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

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
5 Execution Model

5.1 Introduction

The Armada objectives explicitly put initiatives outside of the world made of boxes and sites. On purpose, the user of the Armada system is expected to play an active role in the process of its own query execution and the overall state of the Armada itself. The emphasis is put on the capability of resolving problems by making decisions. This is typically the user, but assisted by a program that performs the execution of queries, hopping between servers to collect pieces of the answer and construct the final answer out of those. Additionally, when clones are in the system, synchronisations between them are the responsibility of those who add or modify data, for which an application can perform a by the user chosen strategy.

When the user in charge of all decisions, the system puts a heavy burden on him or her in terms of interactions, and hence this is not desirable. As seen in Chapter 4, operations can be automated to a certain degree, thereby relieving the user from work. Also here, many actions the user should take can be supported by a program that acts on behalf of the user.

This Chapter presents some techniques that can be applied on top of the Armada model. It shows by simple solutions that the setting where servers deny certain responsibilities, is not unusable.

Assisting Users Assume a user tries to insert 10000 tuples into an Armada. Sites in the Armada can hold at maximum 1000 tuples. Without any help, the user would start inserting tuples on site $S_0$, which in this example initially is empty. After 1000 tuples, $S_0$ reports to be full, and something has to be done to resolve this. The only option the user now has is to to start looking for a new site to host more of its data. If the user fails to do so, it is obvious that it
cannot continue inserting its data. Given that $S_1$ is free and the user found it, it instructs $S_0$ that it should chunk to site $S_1$ using a given function. $S_0$ may refuse this operation, for instance if it cannot reach $S_1$, or when $S_1$ is not willing to cooperate.

Obviously this continues with the user involved all the time for the rest of the chunk operation and further inserts, resulting in an Armada eventually causing more headaches than delights. Instead, an agent in the system can help to alleviate most of the work for the user. Redoing the same example with the agent as mediator between the user and Armada sites, the agent receives that $S_0$ is full and starts searching for a new site on behalf of the user. Agents can use economical models, statistics, first come first serve strategies, etc. to select a site to chunk to from a pool of existing free sites. If the agent fails to do so, it can return to the user with the problem at hand. The agent lets site $S_0$ decide how to chunk to $S_1$, which includes the moving of data from $S_0$ to $S_1$, if any.

The agent then continues to insert tuples on $S_0$. Depending on the chunk function chosen, $S_0$ can handle those tuples, or not. In any way, it has the means to redirect the agent to $S_1$ when the data does not fit any more. Once the agent receives such redirect, it continues inserting data on the new site. Eventually, the cycle repeats itself when $S_1$ reports itself to be full. In an ordinary case, the user need not to be involved in the entire process of chunking the data, when an agent is in effect. As obvious, most of this work that is the implicit responsibility of the user can be performed by the agent based on some heuristics, and preferences of the user.

**Query Execution** After the user’s data has been inserted, the Armada consists of a number of sites. A user can contact any of those sites and request it to execute a query. In the trivial case, a user poses a query at site $S_x$ which is local to the site itself. This is the case if for example the user requests a single key-value. In such case, $S_x$ can simply return the answer to the query, and it resembles regular database query answering. The simple opposite of the previous scenario is when the user has a query that does not address any of the responsible boxes of $S_x$. In this case $S_x$ returns one or more redirects to the user as an indication where to go. The user uses the redirects to continue its query.

A complexer scenario is when a site is only able to handle a query partially. In such case the coverage of one or more of its boxes address a part of the query, but not completely. Suppose the user sends $S_x$ a query which it can only partially handle. $S_x$ returns the results for the part it can handle, accompanied by the sites that the user should try in order to complete the result. The user
can traverse all redirection sites to try and complete the full result for its query. It has to keep track itself which sites have been seen, as the sites it contacts do not know which sites the user already visited. As such they also return sites previously visited by the user together with the partial results they provide. This scenario depends very much on the user to keep track of the query process, without having access to the actual parts the query consists of. In particular, clones in this scheme may confuse a user that only knows what sites to visit.

An alternative to the previously described approach is again to use the agent for the execution of a query. Instead of having the user being confronted with a “query plan” returned by a site its given query, the agent keeps. The query plan consists of sub queries that have to be executed to finish the query. Because the sub queries match boxes, the agent simply starts traversing the query executing the single box queries, and glueing the results together. In some cases this may result in a large result, or a very lengthy execution time. Since the agent keeps the administration of the query execution process, it can present its progress to the user, who can additionally also control further execution in such case.

5.2 Query Resolution

An Armada system contains sites that host boxes. Sites are database powered entities that perform the local tasks for each box they host. They handle requests either by sending data or by sending a redirect. As noted before, a mediator between user and site is available in an Armada system, called an agent.

Humans are still indispensable for the existence of the Armada system. While many types of users exist, for the system the humans are those that can physically influence the system. That is, adding new hardware to the system by means of new sites, or extended capacity. Even though the system could order new components itself, it physically has to have them installed. But also in the use of the system are humans those that can resolve conflicts, by just making a decision which the system itself cannot devise. Input on what parts of the Armada system to clone reflect a human preference for redundancy. Last but not least are the data and its queries that originate from human users, but essentially make up the existence of the Armada system.

Agent Tasks  The Armada agent can independently do some work for the human user. Typically, an agent carries out query execution and follows redirects. As long as there are no problems carrying out its work it is not necessary to
bother its user with the work of following the redirects and continuing the query resolution process. To help even more, the agent tries to construct a full answer when feasible, thereby hiding the distributed fashion of the Armada system, and performing the database operations to construct the final answer out of the parts that it has received. Finally, when the user adds data to the system, the agent can take care of performing chunking transparently during the insertion process as long as sites with free space are available.

The agent needs the user in cases where it cannot possibly act itself correctly. It is up to the user to decide what to do with a query for which one or more pieces of that data are currently unavailable in the system. For long running queries, the intermediate results may be of interest to the user. Incremental query results based on chunk fractions allow a user to abort execution after it has only partially been processed, while still having a partial answer. Obviously this is entirely a user-based decision. If there is no free space in the system, the user needs to come to an action to either stop adding data to it, or to make space available.

5.3 Consistency

In principle the Armada system does not advocate a master/slave setup towards clones. By not doing so, the risk of the entire system to rely on a single master is avoided, as well as that a single update bottleneck is absent. The consequence is a multi-master setting, where each clone can accept updates, independently of others. Synchronizing the independent updates can cause conflicts when the same data is updated at two or more clones, when a uniqueness constraint breaks because of two independent updates inserting a same key, or when data is deleted which happened to be deleted on the other clones already.

Keeping the clones synchronised with each other is a tedious job. The process of synchronisation can be troubled by a number of situations in the system as indicated before. First it has to be determined if, and if so, how many clones there exist and where they are located. In principle this can be solved by looking for clone operations in the predecessor trail. However, there may be multiple clone operations, and those encountered may actually be based on outdated lineage trails, and hence cloned again, or even combined thus effectively removed.

**Synchronisation** Agents need to deal with previously mentioned conflicts and possible non-directly solvable delays. A danger of pushing the clone synchronisation task to the agent, is in the matter in which the agent takes its job seri-
ously. The system risks synchronisations not being done, just because the agent neglects to do the job. However, if the Armada system would perform the synchronisations, it runs the risk of having to make decisions on conflict situations, as well as ending up in dealing with unavailable sites.

To address the risks of the Armada system, it has to be guaranteed that synchronisations eventually can be made. This means agents need not to do their job, while there is enough information in the system to cover up for that, such that other agents can do the synchronisation, if necessary. For this to work, each clone in the system needs to keep track of its own transactions. That is, it needs to store a counter that identifies the current state of the data. For this counter, it also needs to be able to produce a delta between two counter values. This may be implemented e.g. via the transaction log.

![Diagram of Multi master synchronisation counters.](image)

**Counters** Upon creation of each clone, a new counter is initialised. However, to keep track of the state of all other clones in the system, this counter is not a simple single value. Consider Figure 5.1 where two clone operations are depicted. The first clone operation resulted in three clones, the second clone operation cloned one of the clones into two. Consider the first clone operation. Each clone here gets its own counter, which is uniquely identified. For readability purposes, we chose the simple values $a$, $b$ and $c$. A unique identifier can easily be deduced from the path back to the origin for each clone. The initial state of the counter is zero, which is appended to the identifier, separated
by a colon in the figure. If both \( a \) and \( c \) perform a transaction, their counters are incremented indicating a changed state. The result is an increased revision number in the counter, thus: \( a : 1 \) and \( c : 1 \). \( b \) remains in its initial state, as it has not been changed.

Now consider the merging of the transactions among the three clones \( a \), \( b \) and \( c \). When \( a \) is merged to \( c \), \( c \) needs to process the single transaction that was applied to \( a \). In the most trivial case, this transaction is not conflicting at all. Applying the transaction hence is simple and \( c \) would be up-to-date with \( a \). However, even though a transaction does not conflict upon merge, it can yield in problems after the merge. Assume transaction \( a : 1 \) consists of \( p = 5 \) and that transaction \( c : 1 \) consists of \( p = 3 \). Now regardless of at which point in time both transactions were performed, because \( a \) is merged to \( c \), the value of transaction \( c : 1 \) is overwritten, resulting in \( p = 3 \). While the decision whether this is correct or not is up to the user, we are here concerned over the consistency over the clones once \( c \) is merged to \( a \). Because of the transaction \( c : 1 \), \( a \) needs to be updated with this transaction. Merging it as before would yield in \( p = 5 \) on \( a \), since \( c : 1 \) contains this. Obviously both clones, while they are fully merged and synced, are not equal. Needless to say this is an unwanted situation that needs to be resolved.

Revisiting the merge of \( a \) to \( c \), once \( c \) has applied transaction \( a : 1 \), it needs to consider this as a sub transaction of its own current transaction as indicated by its revision counter. This makes sure that the blind write \( p = 5 \) is applied in \( c : 1 \), causing a consistent image in further merges. Still, the counter is not updated. While this is unintuitive, it is fact necessary not to do so to avoid an endless loop of merge operations between in this case \( a \) and \( c \). If \( c \)'s counter would be incremented, the merge of \( c : 2 \) to \( a \) would be of no problem, but the result of that merge would be \( a : 2 \), which then again would have to be merged to \( c \), which obviously has no extra information and causes an endless loop.

**Merge Administration**

So far we ignored how clones can determine what transactions they have merged from the other clones. To do this, each clone needs to administer which transactions from other clones they received. This is administered in the counter. Using the previous example where transaction \( a : 1 \) is merged to \( c \), the counter at \( c \) becomes \( c : 1, a : 1 \), which describes its own state, and the updates from \( a \) it got matching \( a \)'s state at that time. With this notation, it is also easier to accept that \( c \)'s revision counter is not incremented by a merge operation. The counter is extended or updated with the revision number of \( a \) at the time of the merge. Using this notation, two clones are in
sync if their counters are equal. However, this does not necessarily mean they are up-to-date.

With the extended counter form, merges can very flexibly be propagated through the system. Consider the initial state for \(a\), \(b\) and \(c\) again. After transaction \(a : 1\) completes, it is merged into \(c\). The resulting state of \(c\) becomes \(c : 0, a : 1\). Now \(c\) is merged to \(b\), this essentially merges over \(a : 1\), but \(b\) tracks it in its counter as a merge from \(c\). Hence \(b\)'s counter becomes \(b : 0, c : 0, a : 1\). Transaction \(a : 2\) can similarly be merged either directly to \(b\), since \(b\) has \(a : 1\) in its counter, or via \(c\) again, without disrupting any future merge process.

The second clone operation in the example of Figure 5.1 is performed after the system has been running for a while. The revision counter of \(a\) is \(a : 2, c : 1\) at the time of the cloning operation. The latest revision of \(a\) is not yet merged over the entire system. The clone operation produces the clones \(d\) and \(e\), which get their own counters. However, they are copies of the original \(a\), hence they inherit \(a\)'s transaction log. This allows them to bring other clones in the system up-to-date with the last transactions of \(a\). Since \(a\) ceases to exist after it has been cloned, no further transactions are applied to it. Still, \(d\) and \(e\) are part of the original \(a\). Therefore, both new clones retain the last revision of \(a\) in their own counter, such that other clones can sync up to the last changes of \(a\) before it was cloned. Hence, the new clones start with the counters \(a : 2, d : 0\) and \(a : 2, e : 0\). For the system the new clones are equal to the others, and \(a\) appears as a clone that is never updated any more. Since it does not exist any more, it also does not require to be updated.

**Clocks** Our described counters strongly resemble vector clocks [50]. These clocks are an improvement of Lamport’s virtual time concept [44]. Here time is reduced to the simple notion of “happened before”, which means as much as for each two events a causal relation exists (\(a\) happened before \(b\)) or they are unrelated, which says nothing about when the events happened in a physical clock world. The causal relations are caused by messages between processes, in Armada’s case propagations of updates. In the vector clocks, each process keeps a vector of counters, for every process in the system a counter in the vector. Messages between processes transfer the entire clock vector, such that not only the counter of the sending process is transferred, but also those of that were once received by the sending process.

It is not hard to see that in our described system of transaction counters, we also have vectors keeping the counters of the other sites in the Armada. Essential difference here, however, is that our vector are not of a fixed size, and
it is not known how large they have to be. In fact they may differ per site. In practice this does not change much in the vector clocks, since an absent process could just implicitly mean a 0 in the vector. The clocks do not require to know the total amount of processes to determine if some vector describes a time that “happened before” another vector.

Summary

A user of an Armada system is required to be an active participant in its own query execution. This is inevitable for an autonomous and distributed system like Armada, where the high level of autonomy is only retained if clients take their own responsibilities. For the most basic tasks, a user can rely on an agent in the Armada system to help. The agent performs trivial tasks like following redirects and constructing query results. As long as no problems occur, such as unavailable sites or conflicts, the agent can do its job without asking the user.

With Armada pushing actions involving other sites to the user of the system, in case of clones in the Armada, the users need to maintain the consistency of the clones. For several reasons users may end up not doing their duty, and the agents can fail as well given they cannot resolve problems. A system to guarantee convergence is applied to all clones in an Armada. It relies on users performing actual updates, but it keeps the administration on what updates local on the sites. This system that keeps transaction counters on each clone, allows to merge updates from any clone to any other at any time, eventually reaching a globally consistent state.