.

To conclude, an autonomous system in literature is roughly referred to as a system reluctant to share sensitive or critical information that limits its cooperation with others. A compatible vision of autonomy can be found in e.g. [14]: “autonomous, that is, they cooperate to only a limited extent, and do not expose sensitive or critical information to each other”.

**Evolution** In the context of databases, evolution is focused on the contents and schema of a database. This is seen as within the local database, and resolution of problems related to either the changes of the schema, or how to make those changes to the schema or object classes in an object oriented database. Causes that make evolution of this kind necessary include migration of different systems into one, changes of rules, changes in applications and their database requirements and new needs arising from new technologies such as the internet.

Schema evolution typically involves three issues: physical evolution, logical evolution and continuity during the evolution process. In the physical schema evolution, typically the tuning parameters of the database are changed to improve its performance. Here, attributes like buffer sizes, storage types and the number of worker processes are to be tuned to the application needs. Research in this area has focused on mainly automating the database such that it can tune itself. As the amount of tuning knobs continues to grow the perfect combination gets harder to find. Typical example of such self tuning approach is [13] where indexes on the data are automatically chosen based on a workload, as to find a good trade-off between storage overhead and gained performance.

Obviously, many tuning operations that are at the heart of the database system, require it to be taken off-line. During this downtime, the database cannot serve any requests, hence it is important to minimise this downtime.
However, it would be even better if the system was not to be brought down for the tuning operations. This requires changes to the way database systems are built, such that from static structures, dynamic structures are made and for instance indexes can be built while the system remains fully operational, such as for instance in [54].

On the logical side of schema evolution, problems regarding changes in the (relational) database schema are dealt with. While views on the data can help to make a schema available in another form, they are only of limited help as typically updates to views over tables are difficult and mostly not allowed by the underlying database. The only option left is to evolve the schema through complex transformations. Schema evolution tools try to assist on these transformations, such as [6]. This assistance is not trivial in any case, as semantic meanings in schemas often require the human in the loop.

**Decentralisation** In literature, decentralisation is an often used term that comes close to parallelism. Particularly P2P systems refer to decentralisation, because they aim to avoid a central dependency, whereas parallelism does not per sé. In [28] a couple of reasons for using decentralised systems are identified. As noted before at the discussion of autonomy, most organisations want to have control over their own systems. In the light of a distributed system, this means a central controlling entity conflicts with the organisational needs. Hence, a decentralised solution forms a good match. But also heterogeneous systems that are joined together, require their own control. The technical benefits of decentralised systems come close to those of parallel systems. The capacity can grow beyond that of a single system, response times can be improved when the data is placed nearby and availability increases when the systems are geographically widespread.

That the aforementioned benefits can be really attractive, has been proven by the success of P2P systems. Even though their use is mostly for a legally questionable data transfer, fact remains that the current semantics-free, requests for objects by identifier (typically a hash) are quite successful, even though this method is quite limited. P2P systems as such in their current form are quite ineffective to retrieve for instance only the abstract of a given document. Instead the entire document is the granularity of the objects. Data relationships, which are so well understood and supported in database systems, are in particular lacking in P2P systems [29].
2.3 Armada

In the previously described setting of database systems, this thesis explores the path towards evolution, autonomy and decentralisation. Our exploration deals with distribution of shared nothing systems, interconnected by fast networks, as common nowadays. Unlike P2P systems, in Armada we take the database schema and its possible constraints as starting point in our exploration. Distribution and localisation are built on top of the traditional schema approach.

Where the autonomy of a system typically refers to the willingness to share and reveal data to others, in this thesis we lift autonomy to the level where individual systems are the initiators for distribution in the cluster. Due to our schema-centric approach, the systems hosting the data have the required information to do well controlled data distribution. Due to our approach, they also have the required autonomy to perform the distribution using that data. The resulting way of data placement is what we refer to as evolution. In contrast to the known evolution that refers to the data local to a single system, our evolution spans over the entire cluster of participating systems. Hence, a decentralised administration is implied by the autonomous systems that the system comprises of. The essence of decentralisation within Armada is in avoidance of hot-spots in the cluster. Autonomy and decentralisation go hand in hand for this objective.

Our notion of autonomy and evolution are reflected in the first research question of this thesis. In particular the high level of autonomy is articulated by means of site local decisions. Our notion of evolution results in the dynamic growth factor, that addresses the entire cluster, instead of a single system. We answer this first research question with a model that encompasses our notion about autonomy and evolution.

With autonomy in place, those who interface with the cluster encounter changes, in particular the clients issuing queries. Because mediators use centralised information about the cluster, their approach does not work in the previously sketched setup of Armada. The DHTs from P2P systems are well suited to locate objects, but they lack support for the schema based approach chosen. For the second research question we therefore explore a stepwise query resolution approach, where clients navigate through the cluster by itself following schema information.

Evolution in databases is expected to become an automatic self-tuning component. However, the growth of a system is not just a matter of making a performance decision based on a workload. It involves an extension to the sys-
tem that need not to be limiting for following growth operations. In fact, the system needs to be able to keep on evolving all the time, to adjust to changing requirements. This is the area of the third research question.

Summary

Database systems have become an important part of our world. Over the years they developed to mature centre pieces in all systems that rely on data storage and retrieval. Being corner stones, database systems have to endure and cope with high workloads and requirements. This lead to the distribution of the database over multiple systems, a trend that continues to become more popular as networks improve. Many aspects of distributed databases, such as parallelism, data placement and replication have been researched to unleash its full potential. Along the way, new visions on the traditional base of data have been developed as well, such as wide decentralisation and streaming systems.

With Armada, autonomy of a database is pushed to the level where the local database is the initiator for distribution, based on local conditions. This form of decentralised distribution results in evolution of the entire cluster, which adapts to the workload right there where it is necessary, without a central controlling component.