Uppsala Master's Theses in
Computing Science 243
Examensarbete DV3
2003-07-07
ISSN 1100-1836

Wrapping External Data by Query Transformations

Martin Hansson

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

Amos II is an object-oriented database mediator system with its own query language, AmosQL. Amos II permits the definition of *mediator databases*, which are object-oriented virtual databases (views) of combined data from Amos II local database or other Amos II mediators. It is possible to define *foreign functions* in AmosQL, which can be implemented in a general-purpose programming language such as C, Java, or Lisp.

The need frequently arises to also include other data sources that can be accessed through the mediator. This is done by defining *wrappers* for these data sources. A wrapper can be said to consist of an *interface* to the data, implemented foreign functions in AmosQL that answers queries in the data source's native query language, and a *translator* which translates a query in AmosQL into one or more calls to the functions in the interface. The translator API lets a wrapper implementor define the capabilities of the data source, and the translator uses a combination of cost-based and capability-based rewrites of the query in order to minimize the total cost of accessing the particular data source.

As a proof-of-concept a wrapper for relational data that uses JDBC was implemented. The results are based on experiments comparing execution times for translated and non-translated queries.

Supervisor: Tore Risch
Examiner: Tore Risch

Passed:

Contents

1 Introduction

The UDBL group at the department of information technology of Uppsala University has developed a database management system called Amos II. The system consists of communicating nodes connected in a peer-to-peer network where data is distributed among different nodes. Such a network is called a *federation* of database nodes. Its nodes can act as either clients interacting with a user, database servers, or *mediators*. Mediators appear as database server to clients, and as clients to servers. To a user connecting to a mediator it appears as if the mediator contains all the data, but it is transparently connected to a data source storing data in the federation, or to another mediator. The data may also be distributed among several servers. The protocol used for communication is tcp/ip, which allows for the mediators to be distributed over the Internet.

Queries are optimized in order to minimize the total cost of accessing the particular database holding the information. A mediator is dedicated to forwarding a sub-query from another Amos II to one or several Amos systems (which can also themselves be mediators.) If necessary the mediator will split the query further into sub-queries and combining the result for returning to the calling Amos II. If the data is locally present it will simply return the information.

Amos II uses an object-oriented query language called AmosQL. While it is believed that the object-oriented approach will gain in popularity in the future, the relational model still proves sufficient for many applications and many databases are relational and will stay that way. Therefore there is the need to incorporate relational data sources in a federation. This is done by wrapping the source inside an Amos II system so that to the outside world it will appear as though it were an object-oriented database system. Furthermore, the need frequently arises to wrap other data sources within Amos II, for example visual data or CAD models. A wrapper is thus a general concept that should be easy to implement as a separate feature that is simply connected to Amos II through a narrow and well-defined interface. Wrappers consist of two parts: an *interface* which accesses the data and meta-data at the data source with a set of functions, and a *translator* that translate queries in AmosQL into calls to the interface.

In the past wrappers had to be added to the Amos II kernel in an ad-hoc manner and deep knowledge of the internals of the code was necessary. The result of this work is to find a more general way of writing wrappers using only basic parts such as an interface to the source and a relatively simple translator which translates the internal representation of the query into a query in the source's native language, taking the maximal advantage of the data source's capabilities.

2 Related work

There is a rising need in the industry and the computing world at large to be able to integrate data from diverse sources, and being able to ask queries

over heterogeneous repositories of information. Until recently, few commercial systems have attempted to address this issue, but the problem of wrapping external data is receiving greater attention. IBM have developed a wrapper definition extension to their product relational database manager DB2 called Garlic[4], which bears a lot of resemblance to this work. Garlic extends DB2 with facilities to access data from other sources as if they were relational tables. DB2 uses a relational data model, whereas Amos II uses an object-oriented data model. (see section3.1.)

Garcia-Molina, Ullman, and Widom[7] discuss using templates to query the data source. Templates are reminiscent of prepared statements in SQL, in that they are finished queries with parameters that can later be set, and the source is considered capable of a query if the query matches the template. This work uses a somewhat different approach. Any access to a table in a wrapped database is carried out through functions defined in terms of an interface function producing the extent of the table. The definition of the extent function is in-lined by the optimizer. The capability is to retrieve the entire extent of a source.

Garcia-Molina, Ullman, and Widom further introduce the concept of capability-based rewrites which is very similar to this work. However, they have an assumption that mediators will use SQL as their common data model, which is not the case with Amos II. In addition, they have a richer syntax for describing capabilities than the capability description contained in this work. (see section 4.4.) In Amos II it is only possible to specify limitations on an argument to a query so that the value must be specified for the argument (a constant input argument), that it is forbidden to specify it (an output-only argument), or that it may be left unspecified if desired (input or output). Garcia-Molina, Ullman, and Widom have two more capability models to choose from, either the argument must be chosen from a finite set of options, or that it can be left unspecified but if specified it must be chosen from a set.

3 Amos II

This section gives an overview of the Amos II system for which the wrapper definition API was written. The Amos II system is based on the IRIS functional data model [3].

3.1 The Amos data model and query language.

3.1.1 Objects and Types

Amos II is a functional, object-oriented database system with a high level of generalization. Every entity in the data model is represented by an object, and objects are members of one or severals types. Types are used to classify objects, and every object is an instance of one or several types. The *extent* of a type is the set of all instances of the type (see section 3.1.2.) Types correspond directly to the concept `class` in object-oriented programming languages.

Types are organized in a subtype/supertype graph where the most general type is `object`. All other types are subtypes of `object`. Even types are represented as objects, and these meta-objects are instances of the type `type`.

3.1.2 Subtypes and Extents

Objects are classified into types, thus making each object an instance of a type. If an object is an instance of a type t, it is also an instance of all the super-types of t. A type can have several super-types either by inheriting several types directly (multiple-inheritance,) or by being a subtype of a subtype (transitive closure.) The extent of a type is the set of all instances of the type. In the face of inheritance, however, the term extent is ambiguous. Amos II uses the following definitions

shallow extent the shallow extent of a type t is the set of objects that are instances of this type. If t' is a subtype of t only, then the shallow extents of t and t' are disjoint.

deep extent the deep extent of a type t is the set of objects that are instances of the type, union-ed with the shallow extents of all subtypes of t. The deep extent of a subtype of t is always a subset of the deep extent of t.

The term extent is used throughout this paper, and when nothing else is stated it should be taken to mean the deep extent.

3.1.3 Functions and Extents

Crucial in the data model is the *function* concept. They model the properties of objects and relations between objects. Functions are objects too, and they are instances of the type `function`. All functions adhere to the DAPLEX semantics[6], which means that a function may have several return values, i.e. it will *emit* several results. This is normally referred to as *bags* and is denoted with braces and bars: $\{|\ldots|\}$. For example a function that computes the square root would under this semantics be defined as $sqrt\, x = \{|y, -y|\}$.

A function can be implemented in three ways, it can be *stored, derived* or *foreign.* A stored function has its extent stored in the mediator itself. Formally, it is defined as the set of tuples that make up the arguments and the results of a function. For example, the extent of the function defined as

```
create function name(person p)-> charstring name as stored;
```

is the set of tuples $\langle p_i, name_i \rangle$. In the general case, for a function

$$f\,(a_0, a_1, \ldots, a_m) \rightarrow \langle r_0, r_1, \ldots, r_n \rangle$$

the extent is defined as

$$extent\, f = \{|\langle a_0, a_1 \ldots, a_m, r_0 r_1, \ldots, r_n \rangle|\}$$

A derived function is defined in terms of an AmosQL query, and a foreign function is written in some other programming language such as C, Lisp or Java. Stored and derived functions are invertible, and Amos II will compute the inverse when necessary.

Here is an example of a typical AmosQL query:

```
select name(e) from employee e where hobby(e)='sailing';
```

Here `name` and `hobby` are stored functions where each can emit several results, and in order to answer the query, Amos II must be able to run the inverse of `hobby` (as a function from the string `"sailing"` to an object of type `employee`).

3.1.4 Function Notation

Throughout this report, function applications will - for sake of readability - be notated without parentheses in the general case. A function is normally applied to *one object*, if several arguments are desired, they will come in a tuple. For example the function f applied to a single argument x will be notated

$$y = f\,x,$$

while a function g taking arguments x and y, will be notated as

$$z = g\ (x, y)$$

It is also common in Amos that functions have no arguments, only return values. In these cases the reader will always be notified that we are dealing with the function *application*, not the function itself. If nothing is said, however, f will mean the actual function f and not the value returned by f when applied to no arguments, unless otherwise stated.

3.1.5 Adornments

Functions in Amos II can be overloaded on the argument types and size of argument tuple, i.e. one can define the following functions

- `foo(integer x)->integer y`
- `foo(real x)->integer y`
- `foo(integer x, integer y)->integer z`

without redefining `foo` each time. This is because Amos II views the different varieties of `foo` as separate functions. They may all be called using the name foo, but the type-resolver (see 3.2.2) will give these functions separate names by using type *adornments*. Hence the three `foo`:s will be called $\texttt{foo}_{integer \rightarrow integer}$, $\texttt{foo}_{real \rightarrow integer}$, and $\texttt{foo}_{integer,integer \rightarrow integer}$, respectively. The different versions of the function are referred to as *resolvents*. The query processor will decide which resolvent to call.

Because of the invertibilty of functions - i.e. running a function backwards - is normal behavior in query optimization and execution, there needs to be a way to specify not only which type adornment to use. Also, each function application is in addition to type-adornment also *binding adorned*, for example a function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ in the forward direction will be binding-adorned f^{bf}. The letters *bf* is called a *binding pattern*, and the superscripting to a function is called a binding adornment. The meaning of the letters is

1. *b* means that the argument is a bound variable or a constant, e.g. 12 or "a".

2. *f* means that the argument is either a free variable, or a bound variable or constant, as above. Running an adorned function with a bound variable v in the place of an *f* argument results in running the function with v replaced with a different, uninstantiated variable to determine v, and then comparing the result.

Because *f*, for free, means also that the variable can be bound, an *f* adornment is said to *cover* a *b* adornment. For example the binding pattern *bbfbffb* will cover *bbbbffb* but not *fbfbffb*. All binding patterns are not necessarily executable for any function *f*. For stored functions, however, any adornment is possible since for a stored function the extent is simply a set of stored tuples (see 3.1.3). A foreign function can be defined for several binding patterns, with different implementations for each binding pattern. It thus makes sense to talk about type- and binding adorned functions. These are referred to as TBRs (see 3.2.5), and are the actual functions.

3.2 Query Processing

An AmosQL query is defined by a `select` statement. Amos does not separate derived functions and queries, and thus a `select` statement is treated as an anonymous function. The steps in the compilation of a query are shown below. As an example, suppose we have the following types and functions defined,

```
create type employee
    properties(integer salary, name charstring);
create function manager(employee) -> employee as stored;
```

and the following query is posed; finding the name of all employees that earn more money than their bosses:

query 1
```
select name(e)
from employee e
where salary(e) > salary(manager(e)).
```

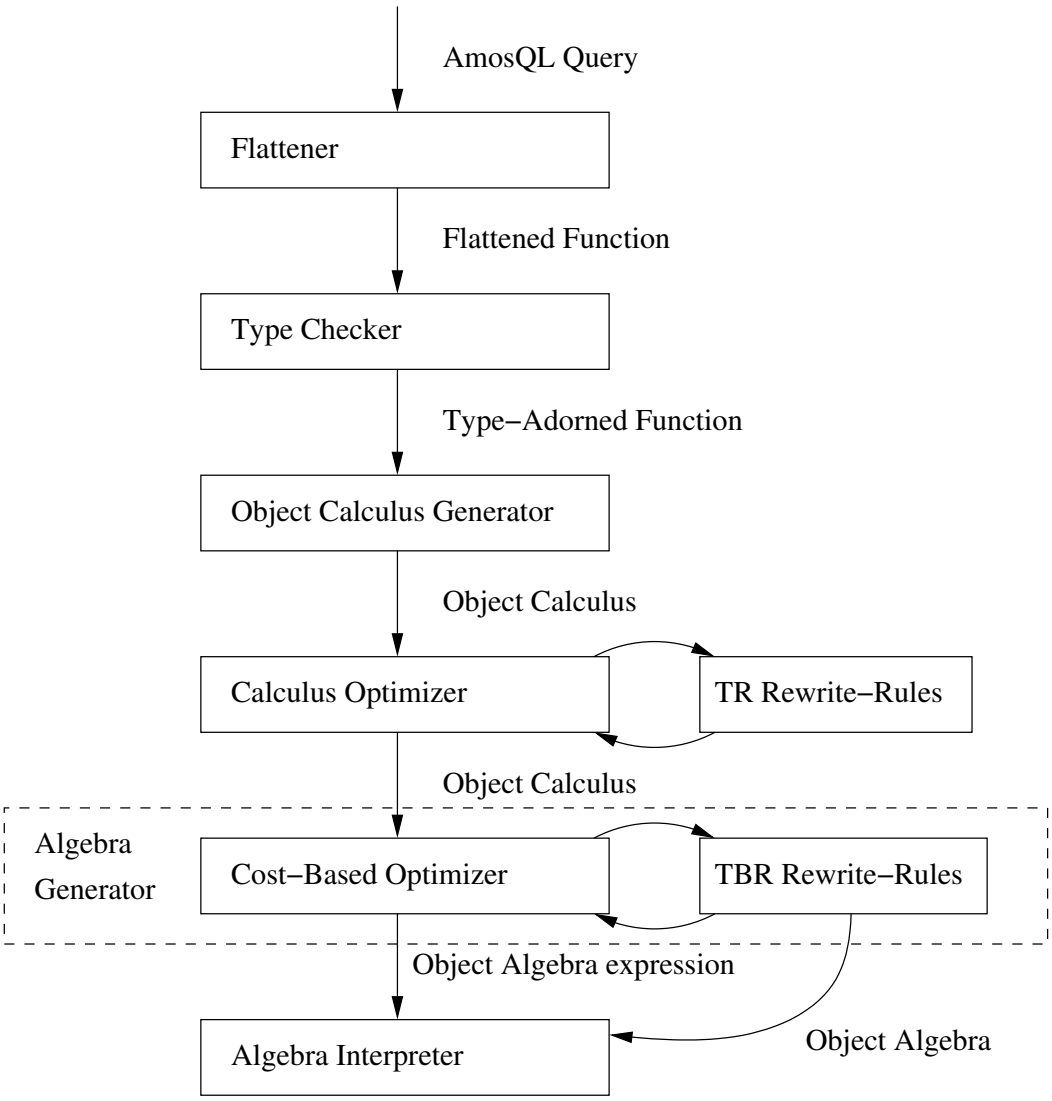


Figure 1: The query processing chain in Amos II.

3.2.1 Flattening

Because the query plan interpreter does not allow nesting of function calls, they are removed from the function definition by introducing intermediate variables. The result of flattening applied to query 1 will result in the following flattened function definition[1].

```
create function *select*() -> charstring v1 as
    select v1
    from employee v2, employee e
    where v1 = name(e) and
        v2 = manager(e) and
        v3 = salary(e) and
        v4 = salary(v2) and
        v3 >v4;
```

3.2.2 Type Resolution

Functions in Amos II can be overloaded on the argument types, and therefore the query processor must decide which resolvent to call. If this can be inferred from the bound variables to a function, this is performed at compile-time. Type resolution applied to the previous step will yield

```
create function *select*() -> charstring v1 as
    select v1
    from employee v2, employee e
    where v1 = name_employee→charstring(e) and
       v2 = manager_employee→employee(e) and
       v3 = salary_employee→integer(e) and
       v4 = salary_employee→integer(v2) and
       v3 >integer,integer→boolean v4;
```

A type-resolved function is also referred to as *type adorned*, or a TA-function.

3.2.3 Object Calculus Generator

After flattening and compile-time type-resolution the function is transformed into an internal *object calculus* representation. This corresponds directly the TBR (type- and binding-resolved) Object Log used in 3.2.5. Litwin and Risch[1] use TR (type-resolved) Object Log for the representation of this step, while the Object Calculus model is used by Fahl and Risch[2]. In this paper, object calculus is used for the TR phase and TBR Object Log is used for the final execution plan.

[1] Amos II gives an anonymous query the function name `*select*` and treats it as an ordinary function, the only difference being that it is executed directly after compilation.

$$\{v_1 \mid v_1 = name_{employee \to charstring}(E) \wedge$$
$$v_2 = manager_{employee \to employee}(E) \wedge$$
$$v_3 = salary_{employee \to integer}(E) \wedge$$
$$v_4 = salary_{employee \to integer}(v_2) \wedge$$
$$v_3 >_{integer,integer \to boolean} v_4\}$$

Note that no order between function calls is defined in this step.

3.2.4 Calculus Transformations

Once a function appears as an object calculus program, various transformations are made to optimize the function. These are the most important ones

- All equalities are removed and one of the values substituted for the other.
- The *-transformation (see 4.3) was introduced at this stage.
- Normalization. The cost-based optimizer accepts only object algebra expressions in disjunctive normal form. This restriction may be lifted in future versions.

In our example no such rule applies.

3.2.5 Object Algebra Generator

The cost-based optimizer transforms the declarative object calculus expression into *object algebra*, where stored functions become facts and derived (and foreign) functions become predicates. For example, if the salary function is defined for $salary(e) = 2000$ for some e of type `employee`, the function is stored as the type-resolved fact $\texttt{salary}_{employee,integer}\texttt{(e, 2000)}$.. A derived function $f\ a = r$ becomes a predicate `f(a, r) :- ....`

In object algebra, as opposed to object calculus, a strict order is imposed on the ordering of predicates, as dictated by the cost-based optimizer. Furthermore, each function call is binding adorned. The product of such a rule is a TBR predicate. For example, a function $f : \mathbb{Z} \to \mathbb{Z}$ will correspond to the predicate $\texttt{f(v}_1\texttt{, v}_2\texttt{)}$, and will be type- and binding-adorned $\texttt{f}^{bf}_{integer,integer}\texttt{(v}_1\texttt{, v}_2\texttt{)}$. When applied to query 1 the TBR Object Log is

$$\texttt{*select*(v}_1\texttt{) :-}$$
$$\texttt{name}^{ff}_{employee,charstring}\texttt{(e, v}_1\texttt{) \&}$$
$$\texttt{manager}^{bf}_{employee,employee}\texttt{(e, v}_2\texttt{) \&}$$
$$\texttt{salary}^{bf}_{employee,integer}\texttt{(e, v}_3\texttt{) \&}$$
$$\texttt{salary}^{bf}_{employee,integer}\texttt{(v}_2\texttt{, v}_4\texttt{) \&}$$
$$>^{bbf}_{integer,integer,boolean}\texttt{(v}_3\texttt{, v}_4\texttt{).}$$

which is a procedural program where an order between predicates is defined. The query optimizer tries to find the optimal ordering between function calls when transforming TR Object Log into TBR Object Log.

3.2.6 Rewrite Rules

The TR⇒TBR stage contains a hook for declaring a rewrite-rule for a function. In this case the system will not do the default transformation from a TR resolvent to a TBR resolvent, as dictated by the cost-based optimizer (see section 3.2.5.) Instead, the specified rule will be called and its result used. Rewrite rules are written in a general-purpose language which in practice enables a rewriter to do anything, and this is indeed where a translator to another data source is introduced.

Due to normalization, the scope of a rewrite is limited to only one disjunct at a time, the disjunct being, of course, a conjunction. Optimization and rewriting are carried out in terms of these conjunctions, with each conjunction handled separately. A transformation applied to one conjunction will not affect any other conjunction and a translator will not be able to see anything outside the conjunction that it is working on. Furthermore, a TR-predicate is purely declarative. No restrictions on bindings are imposed. Hence, a function body that is subject to translation can be viewed as a set of predicates with only the name of the function and its arguments. Some of these can be translated, under certain binding adornments.

4 Extensible Wrappers

This work focuses on wrapping of external data sources within Amos II, in order to get complete and transparent access to these sources through AmosQL. The goal is to access any kind of data source with any kind of structuring, or lack thereof. Even if the data has only trivial structuring, e.g. like a file system, it should still be possible to query the data source. This means that the wrapper must take as input a parsed query in AmosQL, translate this into low-level calls to the source and then post-process the result in order to make up for any limitation on the source's behalf. The greatest challenge is to know in advance which parts of the query can be translated into an external call and which parts must be done inside Amos, and how to represent this knowledge. The problem was divided into the following subproblems

1. Connecting external data sources to AmosQL. This is discussed in section 4.1.

2. The ability to query these sources. The naive - and most general - approach will be discussed in section 4.2.

3. The automatic translation of AmosQL queries into whatever query language the source supports. Section 4.4 discusses this.

4. The translation should be optimized to minimize total query execution cost. While this work implements some simple assumptions, this issue remains to be settled in the general case. See section 7 for a discussion of this.

The approach chosen was to first create a naive wrapper that assumes that the source has no capabilities at all, and thus do all processing in Amos. A data source is thus regarded as little more than a file system, with a function to open and close a file, and nothing more. As we move along we will hopefully discover that the data source is capable of doing much more. This requires a mechanism called a *translator* which will, given a knowledge representation of capabilities, translate a naive query into a more efficient one that takes advantage of the source's capabilities. Thus a wrapper can be split up into two largely independent parts; an *interface* containing only the basic building blocks for importing data to Amos and a *translator* to translate such a query into something more cost-effective than importing the data into Amos and do all the processing there.

4.1 Basic Wrapper Interface

4.1.1 Mapped Types

Since Amos II is an object-oriented system, every entity in the system is represented by an object. Therefore external data must also somehow be represented by objects. Those objects that represent external data are simply empty placeholders without attributes - they contain no data and their functions are *foreign* (see 4.1.3). Such objects are called *proxy objects*, and they are defined by *mapped types*.

Proxy Object Lightweight objects that are simply containers for objects that are stored in a different physical location.

Mapped Type A set consisting of proxy objects that classify such objects by its *extent function*, which returns all proxy objects of the type.

Obviously, it is up to the author of a wrapper to decide to which entity in the data source language an object should be mapped, and this mapping will reflect how the system will treat the data source.

The problem that inevitably arises concerns identity of entities: Objects are explicitly created and can always and uniquely be identified by their object identifiers, or OIDs. Foreign entities may not exhibit this trait, and while it is not completely crucial that there exist a one-to-one mapping between objects and the entities that they represent, an OID must always correspond to only one foreign entity. How this mapping is to be carried out depends completely on the data source. So every mapped type needs a function from objects to foreign entities that is also reversible. In Amos II this function is called `decode`.

4.1.2 The decode function

For every kind of data source there must be some way of identifying tuples, be it a primary key, or a directory path etc, that is unique for the entity. The purpose of the `decode` function is to produce a handle to a foreign entity given an OID. The function is overloaded for each new mapped type that is created.

When a mapped type M is created that maps to a source entity E the system will automatically create the following functions:

1. $decode_{M \to E}$
2. $decode^{-1}_{E \to M}$

For the forward direction (1) the function result is simply stored on the mapped object itself, as a property of some sort. It is supplied when instantiating a mapped type and thus *decode* will operate on any instance of the mapped type. The backward direction (2) is slightly more complicated because we have to create a new object that is a proxy to the source entity. Here it is advantageous to keep the mapped objects as lightweight as possible.

4.1.3 Foreign Functions

Amos II lets a user define functions in other languages than AmosQL, namely C, Lisp, and Java. A function defined in any of these languages is referred to as a *foreign function.* The function is declared as "foreign" and the definition contains a reference to the foreign implementation. A foreign function lets the implementor access any kind of data structure directly through AmosQL and it will pass unaltered through optimization. Amos II normally opens up functions when optimizing a query or a function. but a foreign function is somewhat mysterious to the optimizer. It knows nothing of what will come out as a result, but with a clever use of cost information, the cardinality of the output and the cost can be used to incorporate foreign functions in an optimal plan. Thus a foreign function is not a "black box" to the optimizer, but rather a "gray box".

4.2 The Core-Cluster

The naive wrapper assumes that a data source has no structured facilities for retrieving data. It is merely treated as a file system, where an entire file is scanned through as it is being processed. This means that there is only one primitive needed for each mapped type, namely the extent of the type. In Amos II the extent of a type is the set (see 3.1.2) of all existing objects of the type. By using this definition of an extent function, however, we would be forced to create objects for every entity that is in the extent, even when those objects will not be used. Suppose we have a wrapper for some database of engine parts where you can browse parts by some trait called `part_id` that uniquely identifies a part. Now consider the query

```
select p from engine_part p
where begins_with(part_id(p), 'A');
```

Intuitively, this will create mapped objects to represent those engine parts whose `part_id`:s begin with the character 'A', but *only* for those parts. No other objects will be created. Now, if the wrapper uses the extent function described

above, the meaning of the above statement becomes "create proxy objects for all the engine parts, throw away all of them except the ones whose id:s begin with 'A' and project the rest". This would lead to the creation of a potentially very large amount of unnecessary objects. There is a need for an extent function that works on a lower level, and leaving object creation for a later stage, when more is known about the query. So the minimal building block chosen is known as the *core-cluster* function. It returns rather than the objects themselves a bag of tuples which are the values of their stored attributes in some pre-specified order. For example, consider the engine parts above. Suppose that an engine part has two properties: part id and price. Then we would expect two functions to be present

- $part\_id_{engine\_part \rightarrow integer}$
- $price_{engine\_part \rightarrow real}$

We then define the core-cluster of engine-parts as

$$cc_{engine\_part} = \{|\langle part\_id, price \rangle|\}$$

Strictly speaking the core-cluster is not a function since it has no arguments. However, its value changes over time as the database is updated. So in the mathematical sense it is merely a function over time. Therefore, in practice the core-cluster needs to be recomputed every time it is needed in order to be up to date. The order of elements in the tuples is totally arbitrary, but it is important that one order be agreed on, as we shall see later in this section.

The extent function can always be constructed from the core-cluster function in the following manner

$$extent\, p = decode_{E \rightarrow M}\ cc_{engine\_part}$$

where cc_{engine_part} is the core-cluster itself, even though in practice it is a function, applied to no arguments and returning a bag of tuples. We then have the possibility of doing filtering before object creation like so (execution order is right-to-left):

$$extent\, p = decode_{E \rightarrow M}\ filter_{begins\_with\ (partid, 'A')}\ cc_{engine\_part}$$

Here it is clear that object creation comes in after filtering of tuples. Actually, the decode function will not need the entire tuple in order to create the object, only enough information to distinguish a particular tuple. Thus, a subset of the attributes is designated 'key' and is expected to always be unique. This is why it is important to know in advance which positions in the tuple that correspond to which core properties. It is up to the wrapper implementor to make sure that this holds for the data being wrapped.

4.3 The *-Transformation

One problem that arises in the rewriting of queries involving the combination of core-cluster and decode calls is that, more often than not, the call to the decode function is unnecessary [2]. Those cases are when the initial AmosQL query looks like:

```
select f(e) from wrapped_datasource_entity e where ...;
```

Here what is asked for is simply a field from all objects, with or without a filtering `where` clause. There is no need to introduce the creation of mapped objects. Let's see what this compiles to in the TR stage:

$$\{v_0 \mid$$
$$cc_{wrapped\_datasource\_entity}\ (v_1,\ v_2,\ldots,v_f,\ldots,v_n) \wedge$$
$$decode_{E\rightarrow M}(v_f, e) \wedge$$
$$v_0 = v_f\}$$

As we can see, there is no need to call *decode* here since the result is immediately thrown away. The predicate is redundant (always true and introduces no new tuples.) This is where the *-transformation is used. In Amos II, a variable named * can never be bound and will succeed in any comparison, much like the _ variable in Prolog.

The transformation performs a fix-point iteration consisting of two steps:

1. Identify variables which are never used and replace by a *,
2. remove the predicates which are redundant if one argument is a star, and
3. repeat until no more unused variables are encountered.

A variable is unused if it is a singleton variable that is not an argument to the predicate (neither an input- nor an output variable). A predicate is redundant if it does not produce any new tuples and never fails. It is harder to infer redundancy of predicates, so therefore there is simply a "black list" of such predicates which are redundant under certain conditions, and *decode* can definitely be on that list, since $decode_{E\rightarrow M}\,(*, e)$ will create a mapped object without a mapping, $decode_{E\rightarrow M}\,(v, *)$will create an object and throw away the handle, $decode_{M\rightarrow E}\,(e, *)$will extract the mapping but throw it away immediately, and $decode_{M\rightarrow E}\,(*, v)$will not be executable. The same goes for any direction if both arguments are $*$.

So a *-transformation applied to the above expression will leave the following calculus expression:

$$\{v_0 | cc_{wrapped\_datasource\_entity}(*, *, \ldots, v_f, \ldots, *) \wedge v_0 = v_f\}$$

4.4 Modeling Capabilities

This section discusses the various challenges associated with the representation of capabilities. An intuitive definition of capabilities would be "what can data source A do for me?". Filtering and comparisons are obviously capabilities, for example the question "retrieve all employees for me but only if they make less than 1500 dollars monthly" appeals to a capability to execute comparisons. Capabilities go on to be more complex than this, for example inner and outer joins, cross-products and solving parametric equations in 10 dimensions.

A data source is associated with a set of capabilities, and capabilities are associated to rewriters for this particular TBR function. A rewriter for the combination of a particular TBR function and a particular data source will throughout this text be referred to as an *absorbent*. These will be further explained in 4.4.4, but for now we will only note that they are a variant of rewrite rules.

Commonly, one associates capabilities not with individual data sources but with whole classes of data sources, e.g. with the type rather than the instance. We may for example have a data source $datasource_1$ that is similar to Amos II in the sense that you can write foreign functions, and for a particular running version of $datasource_1$there may be an instance ds, which has an efficient implementation of the function *foo*. Now, the set of capabilities associated with $datasource_1$ are all the features that Amos II and $datasource_1$ have in common, while the capability `foo@ds` can be associated only with ds. However, there is never a need for a capability to be associated with both, since the semantics of associating a capability with $datasource_1$ dictates that it also be associated with ds through type membership, much like `static` members in Java.

4.4.1 Representation of Capabilities

A capability can be viewed as a triple $\langle s, f, \alpha \rangle$ where s is either a subtype to `datasource` or an instance of such a type, that is $s \in deepextent(\texttt{datasource}) \vee subtype(s, \texttt{datasource})$, f is a TBR function, and α is an absorbent. If the capability is hooked to the type it is subsumed to belong to all instances of the type. Armed with a set of capabilities, a data source can, using its associated absorbents, rewrite a subset of a query.

4.4.2 The Variable Environment

The translator system includes an *environment*, which is a mapping from literals (variable symbols or constants) to a variable descriptor, which is a 4-tuple

$$\varepsilon\, s = \langle t, b, e, d \rangle$$

1. t is the type of the symbol. If s is a constant then this is the type of the constant, otherwise it is the type of the variable.

2. b is the current binding of the symbol, so that

$$b = \begin{cases} - & \text{if } s \text{ is bound or a constant.} \\ + & \text{if } s \text{ is free.} \\ 0 & \text{if we are not permitted to specify a value.} \end{cases}$$

3. e is the foreign entity that the variable maps to. For relational databases an entity would be a column, and s could take on any value that is stored in the column. It is up to the absorbent to determine this. If s is a constant the entity is s itself. For example, the assignment

$$\texttt{s} \leftarrow \texttt{ssn}_{person \rightarrow integer}(\texttt{p})$$

leaves `s`'s entity being the column `ssn` in the table `person` if we are wrapping a relational database, because `s` gets its value from this column. Then the translator knows what to translate any occurrence of s into, namely a projection of `person.ssn`. In the same scenario,

$$\texttt{s} \leftarrow 1000$$

means that `s` maps to the value 1000 and nothing else, regardless of data source.

4. d is the data source that the variable maps to.

4.4.3 Accumulators

The translation framework does not translate a query in one go, the motivation of which is flexibility. There is not one single translator associated with a data source but rather a set of capabilities, each with its own specialized translator. The motivation here is that it should be easy to add new capabilities. Capabilities are likely to be discovered and implemented as you go along, often as optimizations.

A difficult situation arises if some function f_1 with argument tuple τ_1 is rewritten by an absorbent $\alpha_1$into a call to the wrapper interface, say $f_1'\tau_1'$, and further down some other function $f_2\tau_2$ is rewritten by a different absorbent α_2 into another interface call $f_2'\tau_2'$. In this case a lot may have been won as far as speed is concerned, but for n rewriteable function calls we still end up with n calls through the wrapper interface. It is also necessary to somehow combine several calls into a single one, if the data source allows it. This can be potentially expensive, however. Consider for example if the communication with the data source is carried out with strings, as is the case with relational databases. Now, consider the object calculus expression

$$\begin{aligned} &\{name \mid \\ &\quad Person(p) \wedge \\ &\quad sal = salary_{employee \rightarrow integer}(m) \wedge \\ &\quad name = firstname_{person \rightarrow string}(p) \wedge \\ &\quad sal >_{integer,integer \rightarrow boolean} 1200 \wedge \\ &\quad sal <_{integer,integer \rightarrow boolean} 1800\} \end{aligned}$$

The above approach involves parsing and generating strings in every step in the translation chain, each step 'eating' one function call:

1. `select name from person;`
2. `select name from person where salary>1200;`
3. `select name from person where salary>1200 and salary<1800;`

A faster approach, which is now used in the implementation, is to introduce an abstract representation of the above `select` statement. Such a representation needs to have facilities for absorbing any new function call that the data source has been found capable of, and it needs a way to translate it into a finalized call to the interface. Such an abstract representation will be referred to as an *accumulator*. The wrapper API expects a data source to have two functions for accumulators,

1. $initialize : datasource \rightarrow absorbent$
2. $finalize : absorbent \rightarrow call \cup \perp$

where the function *initialize* will create an empty accumulator, and the function *finalize* will transform any accumulator into an interface call that can later be executed. $\perp$ is a symbol used to denote failure (see 4.4.6.) Because of the diversity of data sources nothing more can be generalized out of the notion of an accumulator.

4.4.4 Absorbents

An absorbent is a function that translates a TBR function call $f\,\tau$, an environment ε and an accumulator a. The result of this translation is also an accumulator, a' where hopefully $f\,\tau$ has been absorbed. More formally, if F is the set of functions that can be translated, E is the set of environments where this is possible, and A is the set of possible accumulators, then an absorbent α has the following signature ($\perp$ is the failure symbol, see 4.4.6)

$$\alpha : F \times E \times A \rightarrow A \times E \cup \{\perp\},$$

for example, for the f, τ, ε, and a above, the result of applying a would be

$$\alpha\ (f, \varepsilon, a) = \langle a', \varepsilon' \rangle\,,$$

where a' is the accumulator a with some representation of f included and ε' is an updated environment. It is necessary to let an absorbent update the environment also. Consider, for example, an assignment operator $\leftarrow^{bf}$. Any remotely intuitive interpretation of the semantics of this is that when applied as $b \leftarrow a$, it should result in b being bound afterwards. It may be possible to absorb this into a translation in the making, for example by treating any occurrence of

b as an occurrence of a. Either way, b must be bound hereafter, or the correct capabilities will not be found.

If the absorbent does not return $\perp$, it has by definition succeeded, and thus the function call for which it was invoked will be removed from the conjunction.

There is likely to be an absorbent associated with each capability, although it may prove more practical to have the same function translate everything. The important part is that adding new capabilities to a source should not mean redefining previous work.

4.4.5 The Default Absorbent

There is one absorbent that has to always be specified, and it is the one for the core-cluster function that the translator is hooked on. This absorbent is different from the others in three ways:

1. It is not necessarily part of a capability,
2. it will normally only be called once, the only exception being a self-join. Any other absorbent will be called as many times as the function(s) for which it is associated appears in a given conjunction.
3. Whereas no order is guaranteed for when other absorbents are invoked, this one will always be invoked before any other of them.

For some data sources it makes sense to absorb more that one core-cluster function call, the join capability. The source may even be capable of translating any number of core-cluster calls that belong to the same data source, unlimited join. There are only two options. Let `mt` be a mapped type, let $\texttt{cc}_{\texttt{mt}}$ be its core-cluster function, and let $\alpha_{\texttt{cc}_{\texttt{mt}}}$ be the absorbent that absorbs a call to $\texttt{cc}_{\texttt{mt}}$.

1. Register $\alpha_{\texttt{cc}_{\texttt{mt}}}$ as the translator's default absorbent for the mt.
2. As above, but also add the capability $\langle \texttt{ds}, \texttt{cc}_{\texttt{mt}}, \alpha_{\texttt{cc}_{mt}} \rangle$ to the set of capabilities.

In the second case, the core-cluster function itself makes up a capability, and the default absorbent is submitted just as any other absorbent. If this is done then the first and second characteristics above do not apply. The only join capability that can be explicitly expressed is the unlimited join. All others must be implicitly stressed by failing when the full number has been reached. (section 4.4.6).

4.4.6 Failures

An important requirement on the wrapper framework is that is should be able to handle failures. The reasoning behind this is simple: The task of representing every capability for any source in a uniform manner is infeasible. A source may

be capable of translating a large set of function calls but not certain combinations, or it may *only* translate certain combinations. For instance, B-trees (see section 5) accept only closed ranges.

Capabilities in the wrapper framework are to interpreted as "necessary but not sufficient" conditions for being able to translate a set of function calls. There framework never assumes that a translation will succeed until it does. In the mean time, it can fail in two ways.

1. Failing to absorb. An absorbent function may, as noted above, return $\perp$. If it does, then the adorned function call in question will not be removed from the conjunction and generally the effect is the same as if there were no capability for this particular adornment. However, if the same function call appears later, under a different adornment, the absorbent will be called in the new context to see if it works.

2. The optimizer may try different orderings in order to get the optimal absorption. In many cases this means to absorb as many predicates as possible, but this does not always hold as costs may go up for advanced source queries. Such considerations are beyond the scope of this paper, however, and are discussed in 7.

3. Failing to finalize. The finalize function for the translator may fail, in which case all translation is being rolled back. The query optimization algorithm might try again by reordering the function calls, but depending on the algorithm it may not explore the full solution space. In the general case it is not possible to predict which orderings will succeed and which will not.

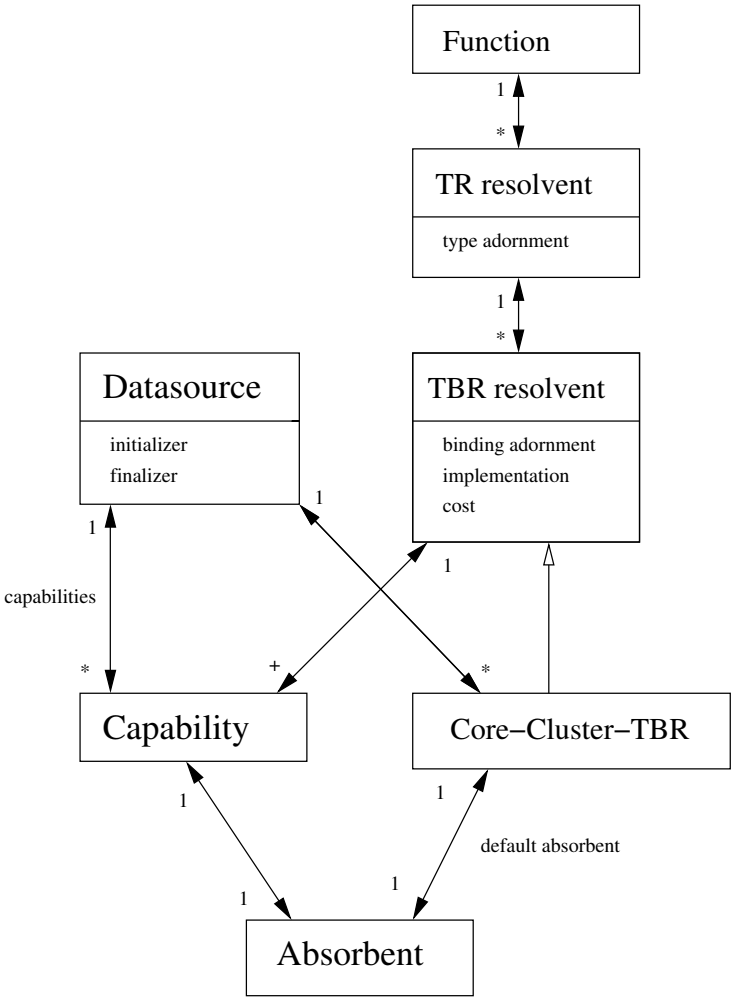
5 B-tree Wrapper

In this section it is shown how the built-in indexing structure, b-tree, is implemented as a wrapper in Amos II, rather than being hard-coded into the database kernel.

5.1 Amos II Indexing

Like most DBMS:es, Amos II has an indexing facility. Indices can be created for any argument or result of a stored function, and can be *unique* or *non-unique*. A unique index prohibits storing different tuples where the indexed argument appears. Indeed, this is how cardinality constraints are implemented in Amos II. For a type `person` an intuitive unique constraint is the property or function $\texttt{ssn}_{person \rightarrow integer}$ (assuming every person has a unique social-security number). The unique index is declared using the keyword `key`:

```
create type person;
create function ssn(person key) -> integer key
```



Legend:

1 has one

***** has one, many, or none

$+$ has one or many

white arrowhead is a/is an

text on arrow Indicates relation. An absorbent is related to a TBR resolvent through a capability, and to core-cluster TBR by being its default absorbent.

Figure 2: The relationships between components of the wrapper definition framework.

In this case the system guarantees that there is a one-to-one mapping between OIDs of type `person` and their social-security numbers. This concept may be a little difficult to grasp if looking at the schema from a relational viewpoint.(Why should both be declared `key`?) The mystery arises from the fact that whereas a relational table has to have at least one column to make sense,, an Amos II type does not, because type instances can always be identified with the OID. The closest relational counterpart to the above declaration would be

```
create table person(oid int);
create table ssn(person_oid int unique, ssn int unique);
```

An equivalent, and intuitive declaration of `person` would be

```
create type person properties(ssn integer key);
```

It is also possible to add indices at a later point using the system function `create_index`. The signature is as follows:

```
create_index(function f, charstring argname, charstring index_type,
   charstring uniqueness)
```

parameters are

`f` the function

`argname` the name of the argument/result parameter to be indexed.

`index_type` Kind of index to put on the parameter. Amos II supports hash (type "`hash`") and ordered B-tree indices (type "`mbtree`").

`uniqueness` Can be "`unique`" or "`multiple`".

5.2 Interface

An internal foreign AmosQL function implements B-tree search:

```
mbt-select-range(function f, integer pos, object low, object
   high)
```

This foreign function accesses (returns) the rows (tuples) of a B-tree index associated with position `pos` of the stored AmosQL function `f` in the interval [`low, high`]. It does not handle open intervals.

5.3 Translator

5.3.1 Access Function Capabilities

The access function handles only closed ranges, i.e. a combination of two particular predicates, and only when they refer to the same indexed argument. There is no way to explicitly express this capability with the capability system. Instead, we declare capabilities for the general comparison operators:

- $\langle \texttt{mbtree}, >_{object,object \rightarrow boolean}, absorb_{>@\texttt{mbtree}} \rangle$
- $\langle \texttt{mbtree}, <_{object,object \rightarrow boolean}, absorb_{<@\texttt{mbtree}} \rangle$
- $\langle \texttt{mbtree}, \geq_{object,object \rightarrow boolean}, absorb_{\geq@\texttt{mbtree}} \rangle$
- $\langle \texttt{mbtree}, \leq_{object,object \rightarrow boolean}, absorb_{\leq@\texttt{mbtree}} \rangle$,

where `mbtree` is the type that represents the main-memory B-tree data source, and the functions *absorb* are the absorbents. Notice that equality(=) is managed by the binding pattern mechanism and need not be implemented as a capability mechanism.

5.3.2 Accumulator

In the translation of a range query, an accumulator will need to keep track of the applicable index and the desired values of the indexed parameters. It is, of course, also necessary to remember the function for which the query applies. So a data structure is to be defined (no particular programming language intended):

```
structure rangequery {
    function f;
    variable indexed_variable;
    function extent_function; // the actual table
    predicate low;
    predicate high;
}
```

For a query

```
select p from person p where ssn(p)>100000 and ssn(p)<=200000;
```

we would have the structure

$\{ssn_{person \rightarrow integer}, v, p\_ssn_{person \rightarrow integer}, index_{person.ssn}, v > 100000, v \leq 20000\}$

when all predicates have been absorbed.

Finally, the finalizer is called, which asserts that both predicates are present, i.e. that the range is closed. the finalizer then produces

1. A call to `mbt-select-range`, which produces a vector of the columns in the storage manager's table.

2. Two or more extraction operators from the vector. (in the case of a one-argument-one-result function we get only two.)

3. Extra predicates to close the interval if open in either end.

Furthermore, the range predicates are removed from the conjunction as they are no longer needed. Thus the final calculus expression becomes (type adornments dropped for clarity:)

$$\{v_1 = mbt - select - range\,(ssn_{person \to integer}, 1, 10000, 20000) \wedge$$
$$p = v_1[0] \wedge$$
$$v_2 = v_1[1] \wedge$$
$$v_2 \neq 10000\}$$

6 The JDBC Wrapper

The purpose of this part of the project was to define a wrapper for JDBC for Amos II. JDBC was chosen because it is at one extreme end when it comes to expressibility of the source language, namely SQL. In fact many queries, when successfully translated will look nearly identical to the OSQL originals that they once spawned out of. The wrapper is very large, about 1000 lines of code, owing to the not entirely trivial task of generating SQL strings.

6.1 Previously Implemented (ODBC) Wrapper

There existed previously for Amos a wrapper for relational data that works through the ODBC interface. It is not based on mapped types but rather on an elaborate mechanism that is heavily integrated with the Amos II kernel. The biggest difference, however, is that the translator must succeed in translating *any* query to SQL. While the old wrapper for ODBC at the user level bears a lot of resemblance to the JDBC wrapper, whereas this wrapper is heavily integrated with the Amos II kernel, the JDBC wrapper uses no system functions at all. However, the ODBC wrapper's primitive interface has the same functionality as that of the JDBC wrapper.

6.2 Interface

One new AmosQL type name `jdbc` was created as a subtype of `relational`, the supertype of all relational data sources.

`jdbc(charstring name, charstring driver)`
the creator for `jdbc` objects.

`load_driver(jdbc ds, charstring driver)`
attempts to load the JDBC driver for the data source.

`connect(jdbc ds, charstring url,`
`charstring name, charstring password)`
connects with a given user name and password.

`disconnect(jdbc ds)`
closes the connection to a database.

`sql(jdbc ds, charstring query)`
direct SQL execution.

`sql(jdbc ds, charstring query, vector arguments)`
parameterized SQL execution, using prepared statements to cache the query for reuse.

`tables(jdbc ds)`
returns a bag of all table names in the database.

`columns(jdbc ds, charstring tablename)`
returns a bag of all column names in the database.

`primary_keys(jdbc ds, charstring tablename)`
returns a bag of all names of columns that make up the primary key of a table.

6.3 Translator

6.3.1 Observations

1. A call to the core-cluster function represents one table access, always. If the same core-cluster function appears more than once, what is asked for is a self-join, and thus the second time the table name must be aliased. If different core-cluster calls appear then either a join or a straight union is asked for. Hence, as soon as a core-cluster function is created, the data source must be declared capable of translating them.

2. Because of the DNF form of the object calculus, there is never the need to translate into `ors`, because we only see one conjunction at a time.

3. The *entity* that variables map to is in this case a column.

^{select}\{employee$_1$.emp_no, department$_1$.emp_no\}
^{from}employee$_1$, department$_1$
^{where}\{employee$_1$.emp_no=department$_1$.emp_no\}
$^{table-incarnations}$\{employee$_1$, department$_1$\}
^{input}\{v_0\}

Figure 3: An abstract select structure

- The `select` part is simply a set of column names, for sake of removing ambiguities prepended with table names (i.e. for self-joins). In SQL it is always legal to use the dot notation with table names when referring to columns, so there is no need to remove them.
- The `from` part is a set of table names.
- The `where` part is simply a set of predicates in Object Log, by contract with the Amos optimizer assumed to be a conjunction of these predicates.
- `table-incarnations` holds a set of tables that have been asked for. If a table is asked for twice its *incarnation number* is increased and the table is added again, for example if table-incarnations is $\{\text{employee}_1, department_1\}$ then the tables employee and department have been asked for one time each. If for example, employee is asked for again - by discovering yet another call to employee@ds_cc - its incarnation number is increased by one and `table-incarnations` becomes $\{employee_1, department_1, employee_2\}$.

6.3.2 Accumulator

The translator uses as accumulator a data type called an `abstract-select`. It is a structure representing an abstract syntax representation of an SQL `select` statement, meant to hold all relevant information until the statement can finally be materialized as a string.

In the wrapper interface it is possible to prepare a query, which means that the query is somehow precompiled at the source but it is still possible to alter one or more parameters. This is of great help when doing joins over different data sources: We shall now as an example introduce the compilation and subsequent translation of a derived function called `manager`. Assume we have two SQL tables, employee and department in the following schema,

employee	emp_no	first_name	last_name	dept_no$_{\mapsto \text{department}}$
	int	char	char	int

department	dept_no	name	mngr_no$_{\mapsto \text{employee}}$
	int	char	int

where emp_no is the primary key of the employee table mapping to mngr_no in the table department, where it is a foreign key. dept_no is the primary key of the department table, and is referenced from dept_no in employee where it is a foreign key. We now import these tables into amos with the following function calls:

```
import_table(ds, "employee");
import_table(ds, "department");
```

import_table automatically adds the core-cluster absorbents to the capabilities of the type ds, so that the wrapper knows that it can translate several of them in one go, as explained in section 4.4.5.

In creating a mapped type, property functions are automatically generated for retrieval of all properties of the types employee and department:

- $\texttt{emp\_no}_{employee@ds \rightarrow integer}$
- $\texttt{first\_name}_{employee@ds \rightarrow charstring}$

$\vdots$

Such a function is implemented in the following way (the function $\texttt{first\_name}_{employee@ds \rightarrow charstring}$ is used as illustration:)

```
create function first_name(employee@ds e) -> charstring
as select name
from charstring name, integer emp_no,...
where employe@ds_cc= ⟨emp_no, name, ...⟩;
```

The user can now define a function manager in AmosQL:

```
create function manager(employee@ds e) -> employee@ds
as select manager
from employee@ds manager, department@ds dept
where dept_no(employee)=dept_no(dept) and
emp_no(manager)=mngr_no(dept);
```

This translates into object calculus, using the *-transformation and substitution of equalities.

$\{employee,\ manager \mid$
1 $\quad decode\,(employee, v_1) \wedge$
2 $\quad employee@ds\_cc\,(v_1, *, *, v_2) \wedge$
3 $\quad department@ds\_cc\,(v_2, *, v_3) \wedge$
4 $\quad employee@ds\_cc\,(v_3, *, *, v_2) \wedge$
5 $\quad decode\,(manager, v_2)\}$

The execution plan is not fixed at this point but for sake of simplicity we will assume that the calculus subexpressions will be dealt with in the order they are numbered above. This happens to be the optimal execution plan, but how this is inferred is beyond the scope of this thesis.

First, the default absorbent for `employee@ds_cc` will be called in this way: `jdbcTranslateCoreCluster`($employee@ds\_cc\,(v_1, *, *, v_2)$, ε, s), ε is the variable environment, and s is an empty abstract select statement. The absorbent will note that $v_1$is bound since the call to *decode* is already binding-adorned (remember that we assume that the order above is followed) and v_2 is free. So v_1 is bound and it is recorded that the variables' `entity`:s are `emp_no` and `dept_no`, respectively. Their `data source`:s are set to `ds`. So the mapping ε is now:

- $\varepsilon\, v_1 = \langle \texttt{integer}, -, \texttt{employee}_1\texttt{.emp\_no}, \texttt{ds} \rangle$
- $\varepsilon\, v_2 = \langle \texttt{integer}, -, \texttt{employee}_1\texttt{.dept\_no}, \texttt{ds} \rangle$

The absorbent notes that v_1 was free, but its `datasource` is not equal to `ds`. It came from the decode function and is meant to be an argument to the `sql` function. The translation continues, and this time `jdbcTranslateCoreCluster` is called with function call number 3. This time v_2 is bound and maps to a column already, so its "meaning" is `dept_no`, while v_3 is a free variable with no entity. Nothing happens to v_2 but for v_3 we get

- $\varepsilon\, v_3 = \langle \texttt{integer}, -, \texttt{department}_1\texttt{.mngr\_no}, \texttt{ds} \rangle$

And the table as well as column are added to the abstract select structure. Last to be translated is function call number 4: First of all we see that table employee was already asked for once so we increase the incarnation-counter by one. This time v_3 and v_2 are both bound with an entities so we add $\texttt{manager}_2\texttt{.emp\_no} = \texttt{department}_1\texttt{.mgrno}$ to the set of predicates in a, and we are done. The wrapper API calls a `finalize` function that spits out a call to the `sql` function, complete with a materialized SQL string.

6.3.3 Performance

Performance of the JDBC wrapper using a translator was measured against an experiment database where the size of the database was altered. Two kinds of queries were compared; queries against a single table and join queries. The purpose of the test was to measure the efficiency gain of translating queries vs. importing all data to Amos II, and do the processing there.

Experiments were run on an Intel Pentium III with 785 952 kB RAM. The interface part of the wrapper uses Interbase's proprietary JDBC driver, and both the database file and the database server were located on this machine. The following schema was used for the experimental database:

```
create table m_department(dept_no int not null primary key);
```

Algorithm 1 Pseudo code for the default absorbent for core-cluster calls in the JDBC wrapper

```
jdbcTranslateCoreCluster(f(v_1,...,v_n),ε, a)
  ds ← get_datasource(f);
  for i ← 1 to n do begin
    c ← get_column_name(v_i);
    if bound(v_i) then
      if datasource(v_i) = ds then
        add_predicate(a, c =entity(v_i));
      else
        add_predicate(a, c =v_i);
    ε(v_i) ← ⟨type_of(v_i), −, c, ds⟩;
  end;
  add_column(a,c);
  add_table(a,c);
```

```
create table m_employee(emp_no int not null primary key, dept
   int not null references m_department, first_name varchar(10),
   last_name varchar(10)));

create unique index i1 on m_employee (emp_no);

create unique index i2 on m_department (dept_no);
```

Two AmosQL queries were executed against the two tables, for range queries the following was used:

```
select emp_no(e)

from m_employee@ibds e

where emp_no(e)<i and emp_no(e)>=j;
```

where i and j were adjusted according to table size in order to give a selectivity of .01 (1% of the tuples). The results can be seen in 6.3.3. The second query was formulated as follows:

```
select emp_no(e)

from m_employee@ibds e, department@ibds d

where dept(e) = dept_no(d) and emp_no=500;
```

The result can be seen in section 6.3.3.

It is apparent from the results that translation and pushdown of selection predicates that take advantage of a data source's capabilities can give enormous gains in execution speed.

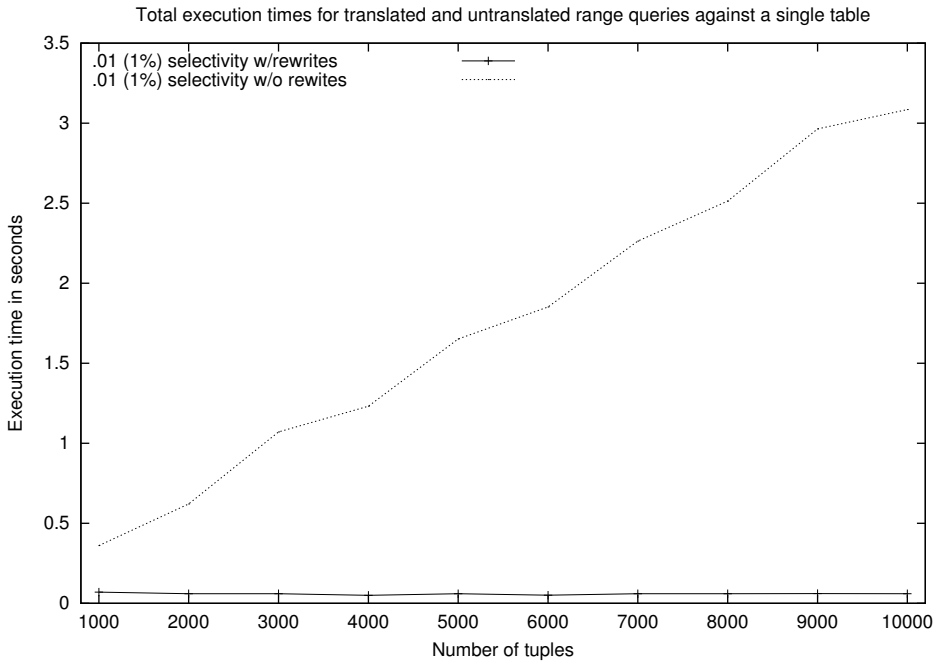


Figure 4: Total execution times for range queries involving a single table with varying size. The range selected was always .01, or one percent of the total table size.

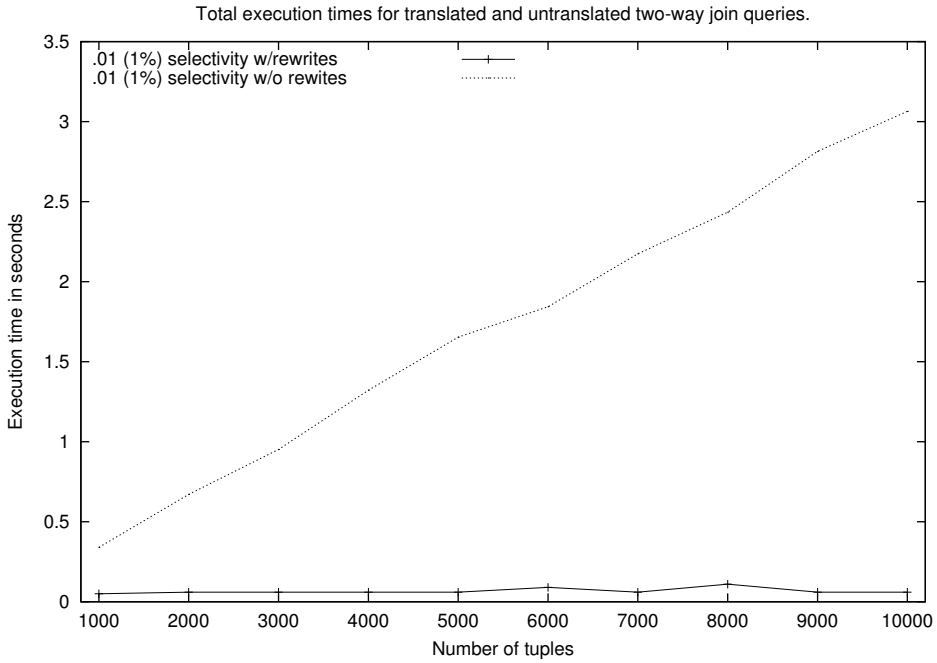


Figure 5: Total execution times for two-way join queries selecting exactly one tuple.

7 Conclusions and Future Work

In this work it is apparent that translation and pushdown of selection predicates that take advantage of a data source's capabilities can give enormous gains in execution speed. However extreme an example a relational database may be, the relational databases are the dominating storage for data.

7.1 Other Data sources

The JDBC wrapper, while fully functional, started out as an experiment platform for testing the wrapper definition API. The API has a broader use, where any data is import-able. The goal is to simplify the creation of wrappers for any data source.

7.2 Limited pushdown

What has not been considered in the development of the JDBC wrapper was the possibility that maximal pushdown may not always be desirable. For example, a query with a cross-product (a join with no join condition) will lead to increased transfer between the database and Amos if computed at the source rather that doing the join in Amos. If the connection is slow then the latter approach is desirable. The problem is not trivial however, since one needs some kind of statistics of both the data source and the connection speed in order to infer when a join actually produces more than the sum of the size of the involved types. It remains to be seen whether this can be done using the API as-is or if the design needs to be altered to accommodate for this.

References

[1] Fahl G, Risch T: Query processing over object views of relational data. *The* VLDB *Journal, Vol. 6 No. 4 November*, 1997, pp 261-281.

[2] Litwin W, Risch T: Main Memory Oriented Optimization of OO Queries using Type Data log with Foreign Predicates, In IEEE *Transactions on Knowledge and Data engineering 4(6),* 1992, pp 517-528

[3] Fishman DH et al: Overvies of the Iris DBMS. In *Kim W, Lochovsky FH (eds): Object-oriented concepts, databases and applications,* ACM *press*, 1989, pp 219-250, Addison-Wesley, Reading, Mass.

[4] Josifovski, Schwarz, Haas, Lin: Garlic: A New Flavor of Federated Query Processing, ACM SIGMOD *Conference,* 2002.

[5] Shipman DW. The functional data model and the language daplex. TODS *6(1)*, 1981, 140-173, 1981.

[6] P. Gray: Logic, algebra and databases, ISBN *0-89791-169-1*, ACM *Press*, 1984, pp 437-443.

[7] Garcia-Molina H, Ullman JD, Widom J: Database Systems the Complete Book, ISBN *0-13-031995-3*, Prentice Hall, 2002, pp 1057-1069.