Integration of Heterogeneous Data Sources with Limited Capabilities in the Object-Oriented Mediator Engine AMOS *II*

Jörn Gebhardt
Laboratory for Engineering Databases, Linköping University, Sweden

September 1999

Abstract

Information becomes a more and more valuable asset in today's organizations. Therefore the need of creating an integrated view over all available data sources arises. Several technical problems must be overcome in the design and implementation of a system for integrating different data sources. To the main obstacles count autonomy, data heterogeneity and different query capabilities of the repositories.

This thesis presents the data integration system AMOS *II*, which is based on the wrapper-mediator approach. The main focus of this work lies on data model transformation and query processing. The following extensions to the AMOS *II* system are described in this thesis:

- A framework for transforming various data models into the object-oriented model of AMOS *II* is presented.
- The roles and tasks of wrappers are described. In particular their participation in query processing and query optimization is discussed.
- A way for describing and utilizing the query capabilities of the different data sources is proposed.
- Two different approaches to query processing over external data sources are developed and analyzed.

All the proposed techniques are implemented in the AMOS *II* system, which runs on a Windows NT platform.

Acknowledgments

Foremost, I would like to thank my supervisor, Professor Tore Risch, for all the fruitful discussions and all his support. He always had time for me when I needed his help and his enthusiasm gave me a lot of motivation. He created an atmosphere of trust and openness in the laboratory that helped me to succeed with my work and to enjoy doing it. I am also grateful to all the other members of the EDSLAB research group. Especially I want to thank Timour Katchaounov for all his patience and support. He was always willing to answer all my annoying questions and got never angry about my disruptions. Furthermore, I want to thank Vanja Josifovski for great discussions during all times of the day. Those conversations usually inspired me to develop new theories.

Furthermore, I want to thank Professor Peter Lockemann and Jutta Mülle, both from the University of Karlsruhe, who enabled my stay in Sweden in the first place.

And last but not least I want to thank my closest family for the generous support and gentle care during my six years of studies. They let me go my own ways, and I always knew that they were there for me when I needed them. I also thank my father Dr. Friedrich Gebhardt for the careful proofreading of the second and third chapter.

Contents

List of Figures

Chapter 1

Introduction

Information becomes a more and more valuable asset in today's organizations. Nowadays it is very important to have information from the whole company available in every department for making good business decisions. The organizational structure of enterprises is changing towards a more integrated and interdependent system. In former days companies usually used a horizontal differentiation, often referred to as departmentalization. The departments were organized either by function (production, finance, personnel etc), by work process, by location, by product or by some combinations of the those four. Communication between the different departments was kept to a minimum and often took place on the managerial level only. However, a rapid changing environment, shorter product life cycles and a higher dependency between the departments lead to two changes. Firstly, cross-functional teams were often created and, secondly, responsibility was pushed down the hierarchy.

This new way of doing business increased the requirements requested from information systems. Information concerning the whole enterprise is needed for day to day activities as well as for long-term decision making. In former days every department has had its own isolated information system with very specialized applications, but now the information stored in these systems is also needed outside the particular department. The development in network technology bridged the physical gap between these systems, however, the data integration problem remained.

A lot of research was spent on integrating different data sources, and two solutions became very popular. The concept of *datawarehousing* was used to store all data in a new database system and extract the needed information out of these very huge centralized databases. However, this concept does not allow to make updates in the original data sources and can only be used for data retrieval or data mining. That is why Wiederhold [31] proposed another approach, namely the *wrapper-mediator approach*, that divides the data integration system in two functional units. The *wrappers* provide access to the data sources and transform the data model into a common data model. The *mediator* provides an integrated view over the different data sources and a query language for accessing the data. The user does not know where the data originates from but is able to retrieve and update all data by using a common query language.

During the early research stages of data integration the main focus was kept on contents of the sources and their relationship to the integrated views provided to the users. But now it becomes more and more important to keep also track of the capabilities of the sources, as the diversity of integrated data increases. This leads to both, the integration of data and the introduction of new operators in the mediator. Not knowing the capabilities of the sources can lead to the creation of non-executable or very inefficient plans.

This thesis presents the data integration framework for data sources with limited capabilities as implemented in the mediator database system AMOS *II*. AMOS *II* is an object-oriented mediator system that has been developed in the EDSLAB at the University of Linköping, Sweden.

The main contributions of this work to the AMOS *II* system are:

- A mechanism for integrating data from sources with diverse data models is introduced.
- The role of a wrapper and its interface to the AMOS *II* mediator system is clearly defined.
- A solution for handling diverse query capabilities is proposed and implemented.
- The problem of query optimization over heterogeneous data sources is discussed and two different approaches to query processing are implemented and evaluated.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of the AMOS *II* system. It describes the state of the system before

the work of this thesis was done and focuses mainly on the query processing part of AMOS *II*. In Chapter 3 my work carried out in the context of this thesis is presented. The data model integration framework is explained, the way of handling different query capabilities is described and the modifications in query processing system are presented. The proposed mechanisms are evaluated in Chapter 4. In Chapter 5 a comparison to related research projects is given. A summary of the work and future work is described in Chapter 6. In Appendix A a list of the used abbreviations can be found.

Chapter 2

Overview of the AMOS II System

2.1 System Structure

AMOS *II* (Active Mediator Object System) is an object-oriented, open, light-weight, and extensible database management system (DBMS). It has its roots in the workstation WS-Iris system [18, 7]. To achieve good performance AMOS *II* is designed as a main-memory DBMS.

AMOS *II* is both a DBMS of its own and a distributed mediator system [31]. Therefore AMOS *II* contains all the traditional database facilities, such as a storage manager, a recovery manager, a transaction manager, and an OO query language, named AMOSQL. Acting as a mediator AMOS *II* has facilities for translating, combining, reconciling, and abstracting data through OO views as well as over other mediators and external data sources.

Figure 2.1 shows an example of a distributed AMOS *II* system. The following AMOS *II* terminology is used in this thesis:

Mediator The term *mediator* was introduced by G. Wiederhold [31] who defines a mediator as "*a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications*". Mediators provide a common data model to applications and hide the heterogeneity of the underlying data sources. Another commonly used term for mediators is *database middleware* [24, 12].

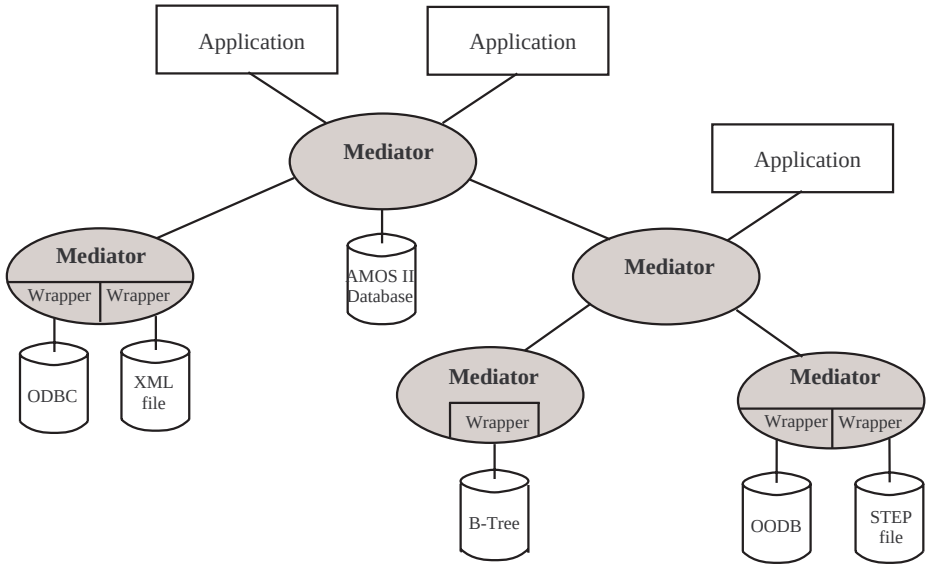

Figure 2.1: An example of a distributed AMOS *II* system.

Extensible Mediator An *extensible mediator* is a mediator whose knowledge and functionality can be extended by dynamically integrating wrappers (see below) for new kinds of external data sources. In the rest of this thesis the term *mediator* is used in the meaning of an *extensible mediator*. AMOS *II* is an extensible mediator system that appear a a virtual database and we will refer to the non extended AMOS *II* system as the *core mediator system*. Furthermore, AMOS *II* mediators are *composable* since a mediator server can regard other mediator servers as data sources.

Wrapper A *wrapper* is an interface between a mediator and an external data source type and encapsulates the knowledge about the query capabilities of that data source. A wrapper is an embedded subsystem in an AMOS *II* mediator. Wrappers are described in more detail in Chapter 3.

Different interconnecting topologies can be used for connecting AMOS *II* servers depending on the integration requirements of the environment. A single AMOS *II* server can perform more than one task and serve

more than one application simultaneously.

AMOS *II* is implemented on the Windows NT/95/98 platform whereby AMOS *II* servers communicate over TCP/IP.

2.2 Data Model

AMOS *II*'s data model in an OO extension of the DAPLEX [26] functional data model. There exist three basic constructs: *objects, types* and *functions*.

2.2.1 Objects

Objects model every entity in the database. There exist two kinds of object representation, *surrogates* and *literals*. The *surrogate objects* have associated object identifiers (OIDs) which are explicitly created and deleted by the user or the system. Examples of surrogates are "real-world" objects such as persons, and meta-objects such as functions. *Literal objects* are self-describing system maintained objects without explicit OIDs. Examples of literal objects are numbers and strings. Literal objects can also be *collections* of other objects. *Vectors* (one-dimensional arrays of objects) and *bags* (unordered sets with duplicates) are the collections supported by AMOS *II*.

Surrogate objects persist in the database until they are no longer referenced from any other object or from external systems. The removal of unreferenced objects is done through an automatic garbage collector.

2.2.2 Types

Objects are classified into *types* (i.e. classes) making each object an *instance* of some types. The set of all stored instances of a type is called the *extent* of the type. The types are organized in a type hierarchy. The AMOS *II* data model supports multiple inheritance, but requires an object to have a single most specific type. If an object is an instance of a type, then it is also an instance of all the supertypes of that type.

There exist five categories of surrogate types:

Stored types have their instances stored in the local AMOS *II* server. The instances are created by the user.

Derived types are specified through a declarative query over their supertypes. Their extent is a subset of the extents of one or more *constituent*

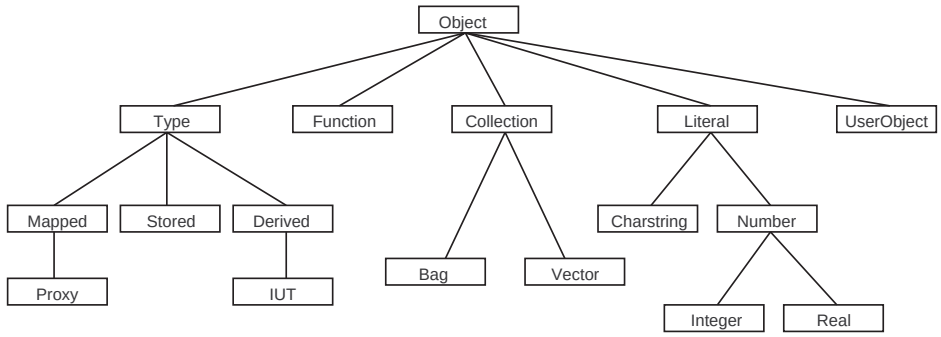

Figure 2.2: Part of the AMOS *II* type hierarchy.

supertypes [14].

Mapped types represent views on the state of an external data source (see Chapter 3).

Proxy types represent objects stored in other AMOS *II* servers.

Integration Union Types provide a mechanism for schema integration of multiple data sources.

Figure 2.2 shows parts of the AMOS *II* type hierarchy. Note that this hierarchy describes the meta types. The types created by the user such as `Person` and `Student` are instances of the meta type Type and their hierarchy has the type UserObject as root.

2.2.3 Functions

Functions model properties of objects, computations over objects, and relationships between objects. Functions are instances of the meta-type `function`.

A function consists of two parts, the *signature* and the *implementation*. The signature defines the types of the arguments and the results of a function. For example, the signature of the function modeling the attribute `name` of type `person` could have the signature `name(person)->charstring`. The implementation specifies how to compute the result of a function given a tuple of argument values. For example `name(p)` obtains the name of a person by accessing the database.

AMOS *II* functions are, furthermore, often *multi-directional*, meaning that the system is able to inversely compute one or several argument values if (some part of) the expected result value is known [18]. This means that there exist different implementations of the same function depending on which variables are bound. Inverses of multi-directional functions can be used in database queries and are important for specifying general queries with function calls over the database. For example, the following query, which finds the age of the person named 'Tore', uses the inverse of function `name()`:

```
select age(p)
  from person p
  where name(p)='Tore';
```

Depending on their implementation the basic functions can be classified into stored, derived, foreign, mapped functions, and database procedures:

Stored functions represent properties (attributes) of objects stored in the database. Stored functions correspond to attributes in OO databases and tables in relational databases.

Derived functions are functions defined in terms of OO queries over other AMOSQL functions. Derived functions cannot have side effects and the query optimizer is applied when they are defined. Derived functions correspond to side-effect free methods in OO models and views in relational databases. AMOSQL has an SQL-like `select` statement for defining derived functions.

Foreign functions are implemented through an external programming language. Currently there exist interfaces between AMOS *II* and the programming languages Lisp, C and Java. Multi-directional foreign functions correspond to methods in OO databases and provide access to external storage structures similar to data 'blades', 'cartridges', or 'extenders' in object-relational databases. To help the query processor, a multidirectional foreign function can have several associated access path implementations with cost and selectivity functions.

Mapped functions represent functions in other databases.

Database procedures are functions defined using a procedural sublanguage of AMOSQL. They correspond to methods with side effects in OO models.

Functions can furthermore be overloaded meaning that they can have different implementations, called *resolvents*, depending on the type(s) of their argument(s).

2.3 Query Language

The query language AMOSQL is based on OSQL [20] with extensions of overloading, mediation primitives, multi-directional foreign functions [18], late binding [8] and active rules [27]. Furthermore, AMOSQL has aggregation operators, nested subqueries, disjunctive queries, quantifiers, and is relationally complete. It is both, a data definition language (DDL) as well as a data manipulation language (DML). The following example illustrates data definition constructs, defining the type `Person` with three stored functions `name(), age()` and `SSN()`[1] representing the type's properties. The result of a function is always a bag of objects and the keyword `key`, e.g. in the declaration of `SSN(Person)`, is used to guarantee the uniqueness of the result.

```
create type Person;
create function name(Person) -> Charstring as stored;
create function SSN(Person) -> Integer key as stored;
create function profession(Person) -> Charstring as stored;
create function parent(Person) -> Person as stored;
```

Queries have the following syntax:

```
select <result>
  from  <type declaration for local variables>
  where <condition>
```

Here is an example of creating a functional view on type `Person` using a derived function. The from clause refers to the extent of type `Person`:

```
create function teenagers() -> Charstring  as
  select name(t)
    from Person t
    where age(t) >= 13  and
          age(t) <= 19;
```

[1] SSN stands for Social Security Number.

A more detailed description of the data model and of the query language AMOSQL can be found in AMOS *II* user's guide [9].

2.4 Query Processing in AMOS *II*

As AMOSQL is a high-level query language the queries must be optimized before execution for achieve reasonable execution times. The query compiler translates AMOSQL statements first into object calculus expressions, then optimizes and rewrites them before they get finally translated into executable algebra expressions. These calculus and algebra expressions are internally represented in a simple logic based language called ObjectLog [18], which is an OO dialect of Datalog [29]. The optimization takes place in a series of different steps. The calculus generator translates the query into a number of predicates. Then view resolution takes place followed by optimization steps for reducing the number of computations. For distributed multi-database queries the query decomposer distributes each object calculus query into local queries to be executed in the different distributed AMOS *II* servers and data sources. A cost-based optimizer on each site determines the execution order of the local predicates based on statistical cost estimates and translates the predicates into procedural execution plans. The query interpreter finally interprets the optimized algebra to produce the result of a query.

The different steps of the query processing mentioned above are shown in Figure 2.3. The following subsections will describe these steps in more detail by using an example query that retrieves all students and the names of their parents:

```
select p, name(parent(p))
  from Person p
  where profession(p) = 'student';
```

AMOSQL treats ad hoc queries as functions without arguments, therefore an ad hoc query is transformed into an anonymous derived function `query()` without any arguments. This function is then optimized, executed and discarded. For our example the function `query()` looks like this:

```
create function query() -> <Person, String>
  as select p, name(parent(p))
       from Person p
       where profession(p) = 'student';
```

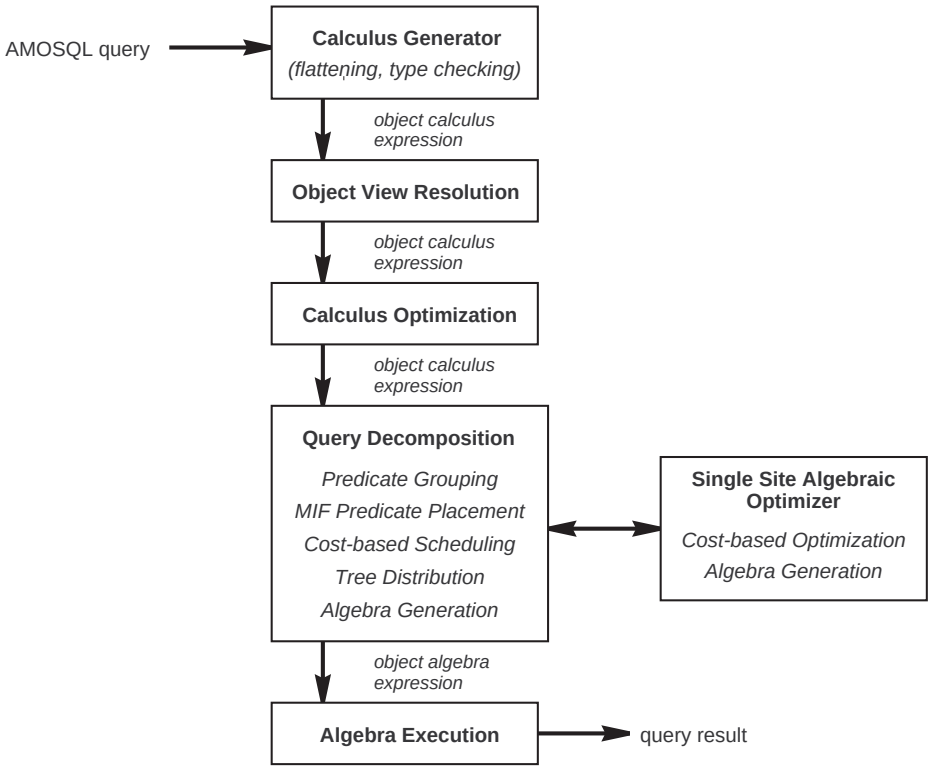

Figure 2.3: Query Processing in AMOS *II*.

2.4.1 Object Calculus Generation

The AMOSQL query is translated into a *flattened* and *type resolved object calculus* expression. *Flattened* means that there remain neither functions in the result list nor any nested function calls. This is achieved by introducing intermediate variables for each nested function call. Consequently, the calculus expression consists of a set of equality and inequality predicates, where the left hand side is either a variable or a constant and the right hand side is either an unnested function call, a variable, or a constant. The object calculus query representation does not impose any evaluation order of the calculus predicates.

As AMOS *II* supports overriding and overloading, the correct resolvent of a function has to be determined. This is done by the *type checker* that annotates the signature to every function call. In those cases where due to polymorphism late-bound function calls are needed, type resolution must be done during runtime [8]. Whenever the type of a variable cannot be guaranteed to be of the desired type (e.g. when the input of a function must be a specific subtype of the type returned by another function), the type checker adds dynamic type checks to the function definition.

The following object calculus is generated for our example query:

$$\begin{aligned}\{p, nm| \\ & p = Person_{nil\rightarrow person}() \wedge \\ & pa = parent_{person\rightarrow person}(p) \wedge \\ & nm = name_{person\rightarrow string}(pa) \wedge \\ & 'student' = profession_{person\rightarrow string}(p)\}\end{aligned}$$

The type check predicate in the first line is added by the system to ensure that the variable p is bound to an object of the extent of the type $person$. The $Person()$ is an extent fucntion that returns all stored instances of its type.

2.4.2 Object View Resolution

AMOS *II* allows to create object-oriented views by creating *derived types*. These can be both subtypes and supertypes. During the *object view resolution* phase queries over derived types are expanded by including predicates for resolving the views and guaranteeing consistency. For a detailed description of derived types and object view resolution refer to [14].

2.4.3 Calculus Optimization

During the calculus optimization phase the optimizer tries to reduce the number of predicates by applying rewrite rules for removing unnecessary computations. Optimization utilizes the declarative unordered format of the object calculus expression. There exist two different types of rewrite rules:

Type Check Removal: The referential integrity system guarantees that the arguments and results of stored function are of the declared type. Consequently, type checking is only needed to assure that derived functions return the correct type [18].

In our example query the type check predicate $p = Person_{nil \rightarrow person}()$ can be removed, as p is an argument in the stored function *parent(p)*. The new calculus expression looks like this:

$$\begin{aligned}\{p, nm| \\ & pa = parent_{person \rightarrow person}(p) \wedge \\ & nm = name_{person \rightarrow string}(pa) \wedge \\ & 'student' = profession_{person \rightarrow string}(p)\}\end{aligned}$$

In case that the argument types of *name(p)* and *age(p)* were supertypes of *person* the type check would have remained to ensure that the anonymous function query() returns the correct type *person* and not a supertype of it.

Predicate Unification Rule: When two predicates have the same predicate symbols, the same constants/variables in the key attributes, and there are no conflicts between the constants of the non-key attributes, then these two predicates can be combined into one [6]. Consider the following example:

$$\begin{aligned}\{nm_1| \\ & nm_1 = name_{person \rightarrow string}(p) \wedge \\ & nm_2 = name_{person \rightarrow string}(p) \wedge \\ & f_{string \rightarrow boolean}(nm_1) \wedge \\ & g_{string \rightarrow boolean}(nm_2)\}\end{aligned}$$

The argument p of *name(p)* is the key value, therefore the first two predicates can be unified and nm_2 is substituted by nm_1 in all other

predicates:

$$\{nm_1| \\ nm_1 = name_{person \rightarrow string}(p) \wedge \\ f_{string \rightarrow boolean}(nm_1) \wedge \\ g_{string \rightarrow boolean}(nm_1)\}$$

Note that this unification can only be done when the function *name(p)* has no side-effects. That is why all functions that cause side-effects (i.e. database procedures) are tagged with a flag to prevent the optimizer from applying this rule.

2.4.4 Query Decomposition

The query decomposition phase determines an execution schedule and assigns every predicate to a site where it is executed. The goal of this phase is to choose a low-cost execution plan. This is achieved by applying heuristics, which reduces the search space for possible plans in order to cut down calculation costs. However, by using heuristics it is not guaranteed that the cheapest of all possible plans will be generated.

Our running example is not a multi database-query and does not need any decomposition. Therefore it goes directly to the single-site algebraic optimizer described in the next subsection.

Before the query decomposition process is briefly described, a classification for the AMOS *II* functions will be introduced. This classification is based on the number of places where a function is defined and executable. There exist two different categories [15]:

1. single implementation functions (SIFs)
2. multiple implementations functions (MIFs)

The functions of the first category can only be executed at exactly one site. For example all *stored functions* are only defined at the site that stores the data. In contrast, the functions of the second class are defined at more than one data source. They are executable at all sites that run the same data source system. For example comparison operators (<, >, =, etc.) are defined in every AMOS *II* and relational database server.

The query decomposition process runs through the following five steps [15] (see Figure 2.4):

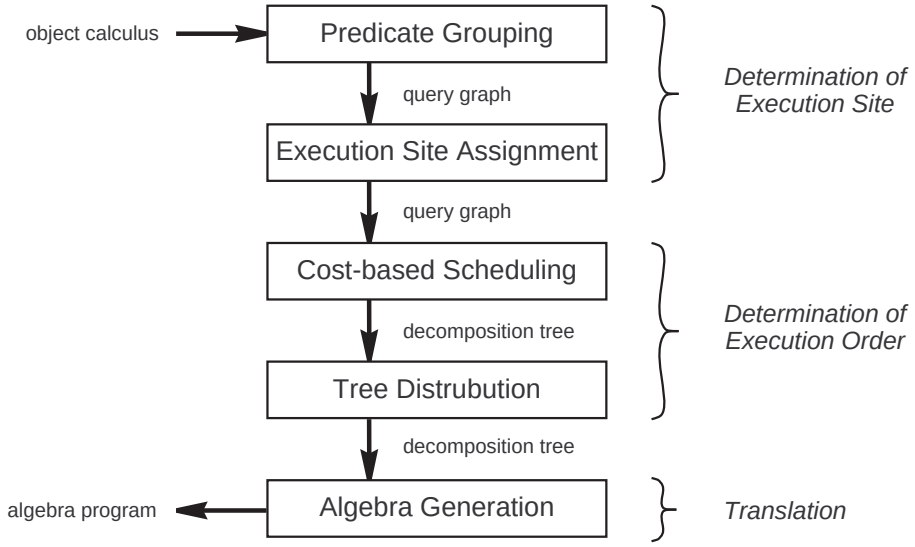

Figure 2.4: Query Decomposition Phases.

1. Predicate grouping
2. MIF predicate execution site assignment
3. Cost-based scheduling
4. Decomposition tree distribution
5. Object algebra generation

The first two steps determine the execution site of a predicate. The following two steps consider the predicate execution order, and finally, the last step translates the chosen execution plan into an object algebra expression. Each of these steps will be described now.

The Predicate Grouping Phase

A calculus expression can be represented as an undirected so called *query graph*, where the nodes represent the query predicates and edges connect those nodes that contain a common variable. Those variables that connect a node with the rest of the graph are called *node arguments*.

As the SIF-nodes can only be executed at the one site they are implemented at, obviously, this site is going to be their execution site. However, MIF-nodes can be executed at more than one site. But before the MIF-nodes site assignment takes place some nodes of the query graph are fused. Two nodes are fused when the following conditions apply:

1. Both nodes are of the same data source type (e.g. AMOS *II* or relational database).
2. Both nodes are assigned the same execution site.
3. The underlying data source type has join capabilities, so that it can handle the grouped predicates.

The fused predicates will be regarded by the system as a single predicate that is represented by a derived function, named *subquery function* (SF).

This predicate grouping has two effects. Firstly, the optimization problem is getting reduced, as there remain less nodes that have to be ordered for execution. Secondly, whenever possible are joins pushed down to the data sources[2].

MIF Predicate Execution Site Assignment

By an effective placing of the MIF predicates the query processing time can be dramatically reduced. However, calculating the costs for all possible choices is not feasible, as an exponential number of possibilities has to be examined. Therefore the following heuristics are applied:

- The introduction of additional cross-site dependencies among nodes is avoided. That is to reduce the transfer of intermediate results. Consequently, only those sites that produce intermediate results for a MIF predicate are considered as possible execution sites.
- The systems tries to combine MIF-predicates with SIF predicates so that the costs of accessing a data source and the intermediate result size can be reduced. In some cases the MIF predicate is even executed in more than one site, for example when a selection predicate is cheap

[2]At the current state of implementation AMOS *II* does not consider to perform a join locally, when it can be done remotely. Although this can lead to higher communication costs the the query optimization problem is reduced.

to execute and reduces considerably the number of tuples in two or more sites.

Based on these heuristics Josifovski [15] analyzes the different cases of the MIF predicate placement that can occur. The result of the MIF predicate execution site assignment phase is a query graph where every predicate is assigned to a site and the edges represent equi-joins over the values of common variables.

Cost-based Scheduling

The next step is to translate the *query graph* into an executable *query plan*. Therefore the query processor must determine the execution order of the graph nodes and the direction of the data shipping between the nodes. For these tasks cooperation with the single site algebraic optimizers (described in the next subsection) is needed, as they determine the execution order of the predicates and return the execution time estimates of a node for different binding patterns as well as an assessment about the number of result tuples.

As mentioned before, the grouped predicates of one node are represented by a subquery function (SF). If the assigned site of the node is an AMOS *II* server, then the SF is defined there. Otherwise, that is when the site has another data source type, the subquery function is defined in the mediator itself and is generated by the wrapper (see Section 3.6.1. The generated SF usually contains foreign function calls to access the data source. For example, the relational wrapper of AMOS *II* creates an SQL statement from the object calculus predicates and then invokes the foreign function SQL() that passes the SQL statement to the assigned data source.

The generated execution schedules for the entire query are represented by *decomposition trees* (DcTs). Each node of a DcT describes one data cycle through the mediator, as will be explained below. All children of a node have to be processed before the node itself can be executed, so the tree is processed bottom up. Note that edges between tree nodes do not represent any data flow but determine only the execution order. The data cycle through the mediator is described by two steps, the *ship and execute* operator and the *post processing list* (PPL) (see Figure 2.5). If a subquery function (SF) is defined in another AMOS *II* system then the ship and execute operator consists of the following three steps:

1. The client mediator ships the needed intermediate results to the server site where they are materialized (i.e. temporarily stored).

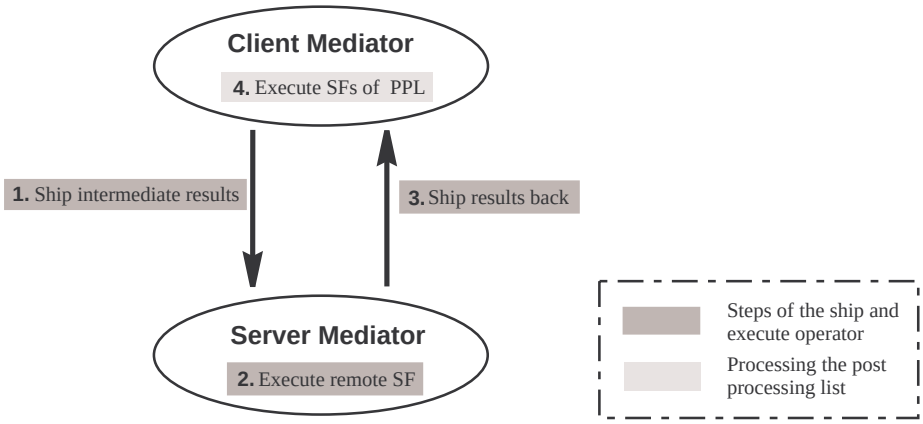

Figure 2.5: Data flow cycle described by a DcT node.

2. The remote AMOS *II* server executes the subquery function.

3. The results are being shipped back to the client and are materialized there.

If the subquery function (SF) is defined locally, the ship and execute operator is empty and no data shipping will take place. In that case the SF is included in the post processing list (PPL). In general the post processing list contains all subquery functions (i.e. query graph nodes) that are executed locally in the client mediator after the ship and execute operator has been accomplished (step 4). This list can be empty as well. Note, that intermediate results are always materialized in the AMOS *II* mediator.

To minimize the optimization problem, the system considers only left-deep decomposition trees, i.e. trees that have only one child. Therefore a total order for executing the subquery functions (SFs) is achieved. But even with this simplification there remains still an exponential number of possible decomposition trees. A variation of the dynamic programming approach is used to determine the optimal plan [15]. Note that the execution cost and the selectivity of every node depends on the binding patterns and the size of the intermediate results.

Decomposition Tree Distribution

During this query processing phase some nodes are merged and by doing so the tree is distributed over different AMOS *II* servers. By now, every data shipment passes through the mediator, which can be very inefficient. In order to avoid this, cross-communication between two AMOS *II* servers is introduced in those cases, where the mediator's role is only to forward data but not to do any processing in-between. This is the case when the following conditions for two consecutive nodes apply:

1. The child node performs a ship and execute operation to an AMOS *II* server *A*.

2. The child node has an empty post processing list (PPL).

3. The father node performs a ship and execute operation to an AMOS *II* server *B*.

In this case the execution costs for pushing the ship and execute operation of the father node down to site *A* are calculated. If this option is cheaper, all needed data is send to server *A*, where the ship and execute operation to server *B* is performed. By doing so, the original decomposition tree is distributed over many AMOS *II* servers. Refer to [16] for examples and performance measurements.

Object Algebra Generation

Next, the decomposition trees are translated into object algebra plans which then will be executed. More precisely, the algebra plan for each DcT node describes the following tasks:

- Materialize the intermediate results produced by the child DcT nodes.

- If there is a ship and execute operation then ship the needed data to the remote AMOS *II* system and execute the subquery function (SF) there.

- Execute the subquery functions (SFs) in the post processing list (PPL).

2.4.5 Single Site Algebraic Optimizer

The single site algebraic optimizer determines the execution order of the predicates and translates them into executable object algebra expressions. This optimization step is based on estimates about execution cost and fanout of the predicates with different binding patterns.

The query algebra used in AMOS *II* has the six operators $\{\pi, \times, \cup, \cap, \bowtie, \gamma\}$. The first five operators have the same semantics as their relational counterparts with the difference that input and output are bags rather than sets. The γ operator (generate operator) performs function application, i.e. a stored or foreign function is called and by doing so new elements are added to the tuples. A formal definition of the operators can be found in [6]. Note that the explicit selection operator σ is missing. It is modeled by the function application operator γ where some arguments are bound to constants.

Three different algorithms for cost-based optimization are implemented within AMOS *II*. *Dynamic programming* is used for an exhaustive search, *rank sort* [18] is used for a greedy approach with a quadratic runtime, and *random sort*, based on dynamic programming with a set of random starting points, is a compromise between an exhaustive search and greedy approaches. Refer to [13] for an overview of randomized algorithms for query optimization and to [3] for an introduction to dynamic and greedy programming techniques.

Figure 2.6 shows two different execution plans for the example query. Plan 1 is a straight forward, unoptimized translation of the object calculus. First, for all persons *p*, all parents *pa* are retrieved. The superscripts "*ff*" indicate that both parameters *p* and *pa* are *free arguments* in contrast to *bound arguments* represented by the superscript "*b*". Next, the name *nm* of all parents is retrieved. The third operator performs a selection based on the person's profession to find all students, and finally the required variables are projected from the selected tuples. Plan 2 is the optimized plan. It selects first all students before it finds their parents and gets the parents' name. Finally, a projection to the required variables takes place.

2.4.6 Algebra Execution

The interpreter uses a top-down interpretation method that corresponds to the nested-loop method in relational databases [18]. During the algebra execution phase every function is invoked with the correct binding pattern. In order to reduce data shipment overhead an advanced protocol for inter

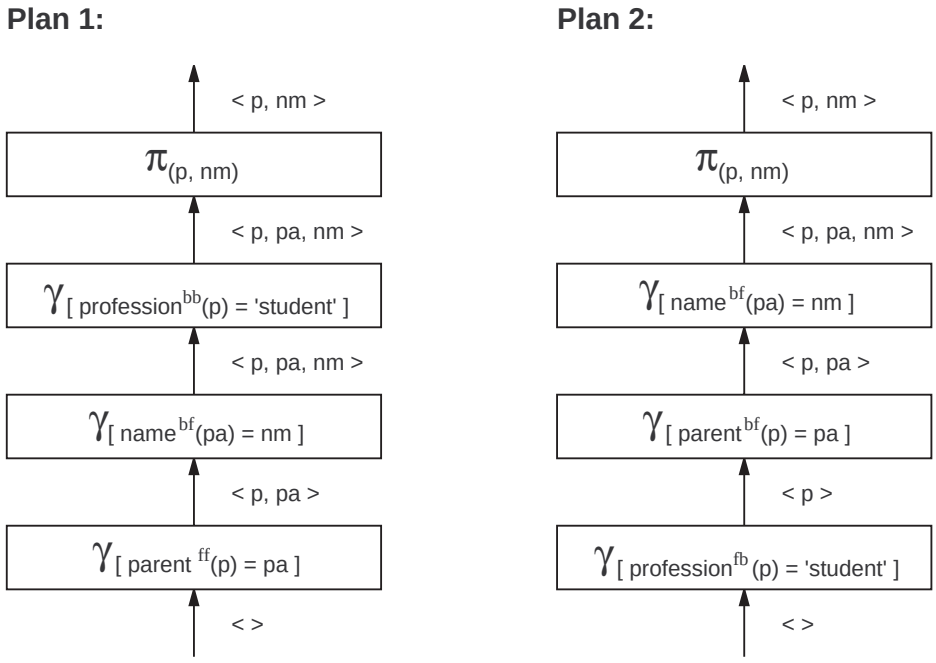

Figure 2.6: Two object algebra representations of the example query.

AMOS *II* communication is used. Data is shipped in bulks, semi-joins are applied, and in order to prohibit the sending of duplicates temporary indices are used. A more thorough description can be found in [15].

Chapter 3

Integration of External Data Sources

The aim of a mediator system like AMOS *II* is to integrate many different data sources. In difference to distributed database systems or data warehouses, the problems are more complex and diverse. Mediators try to integrate different data sources that were originally not designed and developed for being a part or subsystem of a larger database. Many challenges arise due to the heterogeneity and remaining autonomy of the underlying data sources. Mediators try to hide this diversity from the user by creating transparent views on top of them.

In order to transparently use an external data source, also called *repository*, the following problems have to be addressed in AMOS *II*:

- How to access the foreign data source?
- How to express the capabilities of the repository?
- How to transform the repository's data model into the AMOS *II* data model?
- How to integrate the schema of the repository into an existing AMOS *II* schema?
- How to optimize the query processing?
- How to get cost estimates about the execution time?

The solution to the above problems should meet a couple of requirements. The challenge is to achieve the following goals:

- There should be an easy and generic way to hook up new data sources.
- The full range of the repository's capabilities should be utilized.
- Adding new types of data sources should not need any form of modification to the query processor in the core-mediator system. It should only get extended with a new wrapper that encapsulates the needed knowledge.
- The computation time for generating the query execution plan should not grow disproportionately when new data sources are added.
- The usage of external data sources should be transparent to the user.

This chapter describes the approach developed in the context of this thesis to data integration and query processing over external data sources. The proposed methods are fully implemented and running in AMOS *II*. An evaluation of this work is presented in the next chapter and a comparison to other approaches can be found in Chapter 5.

3.1 Wrappers in AMOS *II*

There exists a wrapper for every data source type (e.g. ODBC, XML or B^+-tree) that encapsulates all the knowledge and functionality needed for integrating external data sources of that type. The functionality provided by a wrapper can be divided into three different components (see Figure 3.1):

1. The *meta-data manager* supplies the needed information for integrating external data models.
2. The wrapper's *query participator* participates in query processing and encapsulates the knowledge about the repository's capabilities.
3. Finally, the wrapper provides an *interface* for communicating with the external data source. Communication can include the retrieval and update of some data and the execution of some external operations.

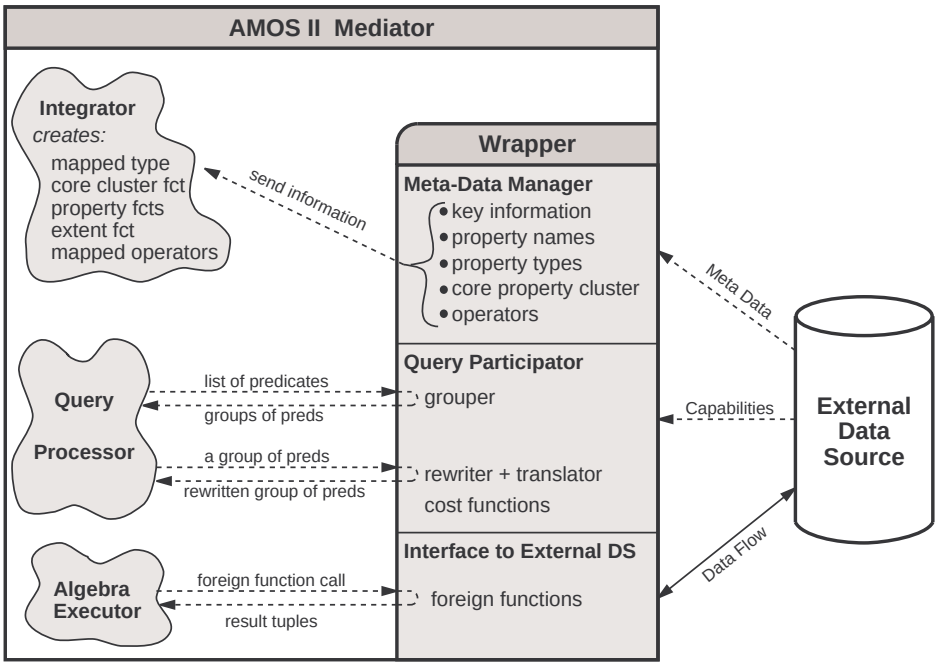

Figure 3.1: Wrappers in AMOS *II*.

An AMOS *II* mediator can be extended with any number of different wrappers to enlarge the range of repositories being accessible from the mediator. To introduce the concept of different data sources, AMOS *II*'s meta type hierarchy is extended with a new type Datasource, which is a subtype of Object. As there is a well defined interface between the AMOS *II* mediator and the wrapper, there are no changes in the core mediator system needed when integrating external data sources.

The following sections describe the three components of a wrapper and how they interact with the AMOS *II* mediator system. For illustration purposes we will first introduce a B^+-tree as an example for an external data source.

3.2 Using a B^+-tree as an External Data Source

We chose to use a B^+-tree as an example of an external data source to be accessed from AMOS *II*, and employed the ISAM Manager [21] for implementation purposes. For a description of B^+-trees refer for example to [4]. A B^+-tree is one of the simplest examples addressing the main problems that occur by including external data sources:

- The data stored in B^+-trees have a data model different from the one used in AMOS *II*. The problem of converting the B^+-tree data model into the AMOS *II* data model is addressed.

- B^+-trees have limited capabilities. Therefore the problem of how to define a repository's capabilities comes up.

- B^+-trees can handle range queries very effectively. Rewrite rules have to be applied in order to group and transform the object calculus predicates into range predicates when possible. The problem of query optimization is addressed.

- B^+-trees represent uni-directional functions. The inverse function can only be calculated by scanning the entire tree content and selecting the correct tuples afterwards. Therefore the problem of defining adequate cost models to be used during query optimization has to be solved.

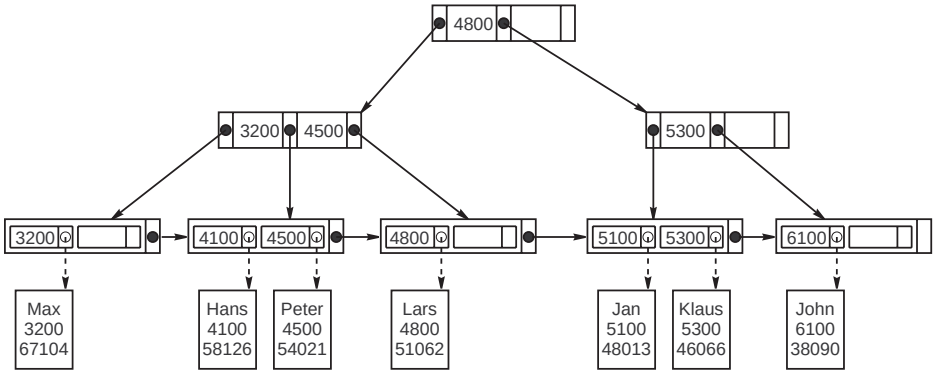

Figure 3.2: An example of a B$^+$-tree

3.2.1 Data Stored in Example B$^+$-tree

Our B$^+$-tree stores the properties `name`, `SSN`[1] and `income` of persons (see Figure 3.2). There exists an index on `income` only, and `SSN` is a unique number and can therefore be regarded as key value. The B$^+$-tree consists of 1000 records.

3.2.2 Capabilities of the B$^+$-tree

The B$^+$-tree has the following capabilities:

- Selection of all tuples stored in the B$^+$-tree by scanning it.
- Selection on the indexed properties (e.g. `income`). However, the B$^+$-tree understands only the selectors $=$, $\leq$ and $\geq$. It does not understand $<$ and $>$ as the range query returns always closed intervals. The $>$ and $<$ selections have eventually to be rewritten into a $\geq$, or $\leq$ respectively, in conjunction with a $\neq$ selection.
- Projections.

Further, the B$^+$-tree is not able to handle the following requests:

- Selections on non-indexed properties.

[1] `SSN` stands for Social Security Number.

- Joins.

These latter tasks must be performed by the AMOS *II* mediator containing the B$^+$-tree wrapper. However, as the B$^+$-tree can always return its entire content, every further processing can be done within AMOS *II*. Therefore arbitrary queries on the B$^+$-tree can be processed. This is not always the case for external data sources. For example imagine a search engine on persons that needs a person's SSN as input in order to return its name and income and that has no other way for retrieving the stored SSNs. In this case it would be virtually impossible to answer a query like

```
select income(p)
from person p
where name(p)= 'Tore';
```

3.3 Interface to External Data Sources

The interface between an extended AMOS *II* mediator and an external data source is completely based on foreign functions (see Section 2.2.3). It is up to the developer of the wrapper to provide an "adequate" set of foreign functions. What an adequate set is depends on the foreign data model and on the capabilities of the external data source. However, there exist a couple of mechanisms for achieving data integration and they require some standardized foreign functions (see next section).

The interface for retrieving data from our B$^+$-tree consists of three foreign functions. The exact-search and the range-search utilize the index and perform a selection on an indexed property, whereas the content function returns all records stored in the B$^+$-tree[2]. The access methods to the B$^+$-tree are wrapped in the following three foreign AMOS *II* functions:

```
function BT_retrieve_exact(BT-type, Index-name, Value,
                           Projection-vector)
         -> bag of tuples;

function BT_retrieve_range(BT-type, Index-name, Lower,
                           Upper, Projection-vector)[3]
```

[2]Notice that all functions in AMOS *II* return their results in a streamed way.

[3]A lower boundary of `_-inf_` represents minus infinity and an upper boundary of `_+inf_` represents plus infinity.

```
          -> bag of tuples;

function BT_retrieve_all(BT-type, Projection-vector)
          -> bag of tuples;
```

`Index-name` is the name of the indexed property the exact-search and range-search queries work on. The `Projection-vector` contains the names of the properties that are returned, and `BT-type` is a Mapped Type object, as explained below.

Here is an example for retrieving the name and the SSN of all persons stored in the type `:BTPerson` whose income lies between 4000 and 6000:

```
BT_retrieve_range(:BTPerson, 'Income', 4000, 6000,
                  vector('Name', 'SSN'));
```

Given the income, a person can be found quite easily by utilizing the existing index with the B^+-tree function `BT_retrieve_exact`. However, to find a person by its name or SSN, the entire B^+-tree must be scanned by using the function `BT_retrieve_all` followed by a select statement on the intermediate results, as these properties are not indexed.

Each of the above functions has a cost. *Cost* is defined in terms of the actual *calculation cost* of the function and its *fanout*, i.e. the expected number of records returned. Accessing the memory has a calculation cost of two. However, the B^+-tree retrieves its data from the disk, which is about a 1000 times slower than memory access. That is why we set the calculation cost to 2000 per record retrieved[4]. The fanout uses some statistics about the number of records stored and the interval range of the indexed attribute. Consequently, the exact search function is normally cheaper than the range query, and the content function is the most expensive operation.

Note that the B^+-tree access functions are not meant to get called directly by the user but that the B^+-tree wrapper translates parts of a query into these function calls.

3.4 Data Model Transformation

External data sources can have all kinds of different *data models*. The data can be structured, semistructured or totally unstructured, and it is the

[4]We assume that the data is not clustered. Otherwise a more elaborate cost model must be defined based on the average number of records stored in a cluster.

wrapper's task to translate the foreign data model into the data model of AMOS *II*. OO views are created on external schemas by using the wrapper's data model transformations.

For creating OO views on foreign data models, AMOS *II* provides three basic concepts: *mapped types*, *mapped objects* and *mapped functions*. Fahl and Risch [6] define a mapped type as "*a type for which the extension is defined in terms of the state of an external database*". Correspondingly, we define a *mapped object* as an object whose state is defined in terms of the state of the repository, and a *mapped function* as an operation that is defined in an external data source and has a mapped type as one of its arguments.

Furthermore, we divide the group of mapped functions into two classes. A *directly mapped function* has always a one-to-one mapping to a multi-directional foreign function call and the system replaces every mapped type in its argument list by the primary keys of the mapped type. For all the other arguments there exists a one to one mapping between the mapped function and the foreign function. More formally spoken, let *DMF* be a directly mapped function that is mapped to the foreign function *FF* and has the arguments $\{MO_1, \ldots, MO_n, a_1, \ldots, a_m\}$ and the results $\{r_1, \ldots, r_l\}$, where MO_x is a mapped object with the key values $k_{x1}, \ldots, k_{xj_x}$. Then during query processing a call of $DMF(MO_1, \ldots, MO_n, a_1, \ldots, a_m)$ will get replaced by the foreign function call $FF(k_{11}, \ldots, k_{1j_1}, \ldots, k_{n1}, \ldots, k_{nj_n}, a_1, \ldots, a_m)$ which returns the same results $r_1, \ldots, r_l$ as DMF.

For clarification consider the following example. `contains(File, String)` is a directly mapped function that checks whether a File contains a String. File is a mapped type and has name and path as key values. `contains` is directly mapped to the foreign function `ffcontains(name, path, expr)` that has the name and path of the file and the string expression as arguments. If `f` is a mapped object that represents the file with the name `test.txt` and the path `C:\documents` then the AMOS *II* system replaces a function call `contains(f, 'wrapper')` by `ffcontains('text.txt', C:\documents, 'wrapper')`.

An *indirectly mapped function* is sent to the wrapper where it gets translated into foreign function calls. Which foreign functions are called with which parameters depends on the context of the mapped function. It is the wrapper's query participator that encapsulates this knowledge.

Using these concepts an OO view over external data is created in the following way:

1. Group a set of attributes together to a new mapped type and declare the type of every attribute. The attributes are called the *properties* of the mapped type.

   In our B^+-tree example we group the three attributes `SSN`, `name` and `income` together to a mapped type called `BTPerson`. `SSN` and `income` are of type Integer and `name` is a String. Another example of an external data source could be a text file manager, where the attributes `name`, `path`, `date`, `size` and `content` are grouped together to a mapped type `TextFile`.

2. Declare which properties form the primary key of the mapped type[5].

   `SSN` is the primary key for `BTPerson`, and the conjunction of `name` and `path` is the primary key for `TextFile`.

3. Divide the properties into different groups. The *core cluster* contains the key properties and all the other properties that are "cheap to retrieve". "Cheap to retrieve" means that it is not worthwhile to retrieve them in separated foreign function calls as it would be more expensive to invoke these additional function calls than to retrieve eventually not further needed data. The properties of this core cluster are called *core properties*. All the remaining properties, i.e. the *non-core properties*, are unclustered and get individually retrieved.

   For clarification consider the following query over `TextFile`, where the function `substring` is implemented in AMOS *II* and checks whether the second argument is a substring of the first argument:

   ```
   select path(f) from TextFile f
     where substring(name(f), 'wrapper');
   ```

   To find the required results the names of all existing files must be examined if they contain the substring wrapper. Here, it is cheaper to retrieve `name` and `path` of all files in one call rather than to get first only all file names and then invoke an individual function call for retrieving the pathname of every matching file. However, if the contents of the matching files were also required, it would not be advisable to always retrieve them together with the name and the path.

[5]The key values are needed for identification. In case of integrating an OO database the unique remote OIDs are used as identifying attribute.

For our B^+-tree we create only the core cluster containing all three properties. However, for the file manager we group `name`, `path`, `date` and `size` together in the core property cluster and `content` forms a cluster of its own.

4. Finally, there may exist some operators on the mapped type that are not defined in the core mediator system. These operators with their argument and result types must also be declared to the mediator system.

   For example there might exist the operator `contains` on the mapped type `TextFile` that takes in a mapped file object and a string expression returns `true` if the file contains the string expression and `false` otherwise.

It is the task of the wrapper's meta-data manager to provide the AMOS *II* mediator system with all the above information. Based on this information the AMOS *II* integrator creates automatically a couple of mapped and derived functions.

First of all a new mapped type is created. This mapped type is a subtype of `UserObject` and has a `Datasource` it originates from. As mentioned above are the instances of this mapped type called mapped objects. Each mapped object needs an OID. To assure that a mapped object always represents the same record, a multi-directional function `coid_`*MappedType*`(`*KeyValues*`)->OID` is introduced, which creates dynamically an OID the first time it is needed. AMOS *II* uses mapping tables, which are stored and maintained in the mediator, for mapping OIDs to primary key values. These tables are validated during query execution if necessary [6]. An alternative way would be to have a mathematical correspondence between an OID and the key values.

Next, a mapped function named `ccluster_`*MappedType* is created that returns the content, i.e. a bag of tuples, of the core properties. We will refer to this function as the *core-cluster function*. This function is an indirectly mapped function for most data sources. Only if the data source has no other capabilities than just returning its content, this function will be directly mapped to the content returning foreign function. Furthermore, the core-cluster function might be unexecutable for some binding patterns. E.g. in the example further above, where a person's name or income can only be retrieved when its SSN is provided, the core-cluster function is unexecutable for all cases where SSN is unbound.

Next, for every property in the core cluster a *core property function* is derived. A *property function* has a mapped object as argument and returns the value of the property. The core property functions perform a projection from the core-cluster content to the desired property. For example, the core property function `name(p)` for the mapped type `BTPerson` looks like this:

```
create function name(BTPerson p) -> Charstring as
   select name
   from Integer SSN, Charstring name, Integer income
   where ccluster_BTPerson() = <SSN, name, income>
     and COID_BTPerson(SSN) = p;
```

Utilizing the core-cluster function and the key property functions an *extent function*, which returns all OIDs of the mapped type, is created. For the mapped type `BTPerson` the derived extent function looks like this:

```
create function extent_BTPerson() -> BTPerson as
  select p
  from BTPerson p, Integer ssn
  where ssn = SSN(p)
    and coid_BTPerson(ssn) = p;
```

Next, for every non-core property a directly mapped function is created. The property function `content(f)` which is directly mapped to a foreign function `ffcontent` would look like this:

```
create function content(TextFile f) -> String as
  select text
  from String text
  where ffcontent(name(f), path(f)) = text;
```

Finally, if the newly introduced operators have a mapped type in their argument list, a directly or indirectly mapped function is created. Otherwise the operators are introduced as foreign functions.

For clarification consider the following three alternative capabilities of the text file manager:

1. The file manager can test whether a string is contained in another string. Then the wrapper would provide the foreign function `contains(string expr, string subexpr) -> Boolean`.

2. The file manager can test if a string is contained in a file. Then the following directly mapped function would be created:

```
create function contains(TextFile f, String str)
                -> Boolean as
  select res
  from Boolean res
  where ffcontains(name(f), path(f), str) = res;
```

   where `ffcontains` is a foreign function provided by the wrapper.

3. The file manager can search for patterns in a text file using the UNIX command grep. In this case the wrapper would provide an indirectly mapped function `contains(TestFile f, String str) -> Boolean`. If then the user states the following query:

```
select name(f)
from TextFile f
where contains(f, 'capabilities') and
      contains(f, 'repository') and
      not contains (f, 'Garlic');
```

   the conjunction of the three `contains` function calls would get translated by the wrapper into the appropriate grep function call.

Note that the B^+-tree does not introduce any new operators as the five selection operators $\{<, \leq, =, \geq, >\}$ are already defined in the core mediator system.

For this way of data model integration a number of foreign functions must be implemented in the wrapper as interfaces to the external data source. The following listing summarizes which foreign functions are needed.

1. The mapped core-cluster function is either directly or indirectly mapped to some foreign functions that return the content of the core properties. These foreign functions must be provided by the wrapper.

2. A foreign function is needed for every property that does not belong to the core property cluster. Its arguments are the key values and its result type is the property type.

3. Finally, all operators introduced by the wrapper must have a corresponding foreign function.

3.5 Schema Integration

The OO views on external data sources can now be used for what is known as *schema integration*. AMOS *II* provides a couple of mechanisms based on sub- and supertyping for integrating different types. Conflicts and overlaps between similar real-world entities being modeled differently in different data sources can be reconciled through the mediation primitives of AMOSQL.

For example the user might want to integrate the type `BT_Person` with an already existing type `amos_person` to a new type `person`. This can be done by introducing *integration union types*. These types "*provide a mechanism for defining OO views capable of resolving semantic heterogeneity among meta-data and data from multiple data sources.*" (Josifovski [15], p. 65). Another way of schema integration is the usage of *derived types* as also described in [15].

3.6 Query Processing over External Data Sources

Integrating external data sources requires the introduction of three new steps during query processing in comparison to using AMOS *II* user types only. These phases are *grouping*, *rewriting* and *translating*. In the *grouping phase* it is determined which predicates are handled by the wrapper and which are left over to the AMOS *II* mediator. *Rewriting* is in some queries necessary for a better utilization of the repository's capabilities. The *translating phase* is used to translate the indirectly mapped functions into appropriate foreign function calls.

We implemented two different approaches to the expanded query processing as shown in Figure 3.3 and Figure 3.4. We will refer to the first one as the *decomposer push approach* and to the second one as the *cost-based pick approach*. Before these approaches are described in detail, conceive what is happening during the first three query processing phases *calculus generation, object view resolution* and *calculus optimization*.

Consider the following query that retrieves the social security number of all persons that have an income in the interval $[3000, 5000[$ and whose name is Charlie:

```
select ssn(p) from BTPerson p
where income(p) >= 3000
  and income(p) < 5000
  and name(p) = 'Charlie';
```

The calculus genarator will create the following object calculus expression:

$$
\begin{aligned}
\{ssn| \\
&\langle ssn, -, - \rangle = ccluster\_BTPerson_{nil \to Int,String,Int}() \wedge \\
&p = coid\_BTPerson_{integer \to BTPerson}(ssn) \wedge \\
&\langle ssn2, -, inc \rangle = ccluster\_BTPerson_{nil \to Int,String,Int}() \wedge \\
&p = coid\_BTPerson_{integer \to BTPerson}(ssn2) \wedge \\
&inc \geq 3000 \wedge \\
&\langle ssn3, -, inc2 \rangle = ccluster\_BTPerson_{nil \to Int,String,Int}() \wedge \\
&p = coid\_BTPerson_{integer \to BTPerson}(ssn3) \wedge \\
&inc2 < 5000 \wedge \\
&\langle ssn4, nm, - \rangle = ccluster\_BTPerson_{nil \to Int,String,Int}() \wedge \\
&p = coid\_BTPerson_{integer \to BTPerson}(ssn4) \wedge \\
&nm = \text{'Charlie'}\}
\end{aligned}
$$

For readability reasons, the elements not further used and returned by *ccluster_BTPerson* are marked with '–'.

Recall from section 2.4.3 that there exist two different calculus optimization rules, namely the *Type Check Removal* and the *Predicate Unification* rule. Especially the second one is applied frequently in queries on mapped types.

Since the four *coid_BTPerson* function calls have the same variable p as result and *coid_BTPerson* is a bi-directional function, the calculus optimizer unifies them to one predicate. $ssn2$, $ssn3$ and $ssn4$ are replaced by ssn:

$$
\begin{aligned}
\{ssn| \\
&p = coid\_BTPerson_{integer \to BTPerson}(ssn) \wedge \\
&\langle ssn, -, - \rangle = ccluster\_BTPerson_{nil \to Int,String,Int}() \wedge \\
&\langle ssn, -, inc \rangle = ccluster\_BTPerson_{nil \to Int,String,Int}() \wedge \\
&inc \geq 3000 \wedge \\
&\langle ssn, -, inc2 \rangle = ccluster\_BTPerson_{nil \to Int,String,Int}() \wedge
\end{aligned}
$$

$$
\begin{aligned}
& inc2 < 5000 \wedge \\
& \langle ssn, nm, - \rangle = ccluster\_BTPerson_{nil \rightarrow Int,String,Int}() \wedge \\
& nm = \text{'Charlie'}\}
\end{aligned}
$$

As the *ccluster_BTPerson* predicates return the same variable *ssn* in the key position, they can also be unified and *inc*2 is replaced by *inc*:

$$
\begin{aligned}
& \{ssn| \\
& \quad p = coid\_BTPerson_{integer \rightarrow BTPerson}(ssn) \wedge \\
& \quad \langle ssn, nm, inc \rangle = ccluster\_BTPerson_{nil \rightarrow Int,String,Int}() \wedge \\
& \quad inc \geq 3000 \wedge \\
& \quad inc < 5000 \wedge \\
& \quad nm = \text{'Charlie'}\}
\end{aligned}
$$

Finally, the *coid_BTPerson* predicate can be completely removed, as the variable p is not used anywhere. The following object calculus expression remains after the calculus optimization phase:

$$
\begin{aligned}
& \{ssn| \\
& \quad \langle ssn, nm, inc \rangle = ccluster\_BTPerson_{nil \rightarrow Int,String,Int}() \wedge \\
& \quad inc \geq 3000 \wedge \\
& \quad inc < 5000 \wedge \\
& \quad nm = \text{'Charlie'}\}
\end{aligned}
$$

The following subsections describe the further processing into interval operators of the B^+-tree in the *decomposer push approach* and the *cost-based pick approach*. A comparison between these two approaches including performance measurements can be found in the next chapter and a comparison to related work is presented in Chapter 5.

3.6.1 Decomposer Push Approach

In the decomposer push approach the decomposer does a pre-grouping of the predicates and pushes the predicates that concern external data sources to the corresponding wrappers (which gives this approach its name). Next, the cost-based scheduling of the joins between the different groups takes place. The rewrite and translation phase are performed in the wrapper and split each group into a conjunction of subqueries:

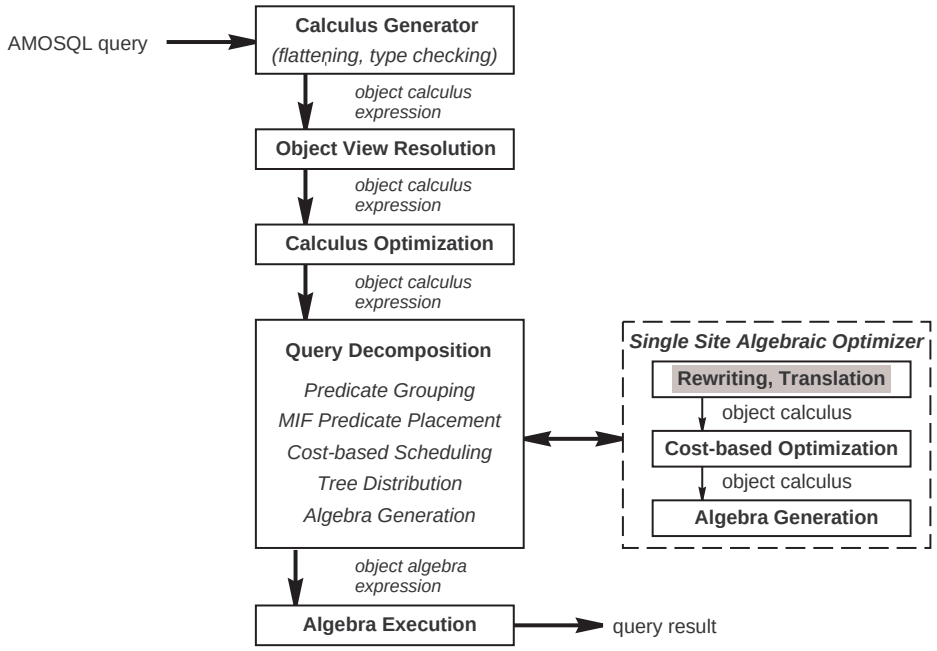

Figure 3.3: Query processing over external data sources in the decomposer push approach. As compared to the original query processor, the Single Site Algebraic Optimizer is extended by a new Rewriting and Translation phase that is performed by a wrapper.

1. Subqueries fully executable in the external data source.

2. Subqueries post-processed in the mediator.

Grouping Phase

The *grouping phase*, which determines the execution site for every predicate, is located in the query decomposer (see Figure 3.3). Recall from Section 2.4.4 that the predicate grouping phase is used to place the single implementation functions (SIF) to the sole site where they are defined and that the MIF predicate placement phase determines the execution site of the multi implementation functions (MIFs) based on some heuristics.

In order to use this mechanism not only for other AMOS *II* servers but for external data sources, too, the decomposer has to know about the repository's capabilities, so that only executable predicates are pushed to the wrapper. And, additionally, the decomposer has to know about the performance of the executable operations in the external data source. For example it could be the case that an external data source is capable of performing some operations but that it would be much faster if these operations were executed in the mediator instead.

As discussed in the literature (e.g. [24], [30]), it is very difficult to exactly specify the capabilities of a repository in a declarative way. These difficulties show up already in the case of our B^+-tree example. The B^+-tree is capable of processing $\leq$ selections on *indexed* properties only. Another question is how to handle $<$ selections. The B^+-tree cannot process them directly, however, when rewritten into $\leq$ in conjunction with $\neq$ selections, it can process the $\leq$ part.

To avoid these problems we have taken a different way in AMOS *II*, which is similar to the Garlic approach [24][6]. The external data sources do not declare what they are *exactly* capable of, but which operations they *might be able* to handle. It is then the wrapper's task to decide which predicates can be executed in the external data source, and which ones are left over to be executed within the mediator. In other words, the mediator does a pre-grouping by creating subquery functions (SFs) and asks the wrapper to perform as much work of the SF in the external data source as possible. The MIF predicates that cannot be handled by the external data source will get executed in the AMOS *II* mediator.

[6] A comparison between Garlic and AMOS *II* can be found in Section 5.1.

For example, the B$^+$-tree declares that it might be able to perform $<$, $\leq$, $=$, $\geq$ and $>$ selections. However, it can actually only perform $\leq$, $=$ and $\geq$ selections on *indexed* properties. So if the wrapper receives some selections on non-indexed properties it will leave them to be performed by the surrounding AMOS *II* mediator system.

Consider the above query. The first predicate is a single implementation function, only implemented in the B$^+$-tree, and is consequently pushed down to the B$^+$-tree wrapper. As the last three selection predicates work on the B$^+$-tree properties and $\geq, <$ and $=$ might be executable operations in the B$^+$-tree, the MIF predicate placement phase will send these three predicates to the wrapper as well. Therefore the decomposer creates only one subquery function (SF) containing all four predicates and sends it to the B$^+$-tree wrapper.

Rewrite and Translation Phase

The rewrite and translation phase is performed within the wrapper. Given a subquery function (SF), i.e. a set of predicates, the wrapper might rewrite the predicates into a semantically equivalent set and translate all the predicates executed in the external data source into new predicates for accessing the repository. The execution order of this new set of predicates is determined afterwards during the cost-based optimization phase.

In the example query the B$^+$-tree wrapper discovers that the $\geq$ and $<$ selection are applied on an indexed property. It rewrites the $<$ predicate into two new predicates containing a $\leq$ and a $\neq$ selection. The predicate $nm =$ 'Charly' cannot be handled by the B$^+$-tree as there is no index on the property *name*. The core-cluster, $\geq$ and $\leq$ predicates are translated into a range-query and the following predicate list remains:

$$
\begin{aligned}
&\{ssn| \\
&\quad \langle ssn, nm, inc\rangle = \\
&\qquad BT\_retrieve\_range_{BT\_type,string,literal,literal,vector\rightarrow vector} \\
&\qquad (BTPerson, \text{'Income'}, 3000, 5000, \\
&\qquad\quad \langle \text{'SSN', 'Name', 'Income'}\rangle) \wedge \\
&\quad inc \neq 5000 \wedge \\
&\quad nm = \text{'Charlie'}\}
\end{aligned}
$$

The first predicate will be executed in the B$^+$-tree, whereas the last two

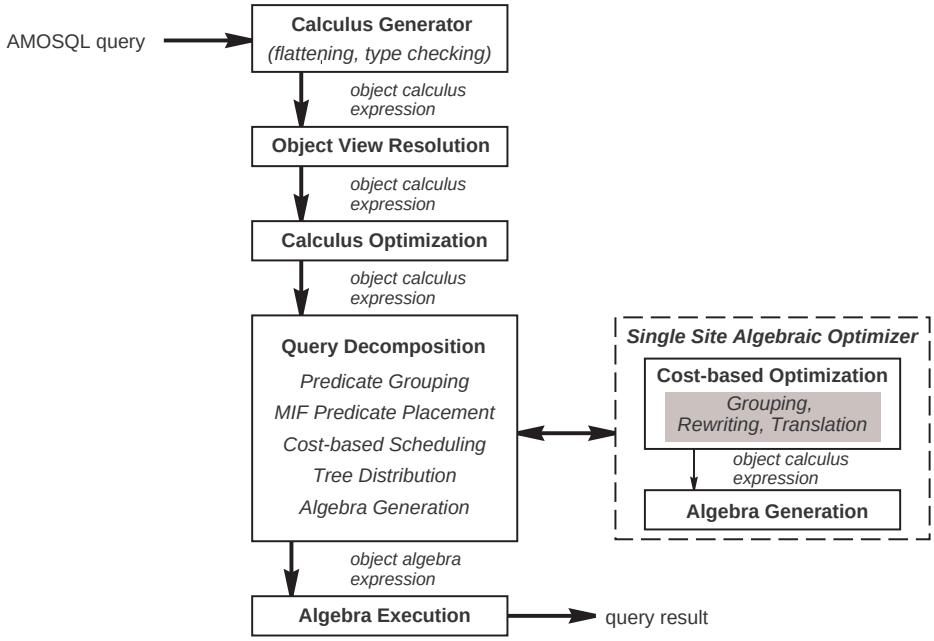

Figure 3.4: Query processing over external data sources in the cost-based pick approach. The steps performed by the wrappers are marked in grey.

predicates are post-processed in the AMOS *II* mediator system.

3.6.2 Cost-based Pick Approach

In the *cost-based pick approach* the decomposer is only used for distributing the query over different AMOS *II* mediators, but not for sending subqueries to the wrappers within the mediator system. During (and not before) the cost-based optimization phase the wrappers pick themselves a group of predicates out of the local (sub)query which they are willing to perform (see Figure 3.4). This has the effect that the MIFs are grouped dynamically rather than being statically pre-grouped in the decomposer. This means that the placements of the MIFs is determined by the costs of different join orders of the single implementation function (SIF) groups. With the decomposer push approach this join order is partially constrained by the pre-placement of the MIFs. For a direct comparison of the two approaches refer to Chapter 4.

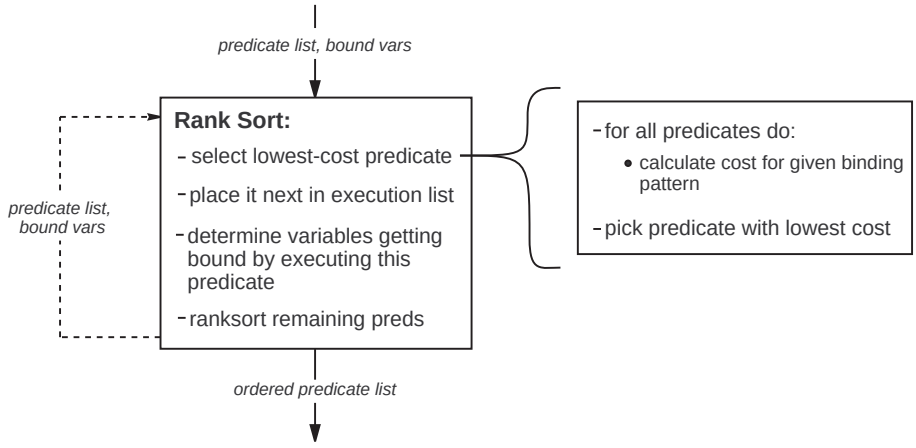

Figure 3.5: The ranksort algorithm for queries not involving external data sources.

Recall from Section 2.4.5 that AMOS *II* has three different algorithms for cost-based optimization, namely *ranksort*, *dynamic programming* and *random sort*.

Figure 3.5 describes how the ranksort algorithm works when no external data sources are involved. Ranksort is a greedy algorithm for creating an *execution list*, i.e. a list of predicates in their execution order. It calculates the expected cost for every predicate, picks the cheapest one to be executed next, and does the same procedure for all remaining predicates. Note that after every predicate placement more variables get bound. This leads for some of the remaining predicates to new binding patterns resulting in lower costs.

The remainder of this section describes how the cost-based pick approach is embedded in ranksort.

Consider the following query that retrieves the social security number of all persons whose income is greater than the budget of any department and also returns the department's name:

```
select ssn(p), name(d)
  from BTPerson p, BTDep d
  where income(p) > budget(d);
```

Both, `BTPerson` and `BTDep` are mapped types stored in a B^+-tree. `BTDep` has

the core properties `name`, `budget` and `manager`, where `name` is the key value, and consists of 10 records. After calculus optimization we have the following predicate list:

$$\{ssn, nm| \\ \langle ssn, -, inc\rangle = ccluster\_BTPerson_{nil\rightarrow Int,String,Int}() \wedge \\ \langle nm, bud, -\rangle = ccluster\_BTDep_{nil\rightarrow String,Int,String}() \wedge \\ inc > bud\}$$

Because no remote AMOS *II* server is involved in this query, the query decomposition phase is not needed and the above predicates go directly to the singe-site algebraic optimizer.

Figure 3.6 shows at what stages grouping, rewriting and translating is performed in the modified ranksort algorithm.

Grouping Phase

The grouping phase in the cost-based pick approach is different from the one performed during query decomposition in the decomposer push approach. Here, the wrappers decide which predicates belong together as these predicates will result in the same foreign function call. Every wrapper involved in the query receives all predicates and returns a list of predicate groups which it is willing to handle and which are regarded as a unit. Every predicate that does not belong to any group, i.e. that must be handled by the mediator, forms a singleton group. Additionally, for every multi-implementation function (MIF) contained in any wrapper group a singleton group is also created, as it might be cheaper to execute the MIF predicate in the mediator instead. Note that the groups are not necessarily disjunctive, i.e. a single predicate can belong to more than one group.

In our example we will get three groups in the first pass of the ranksort loop. Every predicate forms a group of its own. The greater-than selection predicate cannot be included in any of the first two groups, because both variables *inc* and *bud* are unbound in the given context and are therefore not suitable as an upper or lower boundary in a range query.

Rewrite and Translation Phase

The next step in the ranksort algorithm is to calculate the execution cost for every group. In order to get the costs for the groups created by the wrappers,

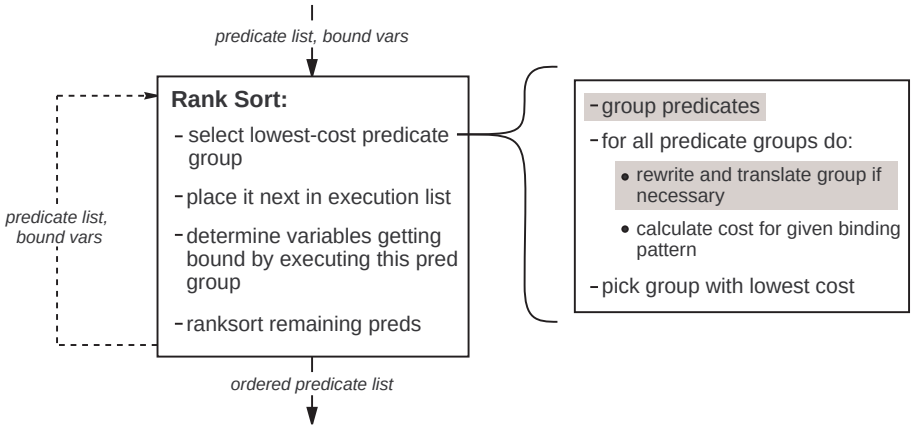

Figure 3.6: The modified ranksort algorithm for queries over external data sources. The added steps are marked in grey.

a rewrite and translation phase has to be introduced. The wrappers rewrite these groups into executable predicates for which execution costs are defined. The least expensive group is then chosen to be placed next in the execution list.

In our example the first two groups, containing only an indirectly mapped core-cluster function, will be translated into `BT_retrieve_all` predicates that return the *SSN* and *income* of all persons and the *name* and *budget* of all departments, respectively. Retrieving the information about the departments is the cheapest option, as the greater-than selection is unexecutable in the given context and there exist fewer departments than persons.

In the next pass through the ranksort loop the B^+-tree wrapper will group the two remaining predicates (i.e. the first and third) together, because this time *bud* is bound and can therefore be used as the lower boundary in a range query. Being a MIF predicate, the greater-than selection forms also a group of its own. However, as the bigger predicate is still unexecutable, the first group containing both predicates will be rewritten into a range query in conjunction with a not-equal predicate. No predicates remain and the ordered predicate list looks like this:

$$\{ssn, nm| \\ \langle nm, bud\rangle =$$

$$
\begin{aligned}
&BT\_rerieve\_all_{BT\_type,vector \rightarrow vector} \\
&(BT\_Dep, \langle \text{'Name', 'Budget'} \rangle) \\
\langle ssn, inc \rangle = \\
&BT\_rerieve\_range_{BT\_type,literal,literal,literal,vector \rightarrow vector} \\
&(BT\_Person, \text{'Income'}, bud, \text{'\_+inf\_'}, \\
&\quad \langle \text{'SSN', 'Income'} \rangle) \wedge \\
inc \neq bud\}
\end{aligned}
$$

Next, this object calculus expression gets translated into object algebra, which is then executed and the required results are returned.

Chapter 4

Evaluation of the Presented Work

This chapter evaluates the data integration framework presented in the previous chapter. The concepts of mapped types and mapped functions are analyzed, the decomposer push approach and the cost-based pick approach are compared and, finally, experimental results are presented.

4.1 Data Model Integration

The main characteristics of the data model integration are:

1. Mapped objects are used to represent an entity of the external data source.

2. The mapped objects are classified in mapped types.

3. The attributes of a type are clustered. The keys and all inexpensive attributes are grouped together in the core cluster, whereas each expensive attribute forms a cluster of its own.

4. Operators on mapped objects are modeled as foreign, directly or indirectly mapped functions.

This way of data model transformation has advantages as well as some drawbacks. I will now discuss some of the strengths and weaknesses:

1. The OIDs are created by the mediator system based on the provided key information. While this simplifies the creation of unique OIDs, these OIDs are meaningless to the wrappers. However, the system provides a function that returns the key values for a given OID. If needed, this function can be used by the wrappers to extract the key values of objects during the translation process.

2. The clustering of the properties is done statically when a new mapped type is imported. In some cases it might be better to dynamically declusterize the core cluster at run-time, for achieving a better performance. On the other hand, we do achieve some simplifications by introducing the concept of core clusters. Firstly, fewer foreign functions are needed, as all core property functions are derived from the core cluster function. Secondly, the grouping of the core cluster is done automatically. This simplifies the tasks of the wrappers as they do not have to this grouping. And thirdly, the number of created predicates during query optimization is reduced, which simplifies the optimization problem.

3. The expensive properties are individually mapped and can therefore be retrieved after some selections and joins took place. This characteristic is essential for mediators, as there exist very expensive properties for example in object-oriented or multimedia databases.

4. The concept of indirectly mapped functions gives the wrappers the possibility to fully exploit all capabilities of the data sources.

5. New operators can be added to the mediator system by either using foreign, directly or indirectly mapped functions. With these three options it is relatively easy to introduce the whole range from very simple to very complex operators.

4.2 Comparison Between the Decomposer Push Approach and the Cost-Based Pick Approach

There exist two major differences between the decomposer push approach and the cost-based pick approach. These are:

1. **The stage during query processing when the grouping, translation and rewriting is done.**

   In the decomposer push approach the grouping, rewrite and translation phases take place before the cost-based optimization, whereas in the cost-based pick approach they are done during the cost-based optimization phase.

2. **The location where the grouping is performed.**

   In the decomposer push approach the grouping is done by the decomposer that has some knowledge about the processing capabilities of the external data sources. In the cost-based pick approach, on the contrary, is the grouping done by the wrappers.

It follows a discussion of the consequences arising from these differences.

In the decomposer push approach the decomposer sends a subquery to the wrappers, and the wrappers try to execute as much of the subquery as possible in the external data sources. The wrappers first translate the predicates into one or more foreign function calls, before the foreign function calls receive an execution order during cost-based optimization. As this approach works fine for data sources with fully fledged query processors like relational databases, it can lead to suboptimal plans for data sources with limited capabilities. The following example demonstrates this.

Consider the query from Section 3.6.2 that retrieves all persons whose income is higher than the budget of any department. This query looks like follows:

```
select ssn(p), name(d)
  from Person p, Dep d
  where income(p) > budget(d);
```

After the calculus optimization these predicates remain:

$$\{ssn, nm| \\ \langle ssn, -, inc\rangle = ccluster\_Person_{nil \to Int,String,Int}() \wedge \\ \langle nm, bud, -\rangle = ccluster\_Dep_{nil \to String,Int,String}() \wedge \\ inc > bud\}$$

When `Person` and `Dep` are stored as two mapped types in a B^+-tree, each of the first two predicates gets translated into a function call that retrieves the

entire content of the type. As the order of the predicates is undetermined at this stage, the third selection predicate cannot be utilized as an upper or lower boundary. Later, during query execution, are either for all persons all departments retrieved or for all departments all persons, before the greater selection is tested within the AMOS *II* mediator system. When `Person` consists of 1000 tuples and `Dep` of 10 tuples and the salary of 45 persons is higher than the budget of one department, 10000 tuples are retrieved from the B^+-tree[1].

In the cost-based pick approach, on the contrary, translation is done during cost-based optimization. Here the grouping of the predicates is done dynamically by the wrapper. The core cluster predicate retrieving all departments will get placed first, as its fanout is lower than that of `Person`. Next, the greater-than multi implementation function predicate (MIF predicate) is grouped together with *ccluster_Person* as *bud* is now bound and can be utilized as lower boundary for a range search. This leads to a much better query execution performance of the above query. First, the 10 departments will be retrieved and then the budget of each department is used as the lower boundary for a range search over the persons. The number of retrieved tuples adds up to 55, i.e. the 10 departments plus the 45 persons whose income is high enough. Furthermore, the post-processing of the greater-than predicate is not needed any more.

However, consider that `Person` and `Dep` are tables in a relational database. Then in both approaches the entire query is processed in the relational database by using the foreign `SQL` function. The relational database optimizes the query by utilizing eventually existing indices and returns only the correct 15 results. No post-processing in the AMOS *II* mediator is needed.

The differences arise because the grouping of the predicates can be binding pattern dependent. In the B^+-tree case it is not possible to translate the entire subquery into one B^+-tree call, as the B^+-tree has no join capabilities. Therefore the wrapper must build two subgroups. In the decomposer push approach, where the grouping and translation is done before the predicate scheduling, the greater-than selection cannot be added to any group, as the execution order of the other two predicates is undetermined. In the cost-based pick approach, in contrast, the greater-than selection can be used as upper or lower boundary after one of the core-cluster predicates is placed.

[1]This holds because materialization (caching) of intermediate results is only implemented for inter-AMOS *II* communication. Therefore for every person are all departments retrieved or vice versa.

In the decomposer push approach the decomposer performs a pre-grouping of the predicates and requests the wrappers to perform some tasks based on the a-priori knowledge about the processing capabilities. This means that the decomposer has some control over the execution places of the different predicates[2]. In the cost-based pick approach, in comparison, does the core mediator system have no control at all over the grouping process. The wrappers are completely in charge of forming the groups. This has two effects. Firstly, it becomes more complicated to write a wrapper as a grouping logic must be added. The wrapper must decide how many groups are created and which predicates belong to which group. Secondly, the query optimizer does not have the chance to try different join orders and methods. These two problems with the cost-based pick approach arise mainly in those cases where the external data source is capable of performing joins.

To summarize the above discussion we realize that the cost-based pick approach can lead to better execution plans but increases the search space and therefore query optimization time. However, in the decomposer push approach the AMOS *II* query optimizer has more control over execution plan generation. A compromise could be to use the decomposer push approach for data sources with join and full query processing capabilities (e.g. relational and OO databases) and the cost-based pick approach for data sources with limited capabilities (e.g. the B^+-tree).

4.3 Experimental Results

The experiments were set up to compare the decomposer push approach and the cost-based pick approach. Furthermore, the effects of the wrapper involvement in the cost-based pick approach during cost-based optimization is analyzed. The results of these experiments confirm the expectation that the cost-based pick approach will lead to better results when the data is stored in a B^+-tree.

The experiments were performed on a Compaq Professional Workstation 5000 with a 200 MHz Pentium processor and 64 MB main memory.

[2]It does not have total control as a wrapper might refuse to handle some predicates due to limited query capabilities.

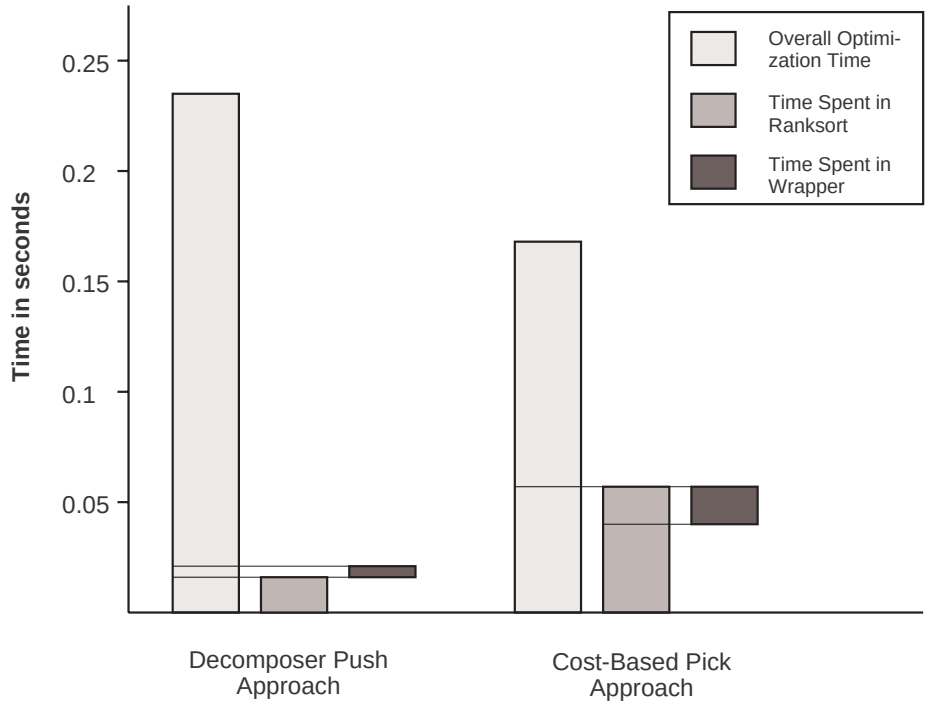

Figure 4.1: Comparison between the query optimization times.

4.3.1 Comparison Between the Decomposer Push Approach and the Cost-Based Pick Approach

This comparison concerns both the query optimization time and the query execution time. For comparison I used the query already discussed in the last section:

```
select ssn(p), name(d)
  from Person p, Dep d
  where income(p) > budget(d);
```

`Person` and `Dep` are stored in the B^+-tree and there exists an index on `income` and `budget`, respectively. Figure 4.1 shows the different optimization times needed for creating the execution plans. There exist two striking characteristics in this figure:

Number of Persons	100	1000	5000	10000
Number of Results	4	9	16	30
Execution time DP Approach	1.16s	12.5s	65.1s	127.3s
Execution time CBP Approach	0.019s	0.044s	0.057s	0.107s

Table 4.1: Comparison between the query execution times.

1. The overall time spent on query optimization is higher for the decomposer push approach, although less time is needed for cost-based optimization using ranksort.

   In the decomposer push approach 0.235 seconds were needed for query optimization. Out of this time 0.016 seconds, i.e. 6.8 per cent, were spent in ranksort. In the cost-based pick approach, on the contrary, 0.168 seconds were needed for query compilation and 0.057 seconds, which is about one third of the time, are spent on cost-based optimization. Even though this cost-based optimization time is higher than in the decomposer push approach, the overall execution plan generation time is lower. This is due to the time overhead produced by the invocation of the decomposer.

2. In the decomposer push approach 0.005 seconds are spent in the wrapper, compared to 0.017 second in the cost-based pick approach.

   While in the decomposer push approach the wrapper is called only once, it is invoked three times in the cost-based pick approach. In the case of a complex query, where the order of much more than three predicates must be determined during cost-based optimization, the overhead produced by wrapper invocation can get much more severe. This is analyzed in the next section.

Next, I analyzed the query execution time for the above query, once the execution plans were created. The type `Dep` contained always 10 objects and the number of persons stored in the B^+-tree was varied between 100 and 10000. The results shown in Table 4.1 demonstrate the huge differences between the decomposer push approach (DP Approach) and the cost-based pick approach (CBP Approach). The much shorter execution times for the cost-based pick approach arise due the utilization of the index on the property `income`.

The results show how important good query optimization is. Creating a bad plan can easily lead to execution times that are several orders of magnitude slower than the optimal plan. For example, utilizing the index in the above query increases the execution time for 10000 persons by a factor of 1000.

4.3.2 The Effects of Wrapper Involvement During Cost-Based Optimization

In the above query we already discovered that the invocation of the wrappers during cost-based optimization produces some time overhead. To determine the magnitude of this involvement we compared the query optimization times for a complex query with and without the involvement of an external data source.

A second point of interest are the changes done in the ranksort algorithm. The modified ranksort algorithm has to check whether an external data source is involved and it uses a different data structure than the original algorithm. This data structure shows the binding patterns for all variables of every predicate to provide the wrappers with all required information. To measure the costs of these changes the original and the modified ranksort algorithm are compared to each other.

The following query was used for test purposes[3]:

```
select matchinfo(m)
  from Match m, Person p
  where spectators(m) < income(p)
    and year(played_in(m)) = 1990;
```

`Match` is a type, `matchinfo` a complex derived function and `spectators`, `year` and `played_in` are stored functions in a database modeling football worldcups. `income` is either a mapped function, when the extent of type `Person` is stored in a B^+-tree or a locally stored function for the case of no external data source involvement. After the calculus optimization phase remain 17 predicates that have to get sorted during ranksort.

Figure 4.2 shows the optimization times for the three different scenarios described above. In the first case, when `Person` is stored in the B^+-tree, the B^+-tree wrapper was called 14 times during cost-based optimization

[3]Note that this is not a meaningful query and that the returned query results are of no interest.

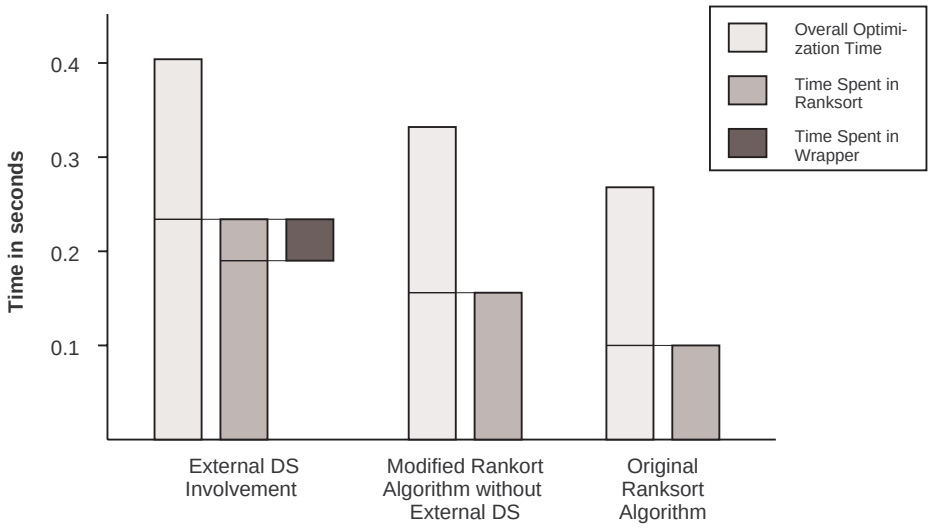

Figure 4.2: Comparison between query optimization times.

and 0.047 seconds were spent in the wrapper, which accounts to 20% of the overall time spent in ranksort. In the second case where only types stored in the local AMOS *II* mediator were used, the time spent in ranksort decreased from 0.234 seconds to 0.156 seconds, which is a decrease of 33%. And, finally, when using the original ranksort algorithm 0.11 seconds were needed for cost-based optimization. This is a decrease of 30% compared to the modified algorithm.

Two conclusions can be drawn out of these results. Firstly, the changes of the data sturucture in the ranksort algorithm lead to significantly higher calculation costs. As these changes occur in a very time critical part of the program, the implementation should be reviewed and optimized. Secondly, the wrapper involvement produces a considerable increase in calculation time. In the above query only one wrapper gets called. In the case of a query over many different data sources the time overhead for grouping and translation in every loop of ranksort gets even more severe. Moreover, for join-capable data sources efficient grouping is much more sophisticated than in the B^+-tree case, which leads to even higher calculation costs.

Table 4.2 summarizes the advantages and disadvantages of the decomposer push approach and the cost based pick approach. These results sup-

	Positive	Negative
DPA	AMOS *II* has control over predicate grouping and predicate placement. The search space is reduced by applying heuristics. The join grouping logic is centralized and encapsulated in the AMOS *II* decomposer.	The placement of the MIFs is binding pattern independent. The join-order is partially constrained by the MIF predicate placement. MIF predicates can be misplaced as they might not be executable in the external data source in the given context. Then post-processing in the mediator system is needed.
CBPA	Dynamic grouping of the MIF predicates depending on the binding patterns. Better plans can be generated. Wrappers have exact knowledge which MIF predicates they are able to process. No misplacing can occur.	The AMOS *II* query processor has no control over the grouping process. Every wrapper must contain a full grouping logic. Increased query optimization time due to frequent wrapper involvement.

Table 4.2: Comparison between the decomposer push approach (DPA) and the cost-based pick approach (CBPA).

port the supposition stated above that the cost-based pick approach should only be used for data sources with very limited capabilities, whereas the decomposer push approach is suited for databases having their own query processor. This differentiation between data sources captures the advantages of both approaches. Firstly, the grouping logic for fully fledged databases is encapsulated in the decomposer and does not have to be implemented within every wrapper. This simplifies the task of writing new wrappers. Secondly, the pre-grouping in the decomposer reduces the number of predicates that have to get ordered in the cost-based optimizer. This can lead to lower optimization times, as the cost-based optimization time grows quadratically to the number of predicates when using ranksort and exponentially when using exhaustive algorithms. And finally, the AMOS *II* query optimizer is used for data sources that do not have their own optimizer. This allows the best utilization of eventually existing indices.

Chapter 5

Related Work

This chapter describes other research projects that are in some ways related to our work. The main issues that were addressed in this thesis concern data model transformation, query processing over heterogeneous data sources, diverse data source capabilities, and semantic rewriting.

The next section describes the Garlic System [24] which is very similar to the AMOS *II* system. The TSIMMIS project [10] uses a declarative specification language for describing a repository's capabilities, as explained in Section 5.2. Finally, Section 5.3 presents other relevant research projects.

5.1 The Garlic System

The Garlic system [2, 11, 12, 25] for multi-database integration is being developed at the IBM Almaden Research Center. Garlic uses an object-oriented data model to uniformly represent data from various data sources, and an object-extended dialect of SQL as a query language. The query processing is performed by a "middleware" layer that interprets object queries, creates execution plans in cooperation with the underlying wrappers, sends pieces of queries to the appropriate data servers, and assembles the query results.

Similar to the AMOS *II* system uses Garlic also wrappers to encapsulate all the knowledge about external data sources. More precisely do wrappers provide four services to the Garlic middleware system. They are utilized to model legacy data as objects, to participate in query planning, and to provide

standard interfaces for method invocation and query execution. The Garlic query processor itself has, like the cost-based pick approach in AMOS *II*, no a priori knowledge about the query processing capabilities of the individual data sources. Instead the query processor identifies the portions of a query that concern a specific data source, and the wrapper determines what parts of the task it is able to perform and creates execution plans. There are three advantages using this non-declarative approach. First, the middleware optimizer does not have to know all the details about the capabilities and restrictions of the underlying data source. Secondly, it is much easier to introduce new kinds of data sources even if they have unanticipated restrictions or capabilities. Neither any declarative specification language nor the optimizer code has to be changed or extended. Thirdly, the start-up costs to write a new wrapper are kept small, nevertheless, the wrapper can be easily extended to exploit more of the repository's native query processing capabilities. However, there are also drawbacks compared to the declarative approach as it is used for example in the TSIMMIS [10] and DISCO [28] project. The main disadvantage is that the Garlic middleware does not know in advance which parts of the query can be handled by the repository. This can lead to a number of unsuccessful work requests.

Garlic uses a dynamic programming approach based on Lohman's framework [19] for query decomposition and query optimization. The execution plans are trees of operators that are called *POPs*, (Plan OPerators). There exist POPs for join, sort, filter (selections) etc. and also a generic POP, called *PushDown*, which encapsulates work to be done at a repository. *Properties* are attached to all plans to indicate the work they are performing. These properties include information about tables being used, predicates being applied and the estimated cost of the plan. Properties are also used by the wrappers to indicate which part of the requested query they are willing to perform. If, for example, a wrapper is not able to calculate one of the requested predicates then it will create a plan that leaves this predicate out. The properties of the returned plan show what part of the requested work is really done by the wrapper, and the Garlic middleware system will do the remaining work. *STrategy Alternative Rules*, for short *STARs*, are production rules of a grammar that are used to construct different execution plans. Each rule determines how a new plan is constructed from one or more partial plans, and there exist conditions for every STAR that guard its triggering. Generic STARs are fired during enumeration when a piece of work is found that must be done by a wrapper. These STARs model the capabilities of the

different wrappers and describe "what" the wrapper can execute, but not "how" it is done.

A complete execution plan is built bottom-up. For example, plans for *select* queries are created in three phases. During the first phase, every collection that is used in the query is accessed by using the generic access-STARs provided by each wrapper. In the second phase, the join order is determined. Dynamic programming is used for first creating two-way joins, then three-way joins and so forth until all needed joins are performed. For every join are three join methods considered. The *RepoJoin* is executed in the repository, the *NestedLoopJoin* is executed within the Garlic system and the *BindJoin* is a semi-join where the Garlic system send intermediate results from the outer objects to the wrapper, which uses these results for filtering the data it returns. In the third and last phase, the final query plans are constructed and Garlic includes all projections, selections and orderings that have not been achieved so far. The least cost plan out of all the created plans is then chosen for query execution.

Although the Garlic approach to external data source integration and query processing is very similar to our approach in AMOS *II*, there exist some differences:

- In Garlic are all attributes of an object retrieved at the same time before the joins take place. In AMOS *II* is only the core-cluster retrieved all at once, whereas the other predicates can be accessed after some joins took place. Doing this can eliminate communication overhead and lead to shorter query execution times.

- Garlic requests the wrapper to handle a set of predicates or to perform a join, whereas in AMOS *II*'s cost-based pick approach the wrappers group themselves those predicates together that they regard as a unit. This avoids unsuccessful work request, however, the mediator has no control over the grouping process.

- The decomposer push approach in AMOS *II* knows roughly which predicates can be handled by the repository. This a priory knowledge is used for a pre-selection in the grouping phase.

- AMOS *II* has the concept of directly mapped functions which avoids the invocation of the wrapper's query processor during plan generation. Moreover, for very simple data sources that can only return their entire content, even the core-cluster function can be a directly mapped

function and the grouper and the translator are not needed at all. In Garlic, on the other side, is the interface to the repository completely hidden to the middleware and the wrappers participate always in the query planning process.

- The Garlic system evaluates the cost for all possible join methods (nested-loop join, semi-join etc.). In AMOS *II*, on the other hand, the wrapper decides based on its capabilities which join method is performed. No other alternatives are considered. Though this can lead to suboptimal plans, the query optimization time is considerably reduced, especially when many join operations are needed. And the latter can be true even for relatively simple queries due to the excessive view creation needed for schema integration over heterogeneous data sources.

- In AMOS *II* does the mediator and not the wrapper create and store the OIDs. In fact, the OIDs are completely unknown to the wrapper and key values are used for object identification.

- Garlic has a centralized architecture, i.e. all data sources are wrapped in a single system. AMOS *II*, in contrast, employs a distributed architecture, i.e. there can exist many mediator servers communicating with each other.

5.2 The TSIMMIS System

The TSIMMIS (The Standford-IBM Manager of Multiple Information Sources) project uses a declarative approach for describing the capabilities of different information systems [10, 22, 17].

The *object exchange model (OEM)* is used as common data model and serves as basis for data integration. The basic entity in OEM in an object. Every object is *self-describing* containing the components *OID, label, type* and *value*. Therefore no external schema for storing meta type information is needed.

As mentioned above, and in contrast to the Garlic and AMOS *II* system, does the TSIMMIS system use a declarative approach for describing the capabilities of the data sources. More specifically, it uses templates specified in the *Relational Query Description Language (RQDL)* to represent the set of queries, called *source queries*, that can be processed by each data source. During query processing a query is divided into *subqueries*, which relate to

predicates in the AMOS *II* system. After view expansion an execution plan is created during the three consecutive phases *matching, sequencing* and *optimizing*.

The *matcher* finds for every subquery all the source queries processing it. This is done by expanding the templates which describe the set of executable source queries. Some pruning techniques are utilized to avoid unnecessary template expansions that cannot lead to matching source queries. The authors claim that in most practical situations the relevant source queries are produced in a polynomial time despite the inherent exponentiality of the problem [22]. The result of the matching phase is a set of source queries of which each one processes a subquery and has eventually some binding pattern requirements.

Next, the *sequencer* creates a set of feasible sequences, i.e. a list of source queries that together process all subqueries. Achieving this it is not just a question of which source queries should process which subquery, but also of determining their execution order to meet the binding requirements.

Finally, the *optimizer* picks the most efficient plan out of all feasible ones by calculating the cost of all plans.

The wrappers are only contacted in the adjacent execution phase. They receive the subquery and return the requested results. It is guaranteed that the data source can perform the subquery as the subquery was derived out of the templates describing all their executable queries.

5.3 Other Relevant Research Projects

Chaudhuri and Shim [1] discuss the topic of cost-based optimization of relational conjunctive queries in the presence of foreign functions. They introduce a simple declarative rule language to express the semantic information about these foreign functions. This declarative rule language is basically a set of rewrite rules that are used to generate alternative equivalent queries. $L(x, y) \rightarrow R(x, z)$ is the notation for a rewrite rule where $L(x, y)$ and $R(x, z)$ are conjunctive expressions and x, y and z are ordered sets of variables. Every variable in the set x is called an *universal variable*, which is a variable that occurs in both sides of the rewrite rule.

As an example consider a relational database with a table *Business* that stores the *ID, name* and the *type* of businesses, and a geographic data manager that provides the two foreign functions *Map(ID, loc)* and *Map_Rest(ID,*

loc). *Map* returns given the ID of a business its location and *Map_Rest* does the same for restaurants only. For obtaining the location of restaurants we can either take a join between *Business* and *Map* or can invoke *Map_Rest*. Ths can be expressed in the following rewrite rule[1]:

$$Business(ID, name, \text{'Restaurant'}), Map(ID, loc) \leftrightarrow Map\_Rest(ID, loc)$$

Observe that it is not always correct to apply this rewrite rule. For example when there exists another table *Owner* in the relational database that stores the name of a business and the name of its owner and we want to retrieve the location of all restaurants whose owner is called 'Bob', we can state the following query Q:

$$\begin{aligned} Q(loc) \ :- \ & Business(ID, name, \text{'Restaurant'}), \ Map(ID, loc), \\ & Owner(name, \text{'Bob'}) \end{aligned}$$

However, applying the above rewrite rule leads to the following query Q' that is not semantically equivalent:

$$Q'(loc) \ :- \ Map\_Rest(ID, loc), \ Owner(name, \text{'Bob'})$$

It turns out that the semantics of the rewrite rules guarantee that the left-hand and the right-hand sides of the rewrite rules are equivalent over universal variables only. Thus, a rewrite rule $L(x, y) \rightarrow R(x, z)$ can only be applied in a query Q when Q has the form

$$Q(u) :- \ L(v, w), G(t)$$

where $G(t)$ represents the conjunction of the rest of the predicates in query Q, and where the set of variables w is disjoint from the set u as well as the set t. An applicable occurrence of a rewrite rule is called a *sound occurrence*.

During query optimization first the *closure* of a query, i.e. the set of all equivalent queries, is generated by iteratively applying all sound occurrences of all rewrite rules. Next, the best execution plan for every query is computed and the cheapest of all these plans will get executed. A smart algorithm using the dynamic programming approach is presented that eliminates the recalculation of common subqueries in different plans. Note that the closure

[1]Rewrite rules and queries are stated in domain calculus expressions as in non-recursive Datalog [29].

of a query might not be finite for some rewrite rules, due to the existence of endless derivations that do not terminate. This must also be detected by the query optimizer.

This proposed approach might be applicable for our B^+-tree example by stating a couple of rewrite rules for transforming the simple B^+-tree scan into a range query. However, it doesn't work for parameterized foreign functions like SQL that need a string expression as input. It is not possible to create the appropriate string expression with these simple rewrite rules expressed in a declarative language.

Related to this approach is the Starburst project [23]. However, Starburst uses a procedural language for expressing the semantic knowledge, which enables the description of arbitrary complex rewrite rules. Moreover, the rewrite rule language in Starburst is not only used to express semantic knowledge, but also to express the rules for query transformation during the optimization phase. Another difference to the work of Chaudhuri and Shim [1] is that Starburst uses heuristics for applying a rewrite rule and do not follow an exhaustive search approach.

Chapter 6

Summary and Future Work

6.1 Summary and Conclusions

AMOS *II* is an extensible object-oriented mediator database system. It is designed to integrate the data stored in a number of distributed heterogeneous data sources. In this thesis I presented the extensions I have made to the AMOS *II* system. These extensions concern three main subjects. Firstly, I created a mechanism for an easy transformation of a foreign data model into the AMOS *II* data model. Secondly, I defined the role of a wrapper and described how to deal with limited query capabilities. Furthermore, an interface for extending the core mediator system with new wrappers was defined. And thirdly, I proposed and evaluated two approaches to query processing over external data sources. For demonstration purposes a B^+-tree package was used. I will now briefly summarize each of these contributions.

For *data model transformation* the concepts of mapped functions, mapped types, mapped objects and directly and indirectly mapped functions were introduced. Furthermore is it possible to clusterize the properties of a mapped type into a core and non-core clusters. Key information is needed for creating unique OIDs in the mediator system. These concepts allow both the representation of data stored in an external data source as well as the introduction of new operators in the AMOS *II* mediator system.

The *role of wrappers* can be divided in three different tasks (see Figure 3.1). The meta-data manager supplies the needed information for inte-

grating external data and operators. The query participator participates in query optimization and encapsulates the knowledge about the repository's query capabilities. It also provides cost functions. And, finally, for providing an interface to the external data sources the concept of foreign functions is utilized. These foreign functions are either called directly by the user or they are directly or indirectly mapped to some other functions.

For *query processing* over external data sources two different approaches were presented and evaluated. The decomposer push approach utilizes the AMOS *II* decomposer to send subqueries to the wrappers for execution in the external data source. For doing that the decomposer has to have some knowledge about the query capabilities of the repositories. As it is nearly impossible to declare all the minute details and restrictions of every data source, the wrappers declare only which multi implementation functions (MIFs) they might be able to handle. All predicates of a subquery that cannot be process in the repository are post-processed in the mediator system. In the cost-based pick approach, on the contrary, are the subqueries formed by the wrappers during cost-based optimization. The wrappers pick those predicates they are willing to perform. It appears that the cost-based pick approach is well suited for data sources with very limited query capabilities that do not have their own query optimizer. The decomposer push approach, however, seems to be the better suited for fully fledged data sources like relational or object-oriented databases. This means that the decomposer is used for decomposing queries over "real" databases whereas the cost-based pick approach is used for extending the AMOS *II* system with data sources that do not have their own query optimizer.

6.2 Future Work

AMOS *II* is a very stable research system and very well suited for further extensions. There exist a couple of interesting questions in the area of this work that have to be further investigated. Some of the main issues are the following:

- The cost-based pick approach is only implemented for ranksort so far. As shown in Chapter 4 should this implementation be reviewed and optimized to reduce the cost-based optimization time. Furthermore, the grouping and translation phases have to be introduced in the other two algorithms, i.e. random sort and the dynamic programming approach.

- The proposed approach for data model transformation should be tested for other repositories like web sources, file managers and very specialized applications. An interesting example for a complex indirectly mapped function is the `grep` command. `grep` is capable of searching files for single words or for whole patterns expressed as regular expressions. Capturing all these capabilities in an indirectly mapped function would be both, an useful extension to the AMOS *II* system, as well as a challenging task well suited for testing the concept of indirectly mapped functions.
- In this thesis the focus was kept on retrieving data from external data sources. There exists a need for introducing generic functions for inserting and updating data in the repositories.
- Caching techniques for materializing intermediate results in the AMOS *II* system should be made available for wrappers. This can lead to huge performance improvements in query execution times.

Appendix A

Abbreviations

COID	create OID (object identifier)
DBMS	database management system
DcT	decomposition tree
DDL	data definition language
DML	data manipulation language
DST	data source type
MIF	multiple implementations functions
ODBC	open database connectivity
OEM	object exchange model
OID	object identifier
OO	object oriented
PPL	post processing list
SAE	ship and execute
SF	subquery function
SIF	single implementation function
SQL	structured query language

References

[1] S. Chaudhuri, K. Shim, "Query Optimization in Presence of Foreign Functions", *Proceedings of the 19th Conference on Very Large Data Bases (VLDB'93)*, pp. 529-542, Dublin, Ireland, 1993.

[2] M. Carey, L. Haas, P. Schwarz, M. Arya, W. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. Williams, E. Wimmers, "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach", *Proc. IEEE RIDE-DOM*, Taipei, Taiwan, March 1995.

[3] T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[4] R. Elmasri, S. Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., second edition, 1994.

[5] G. Fahl, T. Risch, M. Sköld, "AMOS – An Architecture for Active Mediators", *Proceedings of Next Generation Information Technologies and Systems (NGITS'93)*, Hafai, Israel, June 1993.

[6] G. Fahl, T. Risch, "Query processing over object views of relational data", *The VLDB Journal*, 6, pp. 261-281, November 1997.

[7] D. Fishman, D. Beech, J. Annevelink, E. Chow, T. Conners, J. Davis, W. Hasan, C. Hoch, W. Ken t, S. Leichner, P. Lyngbaek, B. Mahbod, M. Neimat, T. Risch, M. Shan, W. Wilkinson, "Overview of the IRIS DBMS", in W. Kim, F. Lochovsky, eds., *Object-Oriented Concepts, Databases and Applications*, Readiong, MA: Addison-Wesley, 1989.

[8] S. Flodin, T. Risch, "Processing Object-Oriented Queries with Invertible Late Bound Functions", *Proceedings of the 21st Conference on Very Large Databases (VLDB'95)*, Zürich, Switzerland, 1995.

[9] S. Flodin, V. Josifovski, T. Risch, M. Sköld, M. Werner, "AMOS*II* User's Guide", available at *http://www.ida.liu.se/∼edslab*, 1999.

[10] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom, "The TSIMMIS Approach to Mediation: Data Models and Languages", *Journal of Intelligent Information Systems*, Vol. 8, No. 2, pp. 117-132, 1997.

[11] L. Haas, D. Kossmann, E. Wimmers, J. Yang, "Optimizing Queries Across Diverse Data Sources", *Proc. of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, Athens, Greece, August 1997.

[12] L. Haas, R. Miller, B. Niswonger, M. Roth, P. Schwarz, E. Wimmers, "Transforming Heterogeneous Data with Database Middleware: Beyond Integration", *Bulletin of the Technical Committee on Data Engineering*, IEEE Computer Society, Vol. 22, No. 1, March 1999.

[13] Y. E. Ioannidis, Y. C. Kang, "Randomized Algorithms for Optimizing Large Join Queries", *Proceedings of the 1990 ACM-SIGMOD Conference on the Management of Data*, pp. 312-321, Atlantic City, NJ, May 1990.

[14] A. Josifovski, T. Risch, "Functional Query Optimization over Object-Oriented Views for Data Integration", *Journal of Intelligent Information Systems (JIIS)*, Vol. 12, pp. 165-190, 1999.

[15] A. Josifovski, "Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration", Dissertation No. 582, Laboratory for Engineering Databases, Linköping University, Sweden, 1999.

[16] A. Josifovski, T. Katchaounov, T. Risch, "Optimizing Queries in Distributed and Composable Mediators", *Proc. of the 4th Conference on Cooperative Information Systems (CoopIS'99)*, Edinburgh, Scotland, September 1999.

[17] C. Li, R. Yerneni, V. Vassalos. H. Garcia-Molina, Y. Papakonstantinou, J. Ullmann, M. Valiveti, "Capability Based Mediation in TSIMMIS", *Proceedings SIGMOID Conference*, 1998.

[18] W. Litwin, T. Risch, "Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, pp. 517-528, December 1992.

[19] G. Lohman, "Grammar-like functional rules for representing query optimization alternatives", *Proc. of the ACM SIGMOD Conf. on Management of Data*, pp. 377-388, Portland, OR, USA, May 1989.

[20] P. Lyngbaek et al., "OSQL: A Language for Object Databases", *Technical Report, HP Labs, HPL-DTD-91-4*, 1991.

[21] Adrian Mardlin, "ISAM Manager, B+Tree/ISAM System For C++", *Nildram Software*, 1994.

[22] Y. Papakonstantinou et al., "Capabilities-Based Query Rewriting in Mediator Systems", *Journal of Distributed and Parallel Databases*, Vol. 6, No. 1, pp. 73-110, 1998.

[23] H. Pirahesh, J. M. Hellerstein, W. Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst", *Proceedings of the 1992 ACM-SIGMOD Conference on the Management of Data*, pp. 39-48, San Diego, CA, May 1992.

[24] M. Roth, P. Schwarz, "Don't Scrap It, Wrap It.", *23rd Int. Conf. on Very Large Databases (VLDB97)*, pp. 266-275, Athens, Greece, August 1997.

[25] M. Roth, P. Schwarz, "A Wrapper Architecture for Legacy Data Sources", *IBM Technical Report RJ10077*, 1997.

[26] D. Shipman, "The Functional Data Model and the Data Language DAPLEX", *ACM Transactions on Database Systems*, Vol. 6, No. 1, ACM Press, 1981.

[27] M. Sköld, T. Risch, "Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions", *12th International Conference on Data Engineering (ICDE'96)*, IEEE, New Orleans, Louisiana, Feb. 1996.

[28] A. Tomasic, L. Raschid, P. Valduriez, "Scaling Access to Heterogenous Data Sources with Disco", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 5, pp. 808-823, Sept./Oct. 1998.

[29] J. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. I & II., Computer Scinece Press, New York, NY, 1988.

[30] V. Vassalos, Y. Papakonstantinou, "Describing and Using Query Capabilities of Heterogeneous Sources", *Proceedings of the 23rd VLDB Conference*, pp. 256-265, Athens, Greece, August 1997.

[31] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer* Vol. 25, No. 3, pp. 38-49, 1992.