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.
7 SQL Approach

7.1 An Implementation’s Architecture

In an Armada cluster, querying is done by an Armada agent. This agent bounces back and forth between nodes of the Armada when resolving queries. This way, the agent relieves the nodes from performing query tasks and interacting with their co-nodes. An important aspect of the agent is that it keeps track of the lineage trail information that it encounters during query traversals through the Armada. This particular behaviour allows the agent to start new queries on a node that suits the best based on its own knowledge. By doing this, an agent avoids needlessly querying nodes that cannot answer its question. In addition this implements a strategy to bypass the origin node in most cases, effectuating hot-spot avoidance.

Workers SQL systems currently do not have any means of providing feedback with the user. This naturally rules out the interaction part of the Armada that needs to go beyond the agent. Conceptually, Armada agents are not related to the data hosting nodes in the system. Each agent comes in the form of a special equipped database system, and needs to run on a node. Each node in the Armada may offer agent functionality next to data hosting, but this is not required. In case both agent and data share the same node, the lineage trail information between the two can be shared to increase both their knowledge. Agents interact with Armada nodes to execute queries, on behalf of a client. While agents need to be able to find the Armada nodes, clients need to be able to find an agent in order to execute a query. For agents it is sufficient to know one node of the Armada, the lineage trails point the agent to the other nodes it needs. However, for a client, such redirection is not in place. Agents can work in a pool of “workers”, where a client just gets one assigned to perform a query.
Here it is not important which agent is being assigned, and the pool may grow or shrink based on the number of client requests.

**Starting Point**  The problem yet remains how a client can get to an agent in the system, willing to do the work. There are ample opportunities here, e.g. using a DHT or an Armada structure. However, eventually they all need a starting point. Either a server that can tell what key to look for, or a first server that can point into the right direction. Hence, we assume that we just have a central server that assigns an agent to a client via a simple redirection a job that is simple enough not to become the actual bottle-neck of the system. The central locator service, maintains a pool of all active agents. A client connects to the locator and gets a redirect to an agent that handles the client’s query. This has the advantage over e.g. DNS-based load balancing that the pool of agents can change without information getting stale, such as out of date DNS entries. In addition, statistics can be collected to be used to control the agent pool size, because all clients first pass through the locator. Last, the locator could track the load of the agents and try to assign new clients to the least loaded agents or another scheme which allows for better load control than randomisation.

**Catalog**  So far we have assumed that clients connect to agents, which in their turn know how to deal with the entire “database” that is hosted in the cluster. In reality, an Armada represents at most a table, but maybe even only a column of such table. As result, many Armadas are present in the cluster, and each agent needs to know about them, to be able to query them. Next to multiple Armadas being present, there is also a relation between most of them. Several columns form a table, some tables form a schema, and so on. Also this information, usually referred to as a “catalog”, is necessary for each agent to function properly. The catalog contains for each Armada at least one node that that hosts that particular Armada — typically its origin — and all relations between the Armadas. Note that growth operations applied to each individual Armada need no modifications to the catalog. The catalog only needs to be modified, when new Armadas are added, old ones removed, et-cetera. Given that such modifications to this catalog are rare, it can be stored centrally, without performance penalties. An already central place in the system is the locator, which assigns an agent to each client request. Storing the catalog in the locator, allows each agent to retrieve catalog information by contacting the locator. However, the locator may soon get overloaded by the many requests of the agents, for this relatively static data. Hence, each agent can replicate the catalog for its own
private use. Here a consistency problem arises, but given the nature of infrequent updates to the catalog, a simple pushed based update from the locator to the agents is sufficient.

For each agent that is created to serve in the system, an initiation ritual has to be performed, such that the agent can start serving client queries. This ritual includes making itself known to the locator and retrieving catalog information such that queries can be resolved. The catalog being received is simply a copy from the locator's catalog. After this has been set up, the agent is registered with the locator, which can start assigning clients to the agent. Note that the catalog does not include the state of each Armada being referenced. The agent starts with a clean record, and as such its first queries are not well targeted. However, along the way the agent soon builds an internal representation of the Armadas it deals with in terms of cached trails, allowing for a smaller data search time. Once an update is made to the catalog, the locator is being updated. It functions as master “database” for all agents as “replicas”. The locator can send out direct updates for all the agents. However, this generates a direct load to perform all these updates, and may on the agent side conflict with pending actions. Hence, a simple delayed update strategy can be used by the locator. Once the locator assigns a client to an agent, it first sends all pending updates to the agent. Since these updates are small and fast operations the delay for the client is negligible while it performs an update of the agent just at the moment when it is necessary, keeping the catalog for the client unmodified — i.e. no transaction aborts due to catalog updates.

Using this scheme, existing agents can drop out by simply denying connections from the locator. Once the locator notices that an agent no longer responds, or appears to be entirely missing, it simply unregisters it from the pool. From that point the agent is not considered to be an agent any more, and since it is no longer in the pool, it is not assigned any client requests. In fact, the locator is not aware of its existence any more. If said agent becomes available again, it simply has to register itself again with the locator, as if it were a new agent. This means its catalog is replaced with a fresh copy from the locator. The cached lineage trails that the agent still has can be kept, as the information stored in them is never incorrect; at most out of date. This can give a reconciling agent a jump start due to its previously acquired knowledge about the Armadas.

**Catalog Updates** An update to the catalog is typically made by a client query. An operation such as creating a new table, and hence a new Armada, needs
to be stored in the catalog. If the agent dealing with such operation stores the modification in its own catalog, conflicts may arise, and other agents are not able to see the new table. Hence, updates to the catalog need to be made on the locator, and immediately synced with the agent performing the update, such that it can continue processing its client’s queries. In detail, adding a table or column requires new Armadas for those to be created. When such update request arrives at the locator, it creates new Armadas as necessary on nodes in the system, and stores those in its catalog. This procedure of finding a suitable node in the system is again supported by the locator in the system.

**Performing Operations** For each chunk or clone operation and for each Armada creation, a new node in the system has to be found. The locator maintains next to a pool of known agents, a pool of data nodes. Once a node wants to perform a chunk or clone operation, or an agent requested a new Armada to be created, the locator picks a node from its data pool. The strategies for picking a node from the pool may be based on simple heuristics such as available resources or usage statistics. Alternatively bidding procedures can take place to pick a site from the pool. In any case, the locator only maintains the pool of nodes and sends messages to each node to obtain information about their current state. Once a node is found, a new box is created on the node, and its trail stored in the catalog, effectuating the operation.

Creating a new agent in this system, is a relatively cheap operation. Basically only the catalog needs to be transferred, and the agent is ready. This makes the loss of an agent only a problem with regard to the available agents in the pool and their workload. Since the state (and involved nodes) of each Armada is not recorded in the catalog, all this information needs not to be copied. This is an advantage over systems where all information about all participating nodes is being kept in the catalog, as this requires much more data copying, as well as keeping that data consistent between the copies.

### 7.2 SQL Armada

The locator architecture lays a foundation for essentially getting a client to perform a query on an Armada. With SQL being the de-facto query language of database systems, we studied the feasibility to implement the Armada model in an SQL database. Mapping the basic metadata administration and structural operations of the Armada model to standard SQL allows us to strive after a *minimal-invasive* Armada-implementation, i.e. instead of pushing the Armada...
functionality inside existing (or new) SQL processors, we simply add it on-top. This approach does not only allow for a system-independent Armada that could even become a heterogeneous system using different DBMSs at different nodes/sites, but also helps to leverage mature DBMSs technology for local query processing.

7.2.1 Simple example

In Figure 7.1 the ideas of the Armada model are graphically represented. Rightmost in the picture, the lineage tree of all boxes is shown. From those boxes, only three are active, meaning they have data. Those boxes’ data are depicted in the middle bar, which represents the full table data, depicted on the left in the figure. As can be seen, the union of all active boxes’ data results in the original table again. In our mapping to SQL, we use this particular observation. From

The lineage trails in Figure 7.2 we can see that boxes $B_0$ and $B_2$ are not active any more, as they have successors. Boxes $B_1$, $B_3$ and $B_4$ contain the actual data in the Armada. Furthermore, $B_0$, $B_2$ and $B_4$ are positioned on the same site, $S_1$.

An Armada such as in Figure 7.1 can emerge as an evolutionary process over time. Each Armada starts as a single origin box, which upon need is chunked. Thereby, new sites can host boxes, to extend the Armada in its evolution. In the example, box $B_0$ and $B_2$ on site $S_1$ were chunked over time to make room for new data.


\[ T_0 = [\%, S_1]:B_0; \{ [f_1, S_2]:B_1 \} \]

\[ T_1 = T_0 \cdot [f_1, S_2]:B_1; \]

\[ T_2 = T_0 \cdot [f'_1, S_1]:B_2; \{ [f_2, S_3]:B_3 \} \]

\[ T_3 = T_1 \cdot [f_2, S_3]:B_3; \]

\[ T_4 = T_1 \cdot [f'_2, S_1]:B_4; \]

Figure 7.2: Sample Armada lineage trails.

### 7.3 Armada Evolution

In a typical database environment, data is available in tables. These database objects are subject to modifications, in particular growth. The Armada model supports these modifications in the case of growth by splitting a table into multiple tables. This can happen when this is required, and Armada keeps the administration. An important aspect within the Armada model here is that each site is considered autonomous. This means that such split is a decision in agreement with the sites involved in the split, and can be initiated by the site hosting the data.

This autonomy reflects in our view on the role of the client within an Armada. Unlike conventional clients, in Armada a client is active. This means for a client that it plays a central role in query execution. Instead of sites contacting other sites during query execution, the active client bounces between the sites.

Due to the autonomy of the sites, they mainly manage themselves. A site can decide that it needs to chunk its data when resources get scarce. Or, when the load is high, to clone its data. These operations can happen whenever the need arises to do so, without intervention of a client as long as sites are available and willing to co-operate.

**Table Chunking** Simply put, the Armada model allows to break up a table in two (or more) sub tables. This split is done based on a chunk function that specifies what part of the original data goes into which of the new tables. After a break up of a table, two new tables are produced and the original table is removed. References to the original table, be it in the system itself, or by users, do not work any more after this operation. To solve this, one of the new tables can be renamed to the original table. This does solve the table from being
unavailable, but introduces confusion as the new table obviously doesn't hold the same data as the original one. Alternatively, all references to the original table can be updated to point to the new ones. This might be hard to achieve (e.g. humans remembering the original name) and undesirable, as now two tables have to be used instead of the original one.

Finally, to overcome the problems imposed by the previous two solutions, a special object can be provided to indicate that the original table was broken up in two. The function of this object is to be both transparent and purely informative about the break up. The object itself simply represents the union of the two sub tables, like the inactive box in the Armada model. It is quite naturally covered in SQL by means of views.

7.4 Seamless Armada-SQL Integration

With the design for a self-managed distributed Armada system at hand, we now turn our attention towards its implementation. Striving after a minimal-invasive and widely portable solution, we now describe how to map the basic metadata administration and structural operations of the Armada model to standard SQL.

7.4.1 Chunking

Each Armada starts at a single site, as a single table. In the SQL world this is an ordinary relation, e.g.:

```sql
CREATE TABLE treasures {
    bag int,
    name varchar(64),
    coins int,
    CONSTRAINT treasures_bag_pkey PRIMARY KEY (bag)
};
```

This simple table keeps track of some treasures on site $S_0$. Each bag has a number which is unique, as defined by the primary key. The Armada model defines that this table can be broken up in two or more new tables. This has the effect of the data being spread over these new tables, which represent the boxes from the Armada model. Breaking up the original table in two, is done as follows.

First the new tables (boxes) have to be created. The creation is followed by insertion of data from the original table. The creation of the box is like the schema of the original table using the following template:
7.4. SEAMLESS ARMADA-SQL INTEGRATION

CREATE TABLE {armada}_B{boxid} (  
{columndefinitions},  
CONSTRAINT {armada}_B{boxid}_{pkey}_pkey  
PRIMARY KEY {pkey},  
CONSTRAINT {armada}_B{boxid}_{pkey}_check  
CHECK ({if:contrachunk}NOT{fi}  
{armada}_F{funcid}({pkey})))  
);

In this template, variable parts are wrapped between { and }. The new tables are named after the original table, {armada}. To distinguish them they have a trailing _B{boxid} which includes the unique id of the box (table). In the template, an extra CHECK constraint is added that checks the primary key {pkey} on validity for the box. A key is not valid, when the used chunk function {funcid} does not hold. Two new tables are created: one that holds the data that matches the chunk function and one that holds the data that does NOT match. The {columndefinitions} and {pkey} are simply inherited from the original table’s schema. For the original treasures table, this template results in the following create statements:

CREATE TABLE treasures_B1  
  
  bag     int,  
  name    varchar(64),  
  coins   int,  
CONSTRAINT treasures_B1_bag_pkey PRIMARY KEY (bag),  
CONSTRAINT treasures_B1_bag_check  
CHECK (treasures_F1(bag))  
);

CREATE TABLE treasures_B2  
  
  bag     int,  
  name    varchar(64),  
  coins   int,  
CONSTRAINT treasures_B2_bag_pkey PRIMARY KEY (bag),  
CONSTRAINT treasures_B2_bag_check  
CHECK (NOT treasures_F1(bag))  
);

The next step is to fill the two new boxes with data from the original table, such that that table can be removed in favour of the two new ones. The following template applies:

SELECT INTO {armada}_B{boxid}  
SELECT {columns}  
FROM {armada}  
WHERE {if:contrachunk}NOT{fi}  
{armada}_F{funcid}({pkey}));

Via an ordinary SELECT INTO statement, only the data from the original table that matches the boxes is inserted. Many fields in the template are the same as
before. Because the templates used are abstract, we illustrate their functioning with an example:

```sql
SELECT INTO treasures_B1
  SELECT bag, name, coins
  FROM treasures
  WHERE treasures_F1(bag);

SELECT INTO treasures_B2
  SELECT bag, name, coins
  FROM treasures
  WHERE NOT treasures_F1(bag);
```

Copying over the data is trivial and includes the same functions as the tables’ `CHECK` function. Since the data was copied over to the two new boxes, the original table can be dropped.

```sql
DROP TABLE treasures;
```

Finally, since the original table `treasures` is dropped, it disappears, as mentioned above. The special object that represents the union of the two new boxes, is implemented by a view:

```sql
CREATE VIEW {armada} AS
  SELECT {columns}
  FROM (    
    SELECT {columns}
    FROM {armada}_B{sucboxid}
    WHERE {armada}_F{funcid}({pkey});
    UNION
    SELECT {columns}
    FROM {armada}_B{contrasucboxid}
    WHERE NOT {armada}_F{funcid}({pkey});
  ) AS {armada};
```

Note that no explicit typing information is included in the view: the SQL standard does not allow us to explicitly encode it, hence the database system has to get this information based on the local box referred to in the view. The variables `{sucboxid}` and `{contrasucboxid}` refer to the box ids of the created two new tables. In the example:

```sql
CREATE VIEW treasures AS
  SELECT bag, name, coins
  FROM (    
    SELECT bag, name, coins
    FROM treasures_B1
    WHERE treasures_F1
    UNION
    SELECT bag, name, coins
    FROM treasures_B2
    WHERE NOT treasures_F1
  ) AS treasures;
```
7.4.2 Cloning and Combining

So far, we focused on the chunk operation of the Armada model. Supporting the clone and combine operations is less trivial, but conceptually possible. Consider a clone operation to take place like previously done for the chunk operation. First the two boxes are created, but unlike the chunk operation, they have the same function as their predecessor. Hence, a clone of box $B_1$ from the example above results in the two boxes $B_3$ and $B_4$:

```sql
CREATE TABLE treasures_B3 (
    bag int,
    name varchar(64),
    coins int,
    CONSTRAINT treasures_B3_bag_pkey PRIMARY KEY (bag),
    CONSTRAINT treasures_B3_bag_check
        CHECK (treasures_F1(bag))
);  
CREATE TABLE treasures_B4 (
    bag int,
    name varchar(64),
    coins int,
    CONSTRAINT treasures_B4_bag_pkey PRIMARY KEY (bag),
    CONSTRAINT treasures_B4_bag_check
        CHECK (treasures_F1(bag))
);  
```

Next, the `treasures` view has to be made available. Here it is, unlike at the chunk operation, not trivial how to express the functionality of the clone operation in SQL. Where the union of the resulting boxes of the chunk operation results in the original box, also for the clone operation the union operator yields in the same behaviour.

Recall that as defined by the SQL:92 standard and up, the `UNION` operator eliminates duplicates by default. Hence, the union of two clones results in the predecessor of the clones. While the functionality of a clone is to provide an alternative during the query execution, the union operator feels quite unnatural. However, SQL does not have any means to express the OR operation on data sources, as to pick an alternative. Hence, using the `UNION` operator, a semantically correct SQL view can be created:

```sql
CREATE VIEW treasures AS
    SELECT bag, name, coins
    FROM (  
        SELECT bag, name, coins
        FROM (  
            SELECT bag, name, coins
            FROM treasures_B3  
            WHERE treasures_F1  
        UNION
    
```
SELECT bag, name, coins
FROM treasures_B4
WHERE treasures_F1
} AS treasures_B1
WHERE treasures_F1
UNION
SELECT bag, name, coins
FROM treasures_B2
WHERE NOT treasures_F1
} AS treasures;

In this view the execution has to query both clones and union the two (identical) answers. While this action does defeat the use of the clones, it is correct from a generic execution point of view. There is room for improvement and optimisation here, which we discuss in a follow-up paper.

Analogous to the previous discussion, combine operations also use the UNION operation in SQL. In particular the duplicate eliminating nature of the combine operation when applied to data overlapping boxes, is very well covered by the SQL UNION operator. The combine operation does not turn one box in multiple new ones, but instead merges multiple boxes into one.

Assume we merge box $B_2$ and $B_3$ from the previous examples into $B_5$. This effectively means a merge of a chunk and a clone of its counter chunk are being merged. Creation of box $B_5$ includes the logical combination of its predecessor functions:

CREATE TABLE treasures_B5 (
  bag int,
  name varchar(64),
  coins int,
  CONSTRAINT treasures_B5_bag_pkey PRIMARY KEY (bag),
  CONSTRAINT treasures_B5_bag_check
  CHECK ((NOT treasures_F1(bag))
  AND treasures_F1(bag))
);

It is not hard to see that the combination of the functions used for the predecessors yields in the whole coverage as for the origin box in this case. It is more probable that this is not the case, however. Next, the creation of the treasures view is in contrast to the chunk and clone operations different in that it simply unions the respective views of the predecessor boxes and substitutes the predecessor boxes with the new box $B_5$. This action represents the combination made of the two input boxes:

CREATE VIEW treasures AS
SELECT bag, name, coins
FROM {
  SELECT bag, name, coins
  FROM treasures_B1
WHERE treasures_F1
UNION
SELECT bag, name, coins
FROM treasures_B5
WHERE NOT treasures_F1
) AS treasures
UNION
SELECT bag, name, coins
FROM (SELECT bag, name, coins
FROM (SELECT bag, name, coins
FROM treasures_B5
WHERE treasures_F1
UNION
SELECT bag, name, coins
FROM treasures_B4
WHERE treasures_F1
) AS treasures_B1
WHERE treasures_F1
UNION
SELECT bag, name, coins
FROM treasures_B5
WHERE NOT treasures_F1
) AS treasures;

The resulting view is large and contains some redundancy. However, we do not optimise the view at this stage. The only action taken here is to rename the predecessor boxes in the view into the newly created box, as they are now covered by the new box by definition.

7.4.3 View inlining

With the current technique, once created views are never updated again, leading to inevitable levels of indirections when trying to resolve them. To avoid this, a simple and relatively cheap operation can be applied on the views to update their definition by inlining the definition of the view they (partially) point to.

Consider again the previous example, with on S₀ the original treasures view which is a simple selection on box B₀. At the time of the chunk operation applied to B₀, the new definition for B₀ (as in the newly created view for it) can be inlined in the treasures view. For this, the reference to B₀ is replaced with the sub-query for it, resulting in the same view as on S₁ or S₂. In a similar way, inlining during the chunk operation in the transparency view on S₂, results in one level of indirection less, because it becomes the same as the treasures view for S₃ or S₄. In general, the transparency view can be updated to equal one of the successors, since that view was constructed using the same rules.
Unless the inlining is done cascading, the effect of this inlining is limited. Since performing this inline operation for each chunk operation to the origin site eventually results in a lot of traffic, this is not desirable. Instead, we envisage an inlining on demand, taking place during the querying process.

### 7.4.4 Improvement on the SQL implementation

The drawback of the previously introduced mapping of the Armada model to SQL, is that for each box, it needs a (physical) table in the database. This requires data to be physically grouped by the chunk function that describes it, and forces moves of tuples from and to the box tables upon chunking operations. In particular the copying from and to boxes on the same site is “expensive” because in principle the data doesn’t physically “move” but needs to match the right box. In addition it introduces a space requirement (during the copy) which might be problematic for large boxes, or on machines scarce on resources.

A solution to the drawbacks sketched above, is to detach the actual storage from the box structure, and have the boxes being represented by a view instead. This allows to basically store the data in any table, or even multiple tables, as long as the box can be expressed by a view. It also gives the freedom to construct boxes from existing data by just defining views for them, instead of having to create them in an evolving matter. This feature in particular comes in handy when data from multiple sites is merged into one Armada instance.

The implementation of this solution is done using one table in the armada schema, named data. Since all data on a site is stored in that one table, there is now the need to create views that map back to the boxes again. Another table in the schema is introduced, the table box. This table simply holds the box IDs and their corresponding constraint as a string. This allows to use complex SQL functions, or simple conditions as checks in e.g. the WHERE-clause of a SELECT query.

### 7.5 Experimentation

Construction of Armada over relational systems stresses their capabilities to manage ever growing views during cloning, chunking, and combining. To assess this impact we conducted small experiments against MonetDB and PostgreSQL. We were unable to test MySQL as it does not support SQL standard syntax.

We can distinguish five different activities and related costs in the SQL approach taken by Armada.
7.5. EXPERIMENTATION

Table creation For each box, a table is created within the system. Because original tables get fragmented, extra tables need to be created. This may be an expensive operation. However, due to the expected frequency and other operations involved in chunking, of which table creation is a part, the impact of this cost is low.

data insertion To actually fill up the system, data is inserted in the boxes. Since this insertion itself is not any different from normal insertion, but that the site to insert to changes every once in a while, this cost is expected not to be any different from normal.

data moving On chunk operations, a part of the original box may be moved to another site. This involves a two-staged “copy and remove” action, with proper isolation and transportation of the data. This cost is obviously only involved if there is distribution by means of remote sites.

view creation Like table creation, view creation happens during the chunking process, and its costs get overshadowed by other operations of the chunking process. Hence, the impact of this cost is low.

view execution The generated views which cover the complete original tables become large. Query optimisers and planners may choke on such large views. Execution may become expensive due to the many tables (boxes) involved. However, since these views are the main route to query the system, the importance of the costs associated to them is high.

Query Complexity We conducted a number of experiments on our proposed SQL implementation of the Armada model. In particular the number of sites participating in Armada and the number of boxes that are created were taken as variables. The number of sites participating influences the necessary (total) communication. The number of boxes stresses the administration, and challenges its effectiveness.

As the number of sites that are populated by boxes increases, the need to contact other sites increases on average as well, as seen in Chapter 6. Such increase obviously comes with additional network costs. However, an increase of sites not necessary means an increase of communication. If queries target data that is located on one site, obviously not much communication is necessary. Moving targets, such as the last events (days) of a log file, typically are in a growing instance. However, ideally queries after such targets never need to consult more than one site.
With numerous boxes, the lineage of new boxes grows. Also, traversal paths may get very long in case a box has to be searched for. Limitations of the used SQL engines make it impossible to do any remote activity. Instead, we simulate everything local to the database, hence only being able to see query complexity and local execution. It also adds additional requirements as becomes clear later on.

**Local Consistency**  When boxes get chunked, their contents is moved to other boxes. Hence, the original box remains empty and should be a pointer to the new boxes that were created. Conceptually, this is easy, as the chunked box can be replaced by an SQL view. However, databases that require their views to be correct, disallow tables that are used in a view to be dropped.

For this, first the view(s) that use the box should be removed. In Armada this yields in a problem. Ideally, the inactive boxes should just be replaced by a view, but the database forbids this action to take place. Of course, this only happens when the database has both the views and the boxes they depend on local, as only then it is easy to check. Unfortunately, this is always the case, because the view that describes the whole system, also describes the box that is chunked. However, if this view, the *worldview*, gets updated after the chunk operation, it should only list the offspring of the chunked box. So what remains pointing to the inactive box, is the worldview of the parent of the chunked box. Whenever this view is on the same site as the inactive box, the database can locally check and see that dropping the box breaks the dependency of that view. Hence, the only way out, is to recreate that view too.

Because this touches the view anyway, it is as expensive to immediately “inline” the new definition of the inactive box, instead of temporarily dropping it, replacing the box with a view and recreating it. When the parent is on a remote site, a view for the inactive box needs to be created, such that the view on the remote site still points to some object on the local site.

**Table Replacement**  When, simulating an instance of Armada on a local site, local consistency checks influence how the operations take place. A DBMS checks the dependencies of objects that are in its catalog. Not only can it check for each foreign key constraint which table is necessary, but also for views it can determine which tables are used. An effect of this is that a table that is used in a view cannot be removed before that view is removed. The same holds for when a view is to be removed that another view uses.

With Armada, views get created to represent disappeared tables. When a
box gets inactive, its corresponding table is replaced by a view. To do this, first the table is removed, then a view put in place. While the end situation in theory should be fine, the replacement of the table by a view is impossible from the DBMS’s point of view. This is caused by Armada creating worldviews which are a view on the whole instance of Armada. As such, each table is part of one or more worldviews. Removing the table then breaks the dependencies of those worldviews. Replacement of the table by a view is not possible without the DBMS noticing in a local situation. If remote sites are referenced, the DBMS cannot see another site depends on an object, unless back references a maintained.

For the local situation, the worldviews that depend on the to be dropped object have to be dropped temporarily. Typically, for a given table, all worldviews that are on the site of the table itself, its sibling and all of its successors have to be dropped, as they point to the table being replaced. In addition to this, all views for the tables of the parent trail need to be dropped as they recursively depend on each other. Remember that a table is replaced with a view to its successors. When one of the successor tables is replaced with a view, it has the same problem as with the worldview. However, when the successor has been replaced by a view, there is a view to view dependency.

This is the case for every box in the parent trail of each table, since every box in the parent trail is inactive. This means that in order to be able to drop a view for an inactive box, its parent has to be dropped first, which recursively goes back to the root of the Armada instance. Hence, each operation in Armada, when simulated on a single site, is very expensive in terms of view destruction and creation.

**Experiments** An instance of Armada is a large collection of views and a few tables that hold the actual data. Not only are there many views, but also are the views very large in size, containing many levels of nested queries. In addition, many views depend on other views in the system. For these reasons, an instance of Armada is challenging in that it puts quite an unusual load on the DBMS.

We conducted a few experiments on a growing instance of Armada, to measure the effects of the views on the DBMS. Since we are not interested in the actual data load, but in the load of the metadata administration, we chose to use boxes of a small size containing between 5 and 15 tuples. With this size of the boxes, an instance of Armada grows relatively fast. In Table 7.1 the number of boxes created per number of tuples, and the inactive boxes is depicted in the first three columns. Inactive boxes in the SQL simulation are views, that due
to the aforementioned dependencies are recreated over and over again. As a result the 1000 tuples run contains 5184 view creation statements.

**Origin Querying** For each site, its worldview addresses every box in an instance of Armada. The difference in worldviews per site is in the amount of encoded lineage information. Sites with a larger parent trail, have more parent boxes encoded in their worldview. This is an advantage over sites with smaller parent trails, such as the root, as they have to “resolve” many boxes through views that point to views of their successors. For this reason, the origin is the most expensive cost-wise when querying its worldview. Hence, to get an idea of the maximum load introduced by the metadata administration of Armada, we measure query times on the origin site.

Querying a site is done with three different queries. First, a query that spans all (active) boxes is being run. It consists of a simple count of all tuples in the instance of Armada. Second, a range select query is performed. This theoretically allows the DBMS to skip a number of tables. Finally, a point query is performed. This always spans just one box in our experiment, since the key is required to be unique.

Table 7.1 lists the loading and query times for MonetDB/SQL 5.0.0_beta1 and PostgreSQL 8.0.8. The loading times are given in seconds in the columns $M_1$ and $P_1$ for MonetDB and PostgreSQL respectively. The columns $M_1$, $M_2$, $M_3$ and $P_1$, $P_2$, $P_3$ represent the average of 5 runs, preceded by 2 runs to assure the data is loaded and ready to be processed by the DBMS. The execution times are wall-clock times specified in hundreds of seconds.

<table>
<thead>
<tr>
<th>tuples</th>
<th>boxes</th>
<th>inact.</th>
<th>$M_1$</th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>37</td>
<td>18</td>
<td>3.50</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>7.21</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>71</td>
<td>35</td>
<td>13.68</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>23.29</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>750</td>
<td>107</td>
<td>53</td>
<td>39.80</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>67.15</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1000</td>
<td>143</td>
<td>71</td>
<td>84.68</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>142.04</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1250</td>
<td>179</td>
<td>89</td>
<td>178.07</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>245.24</td>
<td>13</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1500</td>
<td>215</td>
<td>107</td>
<td>350.07</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>384.32</td>
<td>18</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1750</td>
<td>255</td>
<td>127</td>
<td>614.58</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>590.16</td>
<td>24</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2000</td>
<td>311</td>
<td>155</td>
<td>1131.23</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>956.27</td>
<td>31</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2250</td>
<td>367</td>
<td>183</td>
<td>1852.58</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>1490.17</td>
<td>40</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2250</td>
<td>simple</td>
<td></td>
<td>7.73</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.89</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.1: Experimentation details.
7.5. EXPERIMENTATION

Transaction Settings  By accident the original loading set we created did not include transaction boundaries. Hence, the default transaction mode was used instead. In this mode after every statement a commit is issued. This was causing very long loading times for MonetDB. To solve this problem, the loading set was changed to include transaction boundaries, such that the whole load was seen as one transaction. This allowed us to compare the query results of PostgreSQL with MonetDB.

Apart from the loading times, which are dominated by view destruction and creation, the execution times on all of the sizes used in the experiment are far below a second per query. Yet they can grow up till 40 times the time needed to query a normal table in the case of a full scan in our largest example.

Even though the SQL statements to produce the instance of Armada do not contain any check constraints yet, PostgreSQL is able to avoid a full scan in case of range or point queries that only address one or a few of the tables.

Loading  By observing the loading times, we can note that both PostgreSQL and MonetDB have difficulties with loading larger jobs, in two cases MonetDB actually crashes under the load. Jobs are generated by specifying a sitesize and a tuplecount. The former specifies how many tuples fit on a single site, where the latter specifies how many tuples are inserted in the instance of Armada. The jobs are a run with sitesize = \{10, 50, 100, 500, 1000, 5000, 10000, 50000\} and tuplecount = \{10, 50, 100, 500, 1000, 5000, 10000, 50000\}. Not all combinations are generated, because of limitations in the generator. The avoided runs are those where the active box ratio (tuplecount divided by sitesize) is equal to or exceeds 1000. Note that the active box ratio is for many jobs the same. In such case the sitesize differs, which allows us to gain some insight on the effect of different data volumes.

While the loading times were not repeatedly measured, they are wobbly and must be read as a vague indication only. What we can conclude, however is that equal ratios take longer to load with more data, as one would expect.

Comparisons  The graphs that depict the wall clock times for loading and the three queries, clearly show that for both MonetDB and PostgreSQL the loading times dominate over the query execution times by far. This doesn’t come by a real surprise of course. Two peaks can be noted, for both databases. These peaks are at tests 10 – 5000 and 100 – 50000, which both have a ratio of 500. Obviously the amount of data involved makes a difference. For PostgreSQL this 10 times more data results in an almost 2 times longer loading time, while
MonetDB only needs about 10 seconds more, roughly $1/6^{th}$ of the loading time. If we compare the loading times between the two DBMSs, as in graph 7.3, we can see that there is not a huge difference between the two. Apart from the two peaks discussed before, there is one extra peak for $500 - 50000$, ratio 100 that does not run within 10 seconds. The tendency here is that MonetDB is slightly faster, though not necessarily in all cases, such as e.g. $10 - 1000$. However, this might be very well be explained by startup times or similar, since the loading tests were not run multiple times in a row. Note that MonetDB crashes during loading for the $10 - 5000$ and $100 - 50000$ workloads.

**Querying**  Query 1 consists of a count over all tuples in the relation over the system. This involves a scan over the full relation to count the tuples. The query and the results can be found in Figure 7.4. PostgreSQL clearly falls behind MonetDB in this phase, which is not surprising given the nature of both database systems. Note that there are no results for the workloads $10 - 5000$ and $100 - 50000$ since the loading fails.

The second query does the same count as the first query, but with an additional range predicate. The predicate is sufficiently small to only address a fraction of the Armada in most cases. The results from Figure 7.5 show a different
Figure 7.4: Query 1.
trend compared to the first query. Here PostgreSQL clearly outperforms MonetDB on almost every query. It looks like PostgreSQL is able to recognise only a limited number of tables is necessary to perform the query, whereas MonetDB seems to have equal performance as for query 1. This indicates it calculates the full Armada table, and performs the selection on that table to calculate the requested count.

The last query is a point query, targeting a single box. MonetDB not only falls behind PostgreSQL here, but also crashes on the $10 - 500$ and $100 - 5000$ workloads. Like for query 2, PostgreSQL is able to cut down the query time by identifying that the query only addresses a single table.

7.6 Limitations

Even though the SQL approach taken here deals with growth and querying, it is not a generic solution for a number of reasons. First and foremost does the SQL approach require a remote catalog/query mechanism to be available. As indicated before, not all databases have this functionality, and the many commercial databases have this available in some way, do not necessarily provide it in a form that is usable for Armada. PostgreSQL, MySQL, SQLite and MonetDB do not have said functionality to perform remote queries as of this writing. This observation makes a true heterogeneous implementation of Armada with the proposed SQL implementation impossible without a middleware layer to cater for the missing functionality.

A second issue is updating the data, like performing SQL \texttt{INSERT}, \texttt{UPDATE} and \texttt{DELETE} statements. The view constructions built by the SQL approach allow in principle for specific location of where updates should take place, in the same way as selects over the data can be located. However, updates of this kind over views is to the best of our knowledge not available. To a very limited extent, updates on views are possible in most commercial database systems. The typical functionality here is to allow updates on views that do projections and/or selection only and updates that only involve one of the tables used in the view. Since the Armada SQL model does \texttt{UNION}s of tables, execution of updates on those views require some (semantic) knowledge behind the views and their purpose in Armada. Also this could be solved using a middleware approach, but it would require a full SQL interpreter in the middleware software, including the Armada logic. Such approach would not require any generic (SQL) approach, since the middleware translates all functionality to the target database. This was not the focus of the proposed SQL approach.
Figure 7.5: Query 2.

SELECT count(*) FROM S0 armada WHERE key < 50
SELECT * FROM S0_armada WHERE key = 105

Wall Clock runtime (seconds)

Figure 7.6: Query 3.
### Table 7.2: Extended experiment times.

<table>
<thead>
<tr>
<th>Workload</th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.05</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.005</td>
<td>0.14</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.01</td>
<td>0.135</td>
<td>0.01</td>
<td>0.01</td>
<td>0.04</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.09</td>
<td>0.1475</td>
<td>0.01</td>
<td>0.01</td>
<td>0.17</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 7.2: Extended experiment times.
Lastly, for operations to be executed somewhat efficiently, and to have autonomy in the sense that the databases are not depending on each other, each database must support to have views with “unknown” non-local objects that can only be checked by contacting a remote site. That check, however should not be performed for each and every operation, but instead left to the agent in the system to discover.

A final note on the viability of the SQL approach deals with optimisations specific to Armada, in particular originating from the knowledge stored. For queries that just address a particular range that can be matched upon certain boxes, it is evident that querying all boxes in the Armada is a waste of resources. To avoid this waste, execution should skip boxes that cannot contain requested data. For this, the chunk functions need to be used at the execution to skip contacting remote sites were this is deemed impossible.

Summary

Using the SQL language, we aimed for a straight-forward and relatively non-intrusive implementation of the Armada model in existing database management systems. In particular SQL views are a central concept within the chosen implementation approach. We demonstrated how the chunk, clone and combine operations of the Armada model can be mapped onto SQL, and which actions are required to perform the respective operations.

The distribution aspect of Armada is hidden under the assumption that there is a notion of “remote tables”. This notion allows a database to retrieve data from a table which is on a remote site as if it were local to the database. This functionality is only available in a limited set of traditional databases to date, to the best of our knowledge. Due to the nature of our implementation, Armada can be easily simulated on one machine where all boxes are local first. Whenever remote tables become available, the Armada implementation can easily switch to using real distribution of the data.

The two database systems that we have experimented with, have shown that the approach taken puts an unusual load on the systems, sometimes even resulting in crashes. While the construction of an Armada takes a lot of time, this can be accepted in general, since it is not a frequently occurring operation. The query performance instead is important. Compared to a single large table the fragmented Armada is slower, especially when all boxes in the Armada need to be consulted. Given that it is more probably that this is not the case, the results are acceptable in that area. SQL engines can be optimised for a more
efficient use of large SQL views. Crashes under the load produced by the queries and views are of course a bad sign that the underlying engine is not able to handle the size of the queries, which needs to be tackled first.

Loading succeeds using some tricks, and querying works reasonably well. However, inserting and updating data does not work due to limitations on the SQL views observed on many SQL implementations. The loading tricks also affect the matter in which a database can freely evolve, as in extend itself to others in the cluster. Additionally, the SQL implementation does not allow for the active client model, but turns each server into a recursive query resolver itself. This obviously breaks the autonomy as defined by the Armada model.