Master Thesis

Cost-based query optimization in iMeMex

Author(s):
Blunschi, Lukas

Publication Date:
2007

Permanent Link:
https://doi.org/10.3929/ethz-a-005729133

Rights / License:
In Copyright - Non-Commercial Use Permitted
Cost-based Query Optimization in iMeMex

Lukas Blunschi

Master’s Thesis

Databases and Information Systems Group
Department of Computer Science
ETH Zurich

http://www.dbis.ethz.ch/

November 2006 - May 2007

Supervised by:
Dr. Jens-Peter Dittrich
Marcos Antonio Vaz Salles
Shant Kirakos Karakashian
Abstract

Personal Information Management (PIM) refers to managing all data related to a single individual. In more detail, PIM is: integrating highly heterogeneous data sources, runtime environments which may change dramatically, a query language which is powerful, but still simple to use, semantic integration of all data pertaining to a user, sharing parts of my data with friends, query processing in such a system. PIM is becoming more and more important, but still there is no single system available which is able to handle all challenges. iMeMex [15] is among the first systems aiming to support all of that functionality.

In this thesis we enrich iMeMex so that it can adapt to adding or removing data sources and index structures and look at how to process and optimize queries in iMeMex.

To solve the former we use OSGi as the basic runtime platform and separate our system into basic building blocks, aka bundles. By doing so not only functionality to access different data sources, but also all indexes, materialized views and replicas become hot-pluggable. As an effect our query processor does not have to be changed when new system functionality is added or removed.

To work on the latter we introduce a query processing model that is completely based on query pushdown. Here we profit from having an adaptable system architecture enabling us to treat data sources, indexes and materialized views as low-level query planners. Each of these query planners may decide by itself how much of a query plan it wants to handle and also what cost model it uses. We show how the returned physiological plans can be used in extensible cost-based query optimization. To show the feasibility of these techniques, several use cases are studied in detail.

Keywords: OSGi, heterogeneous data sources, query pushdown, query planning, cost-based optimization, multiple cost models
Contents

1 Introduction 15

2 Background 17
  2.1 iDM 17
     2.1.1 iDM Features 17
     2.1.2 Fragments 18
  2.2 Logical Algebra 18
     2.2.1 Input Operators 18
     2.2.2 Processing Operators 18
     2.2.3 Planning Operators 19
  2.3 iQL 20

3 System Architecture 21
  3.1 Overview 21
     3.1.1 Layers 21
  3.2 Data Source Plugins 23
     3.2.1 Current Data Source Plugins 23
  3.3 Indexes and Replicas 23
     3.3.1 Mapping Set 23
     3.3.2 Indexes 25
     3.3.3 Replicas 25
  3.4 Materialized View Manager 26
     3.4.1 Query Pushdown Containment Test 26
     3.4.2 Query Pushdown Tree Comparison 28
     3.4.3 Materialized View Serialization 29
  3.5 Service Oriented Architecture 30
     3.5.1 OSGi 30
     3.5.2 Use Case 1: Mobile Phone 34
     3.5.3 Use Case 2: Data Warehouse 34
  3.6 Query Flow 37

4 Logical Query Planning 39
  4.1 Previous Query Processing Model 39
     4.1.1 Limitations 39
     4.1.2 Moving towards Logical Nodes 41
  4.2 Query Push Down 42
  4.3 Logical Choose 42
     4.3.1 Logical Choose Specification 42
     4.3.2 Logical Choose Usage 43
     4.3.3 Why a Logical Choose? 43
     4.3.4 Resolved Logical Query Plans 44
     4.3.5 Unneeded Logical Query Plans 44
## CONTENTS

A  **Joins in iMeMex**  
   A.1  Joins in General  
   A.2  Joins in iMeMex  

91

B  **Alternative Merge Simplifiers**  
   B.1  Unneeded Unresolved Alternative Remover  
   B.2  Trivial Choose Remover  

93

C  **Fetch Pushdown and Pull-Up**  
   C.1  Fetch Pushdown  
   C.2  Fetch Pull-Up  

97
# List of Figures

2.1 Resource view and resource view graph .................................................. 17  
2.2 Logical input operators ........................................................................... 18  
2.3 Start views example ................................................................................ 18  
2.4 Logical processing operators ................................................................... 19  
2.5 Logical operators used during query planning ......................................... 20  
3.1 System architecture overview ................................................................. 22  
3.2 Semantically equivalent query plans ......................................................... 28  
3.3 Superset match ....................................................................................... 28  
3.4 Materialized view storage structure .......................................................... 29  
3.5 Bundle dependency graph ....................................................................... 31  
3.6 iMeMex system architecture .................................................................... 32  
3.7 Resource view implementation problem .................................................. 33  
3.8 Resource view mappers .......................................................................... 33  
3.9 Possible deployment scenario for a mobile phone .................................... 35  
3.10 Possible deployment scenario for a data warehouse ................................ 36  
3.11 Query flow ............................................................................................. 37  
4.1 Previous query processing model ............................................................... 40  
4.2 Selection on name and tuple component .................................................. 40  
4.3 Query processing model moving towards logical nodes ............................ 41  
4.4 Possible replacement of pushdown ............................................................ 41  
4.5 Query processing model based on full pushdown ...................................... 42  
4.6 Graphical specification of logical choose ................................................ 43  
4.7 Logical choose usage .............................................................................. 43  
4.8 Unresolved and resolved logical query plans .......................................... 44  
4.9 Unneeded logical query plan ................................................................... 46  
4.10 Alternative merge example - input ........................................................... 46  
4.11 Alternative merge example - after one recursion ..................................... 47  
4.12 Alternative merge example - after merge ............................................... 48  
4.13 Alternative merge example - simplified ................................................ 48  
4.14 Graphical specification of logical fetch .................................................. 49  
4.15 Intuitive logical fetch placement ............................................................. 50  
4.16 Unnests produce new results ................................................................. 51  
4.17 Fragment pushdown decision tree ......................................................... 51  
4.18 Example fetch pushdown ....................................................................... 52  
4.19 Example fetch pull-up .......................................................................... 52  
4.20 Fragment pull-up decision tree ............................................................... 53  
4.21 Pull-up example distinct ....................................................................... 53  
4.22 Pull-up example union .......................................................................... 54  
4.23 Pull-up join example query plans ............................................................ 55  
4.24 Pull-up minus example ......................................................................... 56
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.25</td>
<td>Pull-up combination of intersect and union example</td>
</tr>
<tr>
<td>4.26</td>
<td>Pull-up combination of filter and union example</td>
</tr>
<tr>
<td>4.27</td>
<td>Graphical specification of logical provide</td>
</tr>
<tr>
<td>4.28</td>
<td>Dead fragment detection - intersect</td>
</tr>
<tr>
<td>4.29</td>
<td>Dead fragment detection - union</td>
</tr>
<tr>
<td>4.30</td>
<td>Optimal, but ambiguous logical query plan</td>
</tr>
<tr>
<td>4.31</td>
<td>Graphical specification of a black box</td>
</tr>
<tr>
<td>4.32</td>
<td>Generalization of accessors</td>
</tr>
<tr>
<td>4.33</td>
<td>Physiological query plan and inner life of black boxes</td>
</tr>
<tr>
<td>5.1</td>
<td>Multiple cost models</td>
</tr>
<tr>
<td>5.2</td>
<td>Stacked query planners</td>
</tr>
<tr>
<td>5.3</td>
<td>Working with execution time measuring operators</td>
</tr>
<tr>
<td>5.4</td>
<td>Peak recognition of look back average exchange rate update algorithm</td>
</tr>
<tr>
<td>5.5</td>
<td>Interesting order example</td>
</tr>
<tr>
<td>5.6</td>
<td>Join order example</td>
</tr>
<tr>
<td>5.7</td>
<td>Choose pull-up</td>
</tr>
<tr>
<td>6.1</td>
<td>Query plans used for experiments</td>
</tr>
<tr>
<td>6.2</td>
<td>Experiment 1</td>
</tr>
<tr>
<td>6.3</td>
<td>Experiment 2</td>
</tr>
<tr>
<td>6.4</td>
<td>Experiment 3</td>
</tr>
<tr>
<td>6.5</td>
<td>Experiment 4</td>
</tr>
<tr>
<td>7.1</td>
<td>Structurally changed query plan example</td>
</tr>
<tr>
<td>A.1</td>
<td>Join types</td>
</tr>
<tr>
<td>A.2</td>
<td>Natural join example</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Current data source plugins .................................................. 23
4.1 Logical node type categorization .............................................. 53
4.2 Logical node type categorization - dead fragment detection .......... 59
5.1 Comparison of exchange rate update algorithms ....................... 70
5.2 Cost model of eager choose planner ....................................... 72
### Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>MappingSet interface</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>LogicalQueryPlanner interface</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Index interface</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Replica interface</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>MaterializedViewManager interface</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Alternative Merge</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Cost class</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Exchange officer interface</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>Black box planner</td>
<td>72</td>
</tr>
<tr>
<td>5.4</td>
<td>Eager choose planner</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>OIDArray Class</td>
<td>78</td>
</tr>
<tr>
<td>B.1</td>
<td>Unneeded unresolved alternative remover</td>
<td>94</td>
</tr>
<tr>
<td>B.2</td>
<td>Trivial choose remover</td>
<td>95</td>
</tr>
<tr>
<td>C.1</td>
<td>Fetch pushdown - not fetch type</td>
<td>97</td>
</tr>
<tr>
<td>C.2</td>
<td>Fetch pushdown - fetch type</td>
<td>98</td>
</tr>
<tr>
<td>C.3</td>
<td>Fetch pull-up - part 1</td>
<td>99</td>
</tr>
<tr>
<td>C.4</td>
<td>Fetch pull-up - part 2</td>
<td>100</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Today every person has to manage a growing amount of personal data. Personal data is composed of office documents, calendar data, emails, RSS feeds, pictures, music or even videos. This data is distributed over a huge range of storage devices like desktop computers, mobile phones, email servers, relational databases, web space, subversion repositories, etc. *Personal Information Management* (PIM) covers all activities related to manage personal data. Its primary goal is to offer convenient access to and manipulation of all data belonging to a user.

Recently a new abstraction for information management, *dataspaces*, was proposed [8]. A dataspaces in general should contain all data relevant for a given entity and model rich relationships between different data portions. Furthermore, several services on top of a dataspaces should be offered by a DataSpace Support Platform (DSSP).

A personal dataspaces is a specialization of a general dataspaces. It comprises all data belonging to a single person. The focus of iMeMex [15] and this Master’s thesis is on such dataspaces. A DSSP specialized for personal information management is called Personal DSSP (PDSSP) or *Personal Dataspaces Management Systems* (PDSMS).

iMeMex belongs to this new breed of information systems which act as a middlelayer between the raw storage and the higher-level applications. The purpose is to combine personal data from a variety of data sources, to integrate them in an incremental “pay-as-you-go” fashion, to provide intelligent querying capabilities on the data and last but not least to share the data with other interested individuals.

Personal data can be stored in a structured way like tables, e.g., contact information, but most of our personal data is kept in a wild mix of heterogeneous storage devices and data formats [6]. This has called for a new data model [5] and a new search and query language [9]. Which both have been implemented prior to this work. We show related work and background information in Chapter 2.

For iMeMex one could think of many different deployment scenarios having completely different system requirements. iMeMex should not only run on a mobile phone to manage addresses and contact information, but also on servers to care for all the data of a workgroup. In the former example no replicas and only one index might be enough, where in the latter case replicas might be essential and many indexes could be used. Clearly a system architecture is needed which can be configured to work in various scenarios. We use OSGi [20] to add or remove system functionality at runtime. It is also inevitable to have a query processor that considers the system functionality which is available at the moment. For example when we add a new type of index, our query processor should automatically start to use this index. Our system architecture is presented in Chapter 3.

iMeMex supports a variety of data sources and index structures. Some with almost no querying capabilities (e.g. filesystem), others with limited query support (e.g. full-text index) and even others with very strong query support (e.g. relational database management system). To make use of as much as possible of the provided querying capabilities, we implemented a query processing model based on pushing down a query to lower query planners. The data sources are lower query planners, but also all indexes and the materialized
view manager.

These lower query planners choose how much of the given query plan they can or want to compute, and consequently replace these parts with black boxes. What we receive from these query planner are physiological query plans, containing logical parts which are not yet planned and physical parts that are already planned. Our query processing model along with physiological query plans is introduced in Chapter 4.

Lower query planners are very different in their nature and they may not be able to use a common cost model among all of them. We therefore introduce a mechanism to translate between different cost models. As an effect each query planner can use a cost model that is most appropriate for it. Multiple cost models are explained in Chapter 5.
Chapter 2

Background

This chapter provides background information and shows related work which is needed to understand the remainder of this thesis.

2.1 iDM

The data model used in iMeMEx is called iDM, iMeMEx Data Model [5]. Its basic entity is a Resource View which has the following components:

- **name** ($\eta$): Name of this resource view as a string
- **tuple** ($\tau$): List of attribute name/value pairs
- **content** ($\chi$): Binary stream of data along with a Mime-Type
- **group** ($\gamma$): References to other resource views either as an unordered, (in)finite set or an ordered, (in)finite sequence.

iDM represents all data as a graph. The nodes of this graph are resource views, the connections between the nodes are defined by the group component. See Figure 2.1.

2.1.1 iDM Features

The fact that iDM employs a graph model makes it clear that iDM is able to handle structured, semi-structured as well as unstructured data. It is therefore well applicable to the heterogeneous data mix found in today’s personal data[6].

Besides using a graph model, iDM is also able to handle intensional data and infinite data streams. Intensional data can be treated, because the components of a resource view are lazily computed. Infinite data streams are enabled by using a stream as content and a possibly infinite group component.

![Figure 2.1: Resource view and resource view graph](image)
Finally it is worth noting, that a resource view just represents a piece of information which is not directly related to a physical representation of that data. Thus physical entities like files and folders are represented in the same way as a paragraph in a \LaTeX{} document or an element in a XML file.

2.1.2 Fragments

A resource view is naturally split into components. These components themself may be further dividable into fragments. A fragment can either be a complete component, e.g. the name component is a fragment, or a part of a component, like a single tuple attribute. It is necessary to split the tuple component into its attributes, because otherwise each index and replica that wants to index or store parts of the tuple component would have to index or store all tuple attributes. A materialized view, for example, may want to store a single attribute.

Currently only the tuple component is further divided, but in the future it could be necessary to also split the content component into fragments.

2.2 Logical Algebra

This section explains the logical algebra used by iMeMex. To do so, all available logical operators are shown and explained in a few words. This logical algebra is defined on top of the data model described in Section 2.1.

2.2.1 Input Operators

Figure 2.2 shows the logical operators which may serve as input operators, that is they have no children. \textit{Start Views} are special nodes in a resource view graph that depict nodes from which all other nodes can be reached (via deep unnest, see 2.2.2). Figure 2.3 shows two situations: On the left, a single \textit{Start View} is enough to reach all resource views, on the right two \textit{Start Views} are required. \textit{All} represent the complete resource view graph and a \textit{Black Box} produces the results of a subquery. Black boxes are explained in detail in Section 4.5. A formal and semantic specification of the first two operators can be found in [9].

2.2.2 Processing Operators

The operators shown in Figure 2.4 present logical operators which are used to formulate a query. These operators may appear in a logical query plan after the query string was parsed.

As for the input operators, a formal and semantic specification of all these operators can be found in [9]. For completeness, a summary of their meaning is listed here:
2.2 Logical Algebra

• **Distinct** removes duplicates from the output
• **Filter** filters the input according to a given predicate
• **Intersect** outputs only resource views which appear in both inputs
• **Union** combines the two inputs without removing duplicates
• **Minus** subtracts the right input from the left
• **Shallow Unnest** returns these resource views from the resource view graph which are direct children of the resource views received in the input
• **Deep Unnest** returns all resource views that can be reached by any path starting from the resource views given in the input. Implementations of Deep Unnest must take care to avoid cycles.
• **Project** hides specified fragments from the output
• **Order By** orders the output (ascending or descending) according to a list of fragments
• **Group By** groups the output according to a list of fragments
• **Top-K** produces only the top n results given some weight predicate
• **θ-join** joins inputs by specified θ-predicate
• **Left Outer Join** joins resource views from the left input with fragments provided by the right input
• **Right Outer Join** joins resource views from the right input with fragments provided by the left input
• **Full Outer Join** joins left and right input adding all resource views which neither join to the left nor the right

### 2.2.3 Planning Operators

The last group of operators are these which are used during query planning or which are inserted by query planning. They are shown in Figure 2.5.

Since these operators were introduced during this work, they all are explained in more detail in their corresponding section in Chapter 4.
2.3 iQL

The query language used by iMeMex is called iQL, iMeMex Query Language [9]. It aims to combine structural queries, e.g. “everything related to ETH”, with intuitive content search like “everything containing iMeMex”. It is similar in spirit to XPath [24] and NEXI [23].

It can be best understood by looking at a few examples:

Example (1):  
```
//eth/* iMeMex
```

*Everything which is related to ETH and contains the term iMeMex.*

Example (2):  
```
//*[age>25]/*
```

*Everything which is directly related to everything having age larger than 25.*

Example (3):  
```
join(/students/* as A, /departements/* as B, A.dept = B.id)
```

*Join all students with all departements.*
Chapter 3

System Architecture

A well defined system architecture builds the basis for query optimization and query execution. iMeMex, being a Personal DataSpace Management System (PDSMS), must be able to adapt to a wide variety of deployment scenarios. On the one hand, it should be possible to run iMeMex on devices which have very limited computing power, like mobile phones, where neither indexes nor replicas can be used. On the other hand, iMeMex will be run on desktop computers and workgroup servers which have strong computing power and fast indexes and large replicas may be required.

As iMeMex is supposed to run constantly in the background as a service, it should also be possible to extend its functionality at runtime. Adding new data sources might require to install an extension to support the protocol of the new data source. New system functionality, like support for a new data format, might become available and could also be added while the system is running.

During the course of this thesis we have changed the existing system architecture to support different deployment scenarios as well as to provide a way to adapt the system at runtime. We also changed the query processing model to allow to push down a query to heterogeneous and distributed query planners. We will present this new query processing model in more detail in Chapter 4.

In this chapter we focus on these aspects of the system architecture which are important for query planning, query optimization, and query execution. We give a high-level overview of the system in Section 3.1. Section 3.2 introduces available data source plugins. After that Section 3.3 looks at the indexes and replicas found in iMeMex and Section 3.4 presents the Materialized View Manager. Section 3.5 explains the reason to use a service oriented architecture (SOA) and how we are using OSGi [20] for that purpose. This section also shows the data flow in the lower layers of iMeMex. We conclude this chapter by presenting the query flow in Section 3.6.

3.1 Overview

iMeMex targets to be positioned between native data sources and high-level applications - acting as a middleware layer. iMeMex itself is also designed using a layered system architecture. Figure 3.1 shows an overview of the current system architecture. Notice that iMeMex, opposed to a relational database management system (DBMS), does not take full control over the data sources. High-level applications may still bypass iMeMex to access the data. See [4] for more details.

3.1.1 Layers

In this subsection we will explain each of the major layers found in iMeMex and shown in Figure 3.1.

Data Source Layer

A data source is accessed through data source plugins. Each data source plugin (DSP) provides an implementation of a different access protocol, e.g. HTTP, IMAP, SMB. The output of a data source plugin are
resource views. Before they leave the data source layer, another set of extensions processes their content components. These extensions are called Content to iDM Converters (CC). They try to recognize the format of the content and convert it to a more fine granular resource view graph. A \LaTeX\ file for example could be converted to a graph which consists of one resource view for each chapter and section.

Above the data source layer we have a physically independent view on the data, i.e. all data source specific objects are translated to a common logical data model (iDM), see Section 2.1.

### Indexes and Replicas

If a client decides to index or replicate parts of the data in his dataspace using iMeMex, then the resource view manager transfers that data from the data sources to the indexes and replicas. This component is responsible for providing index and replica implementations. In this layer we also find the Materialized View Manager (MVM) which is responsible to store, read, and apply materialized views. A materialized view is a precomputed result to a query.

### Resource View Manager

The main purpose of the Resource View Manager (RVM) is to provide transparent access to data sources whether they are indexed and replicated or not. To perform this task, it uses a catalog and assigns a resource view ID to each resource view. This ID is a unique object identifier (OID) as used in SHORE [3].

### Query Processor

The Query Processor processes iQL queries which are received from the application layer. First, the iQL string is parsed into a logical query plan. This query plan is then given to the Resource View Manager that decides whether the data source plugins have to be queried or whether the query could be answered using the local indexes and replicas. Either the data sources or the indexes and replicas will then process the query and the result is returned to the query processor. The query processor finishes the query planning and returns a physical execution plan to the client. More details about query planning in iMeMex can be found in Chapter 4 and 5.
3.2 Data Source Plugins

All data in iMeMex is accessed through data source plugins. They can be used in two major ways, which are both important for query processing:

- **Data shipping**: To ship the data we iterate over all physical data items in a data source. The data source plugin converts from the native data model to a resource view graph. This access method is used to index or replicate a data source. We call this approach *warehousing*.

- **Query shipping**: This allows to push down a query to the data source. The data source is responsible to handle the query. It is used when a data source is neither indexed nor replicated. We call this approach *mediation*.

To add a data source to the system, it has to be registered. At this moment we also specify whether a data source is warehoused or if the data source is mediated.

A data source plugin may implement one of either data shipping or query shipping or both access methods. A Google plugin for example would certainly not support data shipping access, whereas a file system plugin may implement data shipping only. A data source can only be warehoused if the data source plugin used to access the data source supports data shipping.

As mentioned in the Introduction, a data source plugin is also a query planner. Now, after we have defined the possible access methods, it is clear that only data source plugins which support query shipping can be used as query planners.

### 3.2.1 Current Data Source Plugins

What follows is a short summary of the existing data source plugins at the time of this writing. Currently all data source plugins support data shipping and most also support to pushdown selections.

<table>
<thead>
<tr>
<th>Data source plugin</th>
<th>Protocols</th>
<th>Data shipping</th>
<th>Query shipping</th>
</tr>
</thead>
<tbody>
<tr>
<td>File System</td>
<td>file, smb</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>IMAP</td>
<td>imap</td>
<td>yes</td>
<td>selection</td>
</tr>
<tr>
<td>JDBC</td>
<td>jdbc</td>
<td>yes</td>
<td>selection, intersect, union, joins</td>
</tr>
<tr>
<td>WebXML</td>
<td>http</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Remote iMeMex</td>
<td>imemex</td>
<td>yes</td>
<td>yes, everything</td>
</tr>
</tbody>
</table>

Table 3.1: Current data source plugins

3.3 Indexes and Replicas

The main purpose of indexes and replicas is to speed up query processing. In this section we show which indexes and replicas are available in iMeMex and also present the Index and Replica interfaces.

#### 3.3.1 Mapping Set

A mapping set generalizes indexes and replicas. It allows to insert or remove mappings from OID to indexed or replicated data.

A mapping set does not need to store all components or fragments of a resource view (see Section 2.1.2 for how a resource view may be shredded). It is therefore important to know which parts of a resource view are stored in the mapping set. The method `getComponentsAvailableFlags()` allows to ask which components are provided by the mapping set. The Mapping Set interface will be changed in the near future to support fragments instead of components.

All other methods of the `MappingSet` interface are shown in Listing 3.1.
/**
 * A mapping set is the basis for indexes and replicas.
 */

public interface MappingSet {

    /* Get availability flags for all components. */
    int getComponentsAvailableFlags();

    /* Insert or update a mapping. */
    boolean insert(IndexView indexView);

    /* Insert or update entries of the given PullOperator into this mapping. */
    boolean insertAll(PullOperator<IndexView> entries);

    /* Remove a mapping */
    void remove(long oid);

    /* Remove all given mappings. */
    void removeAll(PullOperator<Long> oids);

    /* Return the number of entries in this mapping. */
    long size();

    /* Clear this mapping. */
    void clear();

    /* Close this mapping. */
    void close();

    /* Rebuild and optimize this mapping. */
    void rebuild(boolean online) throws OnlineRebuildNotSupportedException;

    /* Flush memory-resident data structures to disk. */
    void flush();

    /* Get the number of elements which are buffered before being stored persistently by the mapping. */
    int getBufferSize();
}

Listing 3.1: MappingSet interface
3.3 Indexes and Replicas

3.3.2 Indexes

In general an index can process a query over a certain set of the data. Commonly, indexes perform selections. In iMeMex an index may also process more complex query plans like path queries, intersects and unions, or joins. We currently do not have a join index, but integrating one into the current system architecture would be straightforward.

```java
/**
 * The logical query planner interface allows to push down a query to another
 * logical query planner.
 * <p>
 * It is implemented by every component which wants to work on logical query
 * processing.
 */
public interface LogicalQueryPlanner {

  /**
   * Process the given logical query tree.
   * @param logicalQueryTree given logical query tree.
   * @return logical query tree, which may have black boxes in it.
   */
  LogicalNode query(LogicalNode logicalQueryTree);
}
```

Listing 3.2: LogicalQueryPlanner interface

Each index implements the LogicalQueryPlanner interface (see Listing 3.2). This interface is not only used for indexes, but for each query planner found in iMeMex. Examples are: The resource view manager, the data source proxy or the materialized view manager. The input to the query() method is a logical query plan, its output is a physiological query plan. That is a plan where subplans may be physically planned. Details about physiological query plans are presented in Section 4.5. Besides implementing the logical query planner interface, each index also implements the MappingSet interface presented in Listing 3.1. This interface allows to add or remove entries from the index. The Index interface (Listing 3.3) combines the MappingSet and the LogicalQueryPlanner interface.

```java
/**
 * An index is a combination of a logical query planner and a mapping set.
 */
public interface Index extends LogicalQueryPlanner, MappingSet {
}
```

Listing 3.3: Index interface

3.3.3 Replicas

Similar to indexes, replicas also extend the MappingSet interface. This allows to add or remove entries in the replica. The idea behind a replica is to have a local copy of some data. A replica could also be used as a backup, but for our purposes it serves only as a local copy. A replica provides a method to scan its data (entries()) and a method to fetch a single element (getEntry()). Its interface is shown in Listing 3.4.
public interface Replica extends MappingSet {

/* get a single element */
IndexView getEntry(long oid);

/* scan over all entries */
PullOperator<IndexView> entries();
}

Listing 3.4: Replica interface

3.4 Materialized View Manager

A materialized view is the precomputed and stored result to a query. Besides storing the IDs of the resource views in the result, a materialized view may also store any number of additional fragments (shredding a resource view is explained in Section 2.1.2).

To create a materialized view, a logical query plan has to be provided. This query plan represents the query which we want to materialize. Besides the query plan, a pull operator providing the results of the query must be provided. It is also possible to provide an optional list of additional fragments to store. Each materialized view also receives a name which can later be used to delete it.

In order to use a materialized view the query which is being processed has to be pushed down to the materialized view manager. The materialized view manager (MVM) is another logical query planner and therefore implements the LogicalQueryPlanner interface, see Listing 3.2. The interface of the materialized view manager is shown in Listing 3.5.

3.4.1 Query Pushdown Containment Test

The functionality presented in this section could also be used for indexes, but currently we only use it for materialized views. When a query is pushed down to the materialized view manager, we have to find out if any materialized view can be used to process that query. To do so, two logical tree walkers are used:

Containment Tree Walker

The first tree walker we use is the containment tree walker. It traverses the received query tree node by node. At each node it performs an equivalence test between the current subplan and all materialized views. This test is performed using two query containment tests, one in each direction: First, whether the current subplan is contained in a materialized view and, second, whether the materialized view is also contained in the current subplan.

If a match is found, we know the current subplan and the materialized view are semantically equivalent, that is the materialized view produces exactly the results which have to be produced by that subplan. The current subplan is then replaced with a black box (see Section 2.2) for the matching materialized view.

It is worth noting that an equivalence test is more powerful than just comparing two query plans node by node. This can be seen from the fact that two query plans may produce the same results even though they are structurally different. An example of such a case is shown in Figure 3.2.

Superset Tree Walker

The second tree walker we use is the superset tree walker. After we performed the equivalence tests with the containment tree walker, there is still a chance left to use a materialized view and postfilter its results.
public interface MaterializedViewManager extends LogicalQueryPlanner {

    /* try to process query. */
    public LogicalNode query(LogicalNode lognode);

    /* create materialized view. */
    public String createMaterializedView(String name, LogicalNode compiledQuery,
                                         PullOperator<OIDResourceView> resultOfQuery)
        throws MaterializedViewAlreadyExistsException;

    /* create a materialized view along with a given set of fragments. */
    public String createMaterializedView(String name, LogicalNode compiledQuery,
                                          PullOperator<? extends OIDResourceView> resultOfQuery,
                                          List<String> fragments)
        throws MaterializedViewAlreadyExistsException;

    /* drop a materialized view. */
    public void dropMaterializedView(String name)
        throws NoSuchMaterializedViewException;

    /* drop all materialized views. */
    public void clear();

    /* update a materialized with new results. */
    public void refreshMaterializedView(String name, PullOperator<
                                         OIDResourceView> newResultOfQuery)
        throws NoSuchMaterializedViewException;

    /* test if materialized view exists. */
    public boolean exists(String name);

    /* get names of existing materialized views. */
    public Set<String> getMaterializedViewNames();

    /* print query plans of existing materialized views to SVG. */
    public String printMaterializedViews();

    /* get a pull operator of index views by name. */
    public IndexViewPullOperator getPullOperator(String name);

    /* get a pull operator of OID arrays by name. */
    public PullOperator<OIDArray> getOIDArray(String name);
}

Listing 3.5: MaterializedViewManager interface
To check for such cases, we look for a Filter node above an All node in the received query plan. If we find such a case, we loop over all materialized views and test if the current subplan is contained in a materialized view.

If a materialized view is found that is a superset of the current subplan, we replace the All node of the subplan with the black box for this materialized view. This is illustrated in Figure 3.3. We assume in this example that the materialized view represented as black box $BB_a$ produces more results than $\sigma_a$ above All.

### 3.4.2 Query Pushdown Tree Comparison

In Section 3.4.1 we have shown a first approach to find materialized views which can be used to process a query. That approach is based on query containment and may therefore be expensive.

In this section we present another approach to find possible materialized views. It is based on simple tree comparisons. Its advantage is that it is much faster to execute than the query containment approach, because it is based only on node comparisons. Its plain disadvantage is the fact that semantically equivalent query plans are not recognized if they are structurally different. Figure 3.2 shows such two plans.

Given this limitation, this approach still works well if the query plans to process can be expected to have the same structure as the query plans used to create the materialized views.

#### Tree Comparison Implementation

To find materialized views which can be used in a received query plan, we instantiate one rewrite rule per materialized view. Such a rewrite rule is a tree walker [9] which matches a subplan that is structurally equivalent to the query plan which was used to create the materialized view.

To actually find matching subplans, we traverse the query plan for all of these rules. If a match was found, we can replace the matching subplan with a black box for the materialized view corresponding to
3.4 Materialized View Manager

3.4.3 Materialized View Serialization

Until now we have discussed how the materialized view manager is used and how queries are processed. Another important feature of the materialized view manager is how the materialized views are stored on disk.

Writing Materialized Views to Disk

We use two files to store a materialized view: A data file and a metadata file.

After opening the two output streams, but before storing the first resource view, we first write the beginning of the metadata file. For each additional fragment, we write its name and its data type. We support all primitive data types supported by `DataOutputStream`:

- Boolean
- Integer
- Long
- Float
- Double
- String
- Date

Now we write the data file. For each resource view we store its ID and the additional fragments. After all resource views are processed, we append the number of resource views in this materialized view to the metadata file and we are done.

An example of the two files used to store a materialized view is given in Figure 3.4.

Reading Materialized Views from Disk

To read a materialized view, we use a buffer which can hold several resource views. Whenever the next resource view is requested, we first make sure the buffer is not empty. To read resource views from disk, the read methods of `DataInputStream` are used.

From the metadata file we know which additional fragments have to be read and in which order. This is required, because the fragments are stored one after each other in the data file without the name or the type of the fragments. From the metadata file we also know how many resource views are in this materialized view. If all resource views are consumed, we close the input streams and we are done.
Currently, we do not handle heterogeneous fragments. A heterogeneous fragment is a fragment which can have several types. Support for heterogeneous fragments should be added, because iDM is supposed to be schema-free.

### 3.5 Service Oriented Architecture

A Personal DataSpace Management System like iMeMex will be used in very different situations.

On a mobile phone it could be used to manage contact information, provide access to RSS feeds and emails and search in messages stored on the mobile phone. It would be used to handle calendar data formats, plain text, XML and email messages. Indexes and replicas might not be feasible, because storage space might be limited.

On a personal computer or workgroup server a broader variety of data formats and data sources should be supported. Indexes and replicas are to be expected.

As we can see from these scenarios, iMeMex should provide a mechanism so that it can be reconfigured with little effort for one situation or another. A service oriented architecture (SOA) gives this required flexibility. We have chosen OSGi [20] as our basis platform, because it is a well known and widely used service oriented architecture (e.g., Eclipse is based on OSGi).

In Section 3.5.1 we introduce our current system architecture which is based on OSGi. Sections 3.5.2 and 3.5.3 show two example use cases.

#### 3.5.1 OSGi

OSGi provides a platform for a service oriented architecture. It does this by defining a bundle concept. A bundle contains all functionality related to a single system component. This functionality can be offered to other system components by exporting it as a service. Other bundles in turn can import that service and use it. A system will contain a set of bundles which all are connected to each other in a bundle dependency graph. This graph has to be a directed acyclic graph (DAG), because otherwise bundle dependencies cannot be resolved.

An OSGi implementation provides the basic functionality to load and unload bundles at runtime while ensuring bundle dependencies are fulfilled. Besides that, the OSGi specification defines a large wealth of commonly useful services. These additional services can be implemented by an OSGi implementation vendor or any other group or company. All these additional services are regular bundles that can be loaded and unloaded at runtime.

iMeMex only uses the basic functionality to load and unload bundles as well as the HTTP service specified by OSGi. Its bundle dependency graph is shown in Figure 3.5. iMeMex currently uses Oscar [19] as its basic platform and the HTTP bundle from Knopflerfish [16].

Figure 3.6 shows a schematic view of our system architecture, which maps closely to the bundle dependency graph of Figure 3.5.

#### Bundle Dependencies and Resource View Implementations

Having introduced the need for a service oriented architecture and our usage of OSGi, we will now look at a few use cases, but before this, we highlight one technical implementation issue related to OSGi. It is important for query processing, first because it influences how the indexing process works for data sources in warehousing mode, second because it requires new resource view implementations for data sources in mediation mode.

To show the problem, consider Figure 3.7. Assume there is a resource view interface, A, defined in the data sources layer and another resource view interface, B, defined in the indexes and replicas layer. A and B both can be used by the resource view manager, because the resource view manager depends on the data sources layer as well as on the indexes and replicas layer. The problem, however, arises if the query processor also wants to use A or B, because the query processor should not depend on neither data sources nor indexes and replicas.
Figure 3.5: Bundle dependency graph
The solution to this problem is to define another interface C in the resource view manager and only forward resource views of type C to the query processor. Again see Figure 3.7. The only drawback of this solution is that C itself also may not depend on either A or B, because otherwise the query processor again would depend on one of the lower layers. What this means is that each resource view produced by either the Data Source layer or the Indexes and Replica layer has to be wrapped in the resource view manager to “hide” where the resource view came from to higher layers like the query processor.

Figure 3.8 shows the three important building blocks which are affected by this problem: Data sources, Indexes and Replicas and Resource View Manager.

The resource view interface defined in the data sources is the PluginIdentifiable Resource View. It is used for the communication between resource view manager and data sources. Indexes and Replicas also define one resource view interface: OID Resource View and another interface IndexView. The IndexView interface allows to specify which parts of a resource view are available. The resource view manager sends OID resource views (having all fragments) to the indexes and replicas. Indexes and Replicas return index views, which may have only certain fragments.

As mentioned in the solution description above, wrappers are needed

- to forward resource views from the data sources to the higher levels,
- to forward resource views from the indexes and replicas to the higher levels and
- to send resource views from the data sources to the indexes and replicas.
3.5 Service Oriented Architecture

Figure 3.7: Resource view implementation problem

Figure 3.8: Resource view mappers
3.5.2 Use Case 1: Mobile Phone

The first use case we look at is running iMeMex on a mobile phone. We started to explore this scenario in Section 3.5.1, and now provide more details.

Besides making phone calls, people today also want to use their mobile phone to store calendar data, contact information and short messages. Frequently also music, pictures and sometimes even movies can be found. Ever faster mobile internet access also allows to read emails and browse on the web.

Given such a scenario we have the ideal situation to use iMeMex: a user might like to have tighter semantic integration between contact information and all the other data stored on his mobile phone. Going a step further semantic integration could be extended to cover all the data belonging to a person. A few examples are: data stored on the person’s desktop computer, online storage and emails on different servers. The iTrails framework [9] could be used to do that. This thesis focuses on query processing and therefore we would like to know how iMeMex could be deployed on a mobile phone to enable querying in such a use case.

Limited Computing Resources

Mobile phones are growing in memory, storage space and processing power. But still these resources are very limited because of their cost. A first requirement to deploy iMeMex on a mobile phone is therefore the ability to remove unneeded system functionality.

Sparse Storage Space

Many mobile phones have either no filesystem at all or very little storage space at best. Therefore all system functionality related to indexes, replicas and materialized views could be avoided. A catalog providing information about which data sources are known to the system and possibly other information could still be kept in a in-memory database.

Data Formats

Even though people start using different multimedia formats on their mobile phones, there are still not as many data formats in use as on personal computers. An office document, for example, is hard to edit on a tiny mobile phone screen. Since fewer data formats are available on mobile phones, not all content to iDM converters have to be deployed.

Data Sources

Similar to the data formats, it might not be needed to support all data source access protocols that are needed on a regular computer. If no applications need access to a relational database, then no JDBC data source plugin is needed.

Deployment Scenario

Considering the information collected while studying this use case, a deployment scenario like the one shown in Figure 3.9 would be an option. Its main features are fewer data source plugins, fewer content to iDM converters and no warehousing services at all. System components painted in grey could be removed.

3.5.3 Use Case 2: Data Warehouse

A completely different use case compared to the mobile phone is a read-only data warehouse used to index, replicate and query huge amounts of data. On such a read-only server, no data source access protocols and therefore also no content to iDM converters are used. Since a warehouse has huge amounts of disk space and amazing CPU power, indexes and replicas can be used extensively.

A possible deployment scenario for a data warehouse is shown in Figure 3.10. System components painted in grey could be removed.
Figure 3.9: Possible deployment scenario for a mobile phone
Figure 3.10: Possible deployment scenario for a data warehouse
3.6 Query Flow

Query processing involves all the components described in this chapter. A possible query flow in iMeMex is depicted in Figure 3.11. We start in the Console where a user might type a query in iQL. The iQL string then passes through the dataspace services to the query processor. The Query Processor parses the iQL query and compiles a logical query plan. The query plan gets rewritten by the iTrails framework and is sent to the Resource View Manager. The Resource View Manager sends a copy of the query plan to the Materialized View Manager as well as to each Index and merges their answers. For all mediated data sources, a copy of the query plan is sent to one of our Data Source Plugins. Finally all results are unioned and returned.

Figure 3.11: Query flow
Chapter 4

Logical Query Planning

In this chapter a possible approach to query planning in iMeMex is presented. We first sketch the query processing model before this thesis and list its limitations. Afterwards a query planning model based on pushing down the query is introduced. We will see, that this type of query planning allows to effectively distribute the planning among heterogeneous data sources.

The remainder of this chapter then addresses how cost based planning in such an environment can be attacked.

4.1 Previous Query Processing Model

At first, all we did was pushing down predicates along with the component that we wished to query. A component is a part of a resource view. The concept of dividing resource views into smaller units is explained in Section 2.1. Figure 4.1 shows the query processing model before this thesis. Doing so allowed us to pass selections to our data source plugins as well as to our indexes and replicas. The results returned are physical pull operators which can directly be used to get resource views.

4.1.1 Limitations

As you might suspect, this leads to very harsh limitations. Since pushing down the selections is done separately for each component, there is no way to handle selections which are based on more than a single component. So for example a query on the name component cannot be combined with a restriction on the tuple component:

Example (1):  //eth*[size > 4096.0]

Here two logical filter nodes are generated: one which makes sure the name matches eth*, the other to get only those resource views which have a size attribute larger than 4096.0. See Figure 4.2.

By not providing selections on more than one component, we also cannot formulate a join on more than one component to execute on an index or data source plugin. Considering the fact that we might have a relational database as a data source plugin, we should be able to send joins to the data sources.

A second limitation came from the fact that we only pushed down predicates. So either you have very complicated predicates, which becomes unfeasible for aggregations or joins, or you allow to push down complete query trees. Doing the later naturally allows the data sources plugins and indexes to work on whatever type of query is supported in the system. As an example, have another look at the query plan produced by our iQL compiler in Figure 4.2. The two logical filter nodes are combined using an intersect. Now to be able to push down the intersect, you would either have to rewrite the intersect together with the two filter nodes to a single filter with a large predicate, or you enable the system to push down query trees. We followed the second approach.
Figure 4.1: Previous query processing model

Figure 4.2: Selection on name and tuple component
4.1 Previous Query Processing Model

4.1.2 Moving towards Logical Nodes

The limitations presented in 4.1.1 were no longer bearable. A relational data source plugin using JDBC [22] has evolved and a remote iMeMex data source plugin [7] which allowed to connect several iMeMex instances was planned. Both of these data source plugins are able to process much more complex queries than simple selections, therefore the data source plugin interface was changed to receive logical nodes. See Figure 4.3. An introduction on this can be found in [9].

The great advantage of this was that now powerful data source plugins could really work on the complete query tree. The type of results produced changed from pull operator to logical node. The logical query trees returned were allowed to contain black boxes and therefore the query planners in the data source plugins gained the freedom to plan as much from the received query tree as they wanted. Black boxes are shortly explained in Section 2.2 and a more detailed discussion is given in Section 4.5.

As an example look at the query plan shown on the left of Figure 4.4. In the case where you want to pass the intersect to the data source plugins, you now just send a copy of the complete query plan down to the data source plugins. They would in turn replace the complete query tree with a single black box (if they are able to handle intersect and filter). The result is shown on the right of Figure 4.4.

Since the results became logical nodes, one more step has to be performed, before we get the physical pull operators. This step, which is to unwrap the physical pull operators stored in the black boxes is trivial, as long as you only have a single black box. If you receive a logical tree which consists of more than a single black box, you will have to traverse that tree until you find some black boxes, unwrap its pull operator and use it to plan the remaining logical nodes on top of the black box.
4.2 Query Push Down

The idea to push down complete query trees was born. What remained to do was to fully apply the idea to the whole system. Until now only the data source plugins received logical nodes, but the indexes still did not. And also were the indexes still separated by resource view component which made it impossible to have indexes spanning several components, e.g. no index was possible that at the same time indexes the name- and the tuple component.

After converting all indexes to receive logical nodes, the querying push down model now looks as shown in Figure 4.5. A copy of each query tree received by the Resource View Manager is now given to all data source plugins as well as to all indexes. Each of them tries to handle as much as possible of the query tree and replaces that part with a black box. From the returned results, which are still logical trees, all black boxes are collected and the possible execution plans get merged. How these black boxes are collected and merged afterwards is explained in Section 4.3.6.

4.3 Logical Choose

As described in Section 4.2 a query tree that is given to the Resource View Manager is copied several times to be given to each data source plugin and index. Each copy may result in a possible execution plan. Since we do not want to decide which possibility to use at the level of the Resource View Manager (or even further down), we need a way to collect these possibilities. A logical choose will do exactly this, see [11].

In this section we first give the definition of a logical choose, then show how it is used and at the end explain how combining different possibilities work.

4.3.1 Logical Choose Specification

A logical choose has no additional members, but has at least two children. See Figure 4.6. Each child is the result of processing a copy of the same logical query tree. That is each child may hold a possible execution plan for the same query.
4.3 Logical Choose

**Figure 4.6: Graphical specification of logical choose**

```
RVM input
\[\sigma_{\text{name} = \text{eth}}\]
```

**Figure 4.7: Logical choose usage**

### 4.3.2 Logical Choose Usage

To explain the usage of a logical choose, we will look at the following example:

**Example (2): */eth/***

*Everything which is directly related to everything which is called eth.*

The Resource View Manager receives the query, which is a logical tree. Since the Resource View Manager cannot process this query by itself, a copy of the logical tree is given to the Materialized View Manager and another copy is given to our Indexes and Replicas. Both Materialized View Manager and Indexes and Replicas try to process the query and return a logical tree. Depending on how much one could do, parts of the query may be replaced with black boxes. Now since the Resource View Manager receives two possible logical query trees, it adds them as possibilities to a choose. See Figure 4.7.

The Materialized View Manager found a Materialized View that exactly matched the posed query and therefore was able to replace the complete query tree with a black box. The Indexes and Replicas have not been able to handle the shallow unnest, but managed the filter over all. Therefore the shallow unnest appears on top of a black box in the result of the Indexes and Replicas. The two possibilities are then combined by the Resource View Manager inserting a choose and adding the two possibilities to it.

### 4.3.3 Why a Logical Choose?

It could be argued that a logical choose only collects possible execution plans. So why introduce another logical node type?

1. In fact, A logical choose really is not much more than a list of possible execution plans, but it has the great advantage, that it still is a logical node. This is important, because we do not require from a query planner to fully plan a received query. That is a result returned from a query planner may
still be partly logical and only have “physical” black boxes as leave nodes. The logical choose then allows us to combine these returned logical query trees and stay within a logical tree.

2. A logical choose could in a later phase of query planning be replaced with a physical choose. By introducing a logical choose we do not require to do so, but we mark the place where it would be feasible to insert such a physical choose. See Section 7.6 for more details about physical chooses.

4.3.4 Resolved Logical Query Plans

In one sentence, a resolved logical query plan is a logical query tree which has only black boxes as leaf nodes. The interesting feature about such a query plan is that all data accesses are already planned. To fully physically compile such a query plan, only operator implementations (like filter, union, intersect, . . . ) have to be available. This can also be seen in the fact that no logical nodes of type Start Views or All are allowed in such a plan.

The two first example logical query plans shown in Figure 4.8 still contain data accessor nodes and are therefore not considered resolved. The plans shown on the right instead, contain only black boxes as leaf nodes and are therefore resolved. The second example plan as a whole is not resolved, but it contains twigs which end in black boxes. We call such a query plan partly resolved.

4.3.5 Unneeded Logical Query Plans

In one sentence, an unneeded logical query plan is a logical query tree below a logical choose and besides a resolved query plan (as explained in 4.3.4). The interesting characteristic of such a query plan is, that it is not needed and if it happens that it is not resolved either this query plan can be safely removed.

For instance, if you have a look at Figure 4.9, you can see a logical choose on top and a resolved logical query plan on the left (a single black box). The query tree on the right (shallow unnest over start views) is not resolved and also unneeded and can therefore be removed - it is an unneeded and unresolved logical query plan.

4.3.6 Alternative Merge

As shown in Section 4.3 a logical choose combines alternatives produced by different logical query planners. Not all of these alternatives necessarily have to be resolved (as explained in Section 4.3.4) or needed (as explained in Section 4.3.5).

This section explains a recursive algorithm to merge all the different resolved logical query plans. Its implementation is shown in listing 4.1.
public static LogicalNode merge(LogicalNode original, List<LogicalNode> possibilities) {

    // choose on top
    // add black boxes as possibilities
    // and collect non-black boxes
    Choose choose = new Choose();
    List<LogicalNode> remainingPossibilities = new ArrayList<LogicalNode>();
    for (LogicalNode possibility : possibilities) {
        if (possibility.getType() == LogicalNodeType.BLACK_BOX) {
            // black boxes are added as possibilities
            choose.addPossibility(possibility);
        } else {
            // keep remaining possibilities for further traversing
            remainingPossibilities.add(possibility);
        }
    }

    if (remainingPossibilities.size() > 0) {
        // given logical node was not fully processed - continue
        choose.addPossibility(original);

        // recurse for all children
        for (int i = 0; i < original.nrOfChildren(); i++) {
            // find next original and next possibilities
            LogicalNode nextOriginal = original.getChild(i);
            List<LogicalNode> nextPossibilities = new ArrayList<LogicalNode>();
            for (LogicalNode remainingPossibility : remainingPossibilities) {
                nextPossibilities.add(remainingPossibility.getChild(i));
            }

            // recurse
            LogicalNode nextChoose = merge(nextOriginal, nextPossibilities);
            original.replace(nextOriginal, nextChoose);
        }
    }

    return choose;
}

Listing 4.1: Alternative Merge
To explain the algorithm, we use the example show in Figure 4.10. On the left you can see the original query plan to process. On the right, four results of a query plan push down are given. These four query plans are our alternatives which we want to merge:

1. A single black box (everything was replaced by the logical query planner)
2. Two filters replaced
3. Two other filters replaced
4. One intersect replaced

For each recursion step, the algorithm starts by putting a logical choose on top (line 7). Then it loops over all given alternatives (line 9-19). If an alternative is of type black box, it will be added to the newly introduced choose (line 13). For any other type of logical node, we keep that alternative for the next recursion (line 17). If there are alternatives left (line 21), we add the original query tree given at this level to the choose (line 24). We do this, because the remaining alternatives all have this logical node on top. Now we recurse on all children of the original query tree at this level (line 27-39). So for each child of the original query tree, we get the according child of all remaining alternatives (line 32-34) and recurse (line 37+38). Finally we return the new choose (line 42).

To make all this more clear, please have a look at Figure 4.11, which shows the resulting query tree after one recursion. Here a new logical choose was introduced on top and alternatives which had a black box on top (only the first alternative in this example) were added to it. Since there were remaining alternatives left, the original query tree was also added to the choose and we have to recurse.
4.4 Logical Fetch

The new input to the next recursion is also highlighted in Figure 4.11. What we see is that the recursion continues on the first child of the previous original query tree.

The result of this algorithm is shown in Figure 4.12. Looking more closely at it reveals, that several branches of it are not neccessary. For example the branch on the right of $BB_{21}$ is not needed, because we already have two resolved logical query plans besides it (for more details on resolved and unneeded, see Section 4.3.4 and Section 4.3.5). After removing such branches, we get the query plan shown in Figure 4.13 on the left. If we also remove trivial chooses, we get the final result shown on the right in Figure 4.13.

4.3.7 Complexity of Alternative Merge

It is interesting to notice that the complexity of the algorithm presented in Section 4.3.6 is linear in the size of its input (original query plan and all alternatives):

$$C_{\text{AlternativeMerge}} := O(3 \times n \times p)$$

(4.1)

Where $n$ is the number of nodes in the original query plan and $p$ is the number of alternatives. The constant three is caused by the fact, that we traverse the tree three times, once for the actual merge, once more for the removal of unneeded branches and a third time to remove trivial chooses.

Since many alternatives may replace a large part of the original query plan with black boxes, the average case is most probably even quite a bit smaller than the one state in the equation above.

4.4 Logical Fetch

For any query that should be executed, all that is given to the query processor is either a string in iQL which will be compiled to a single logical query tree or a single precomputed logical query tree. In both cases, all we get to execute a query is a single logical query tree. The results returned by the query processor are pull operators of Resource Views.

A query planner would like to know which parts of the returned Resource Views the client is interested in (a Resource View can be shredded vertically and horizontally into fragments, see Section 2.1.2 for details). There are several reasons why this knowledge is important for query planners:
Figure 4.12: Alternative merge example - after merge

Figure 4.13: Alternative merge example - simplified
1. For a physical query planner, depending on what data is finally needed, it might be cheaper to fetch the data for all results in one shot instead of requesting a small amount of data for each result.

2. For a cost-based query planner, the cost of an operator might vary extensively, whether the data is already there, or if it has to be fetched first. As an example, consider a filter on a tuple attribute. If the attribute value is already in-memory, evaluating the filter might become a very cheap operation compared to evaluating the filter and fetching the attribute values first.

Both of these reasons arise because of the fact that iDM [5] is inherently lazy. A lazy data model is nice, because you always get everything but you only pay for it, when you really want to see it. On the other hand in many cases it might be faster to have a non-lazy data model (as described in the first reason above). To make use of both advantages, you only need a way to make your query executions partly eager - a logical and/or physical fetch can do that for you.

Another reason why it is nice to have a logical fetch is that it allows you to specify already on the logical query tree in which points of the plan the data in the final physical tree should be fetched.

As for the logical choose 4.3, a logical fetch does not imply to have a physical fetch. An example for such a case would be, if you are only interested in cost-based query planning. There you could just ignore the logical fetch during physical query planning.

### 4.4.1 Logical Fetch Specification

A logical fetch has exactly one child and at least one requested fragment. See Figure 4.14.

![Figure 4.14: Graphical specification of logical fetch](image)

### 4.4.2 Logical Fetch Placement

One obvious place for a logical fetch is the top of a logical query plan. Such a fetch can tell the query planner which fragments a client is interested in. Another straightforward place is below logical nodes which during physical planning require a certain fragment before they can be executed. Such an example is a filter with some predicate over for example the name. Both situations are presented in Figure 4.15.

Having fetches in these intuitive places ensures, on the logical side, that query planners may consider fetching the data in a reasonable way and ensures on the physical side, that the operators on top can be executed in a speedy way. What it does not help us with currently is the cost-based query planning, because to correctly estimate the cost of a logical node, we must ensure that:

1. All the fetches in the logical query plan are there if they are needed and,

2. equally important, are not there if they are not needed.

Any logical fetch is only not needed, when somebody else already provides the fragments it requests. Now this somebody can only be a black box (which is our way of getting data). Therefore a black box needs to be able to tell which fragments it provides. It does that by providing the method `getFragments()`. For more details about what else a black box provides, have a look at Section 4.5.
4.4.3 Removing Unneeded Fetches

Introducing logical fetches in the intuitive places explained in Section 4.4.2 puts fetches into all places where they might eventually be needed (criterion one of Section 4.4.2). What is left to be done is to remove fetches where they are not needed, because black boxes already provide the requested fragment.

One approach is to push down the fetches until they hit a black box and afterwards pull them up as far as possible. This approach is studied in Section 4.4.4. Another approach would be its inverse, to first pull up what is provided by the black boxes, and afterwards to push down. This approach is similar to dead variable detection presented in [1] and is considered in Section 4.4.7.

4.4.4 Fetch Pushdown and Pull-Up

In this subsection we present a recursive algorithm which allows to remove unneeded fetches. To start with, we assume fetches are present in the places as explained in Section 4.4.2.

This algorithm pushes fetches down until they are stopped by black boxes (pushdown phase) and afterwards pulls the fetches up as far as possible (pull-up phase).

Reduced Logical Algebra

During the pushdown phase, not all possible logical node types (see Section 2.2) have to be considered:

- From the possible input nodes, only black boxes may appear at this stage of the query planning.
- Logical measure nodes are not yet inserted into the plan, therefore they too can be ignored.

No Pushdown Through Unnests

For most of our logical node types you can gain something by pushing down the fetch. This is so, because most of our logical node types represent operators which forward resource views that appear in their input
4.4 Logical Fetch

Figure 4.16: Unnests produce new results

and therefore fragments which are present in the input may appear again in the output.

On the other hand it does not make sense to push down a fetch through a shallow- or deep unnest, since these logical node types produce results that do not appear in their input. Compared to all other types of logical nodes, unnests do not forward resource views which appear in their input. They act as a kind of data accessor nodes. See Figure 4.16.

Fragments that are requested above an unnest are not needed to process the unnest (no fragments have to be present to run an unnest). Fragments that are present below an unnest are lost after processing the unnest, because the unnest outputs new results.

The Pushdown Phase

The pushdown phase is relatively easy to understand. We start at the top and work our way down. If we encounter required fragments, they are added to the fetch being pushed down. If we hit a black box, fragments stored in the black box are removed from the fetch and we return. If we reach an unnest, we don’t push collected fragments through, but continue below it. See Figure 4.17.

As an example of how the pushdown phase works, have a look at Figure 4.18.

The implementation of the fetch pushdown algorithm is given in the appendix in Section C.1.

Pull-Up

After the pushdown phase, we have removed fetches which were provided by black boxes and were therefore not needed. Now we need to pull up fetches in the query tree as high as possible. This can be seen from the fact that cardinalities decrease the higher we are in the query tree.

As an example have another look at Figure 4.18. The logical fetch shown in the right most figure is actually not needed, because all intersected resource views will have the name fetched from the right input. To recognize such cases, we run the pull-up phase of this algorithm. Figure 4.19 finishes the previous example.
In Figure 4.19 the fetch on the name fragment was removed, because the right input to the intersect already provided that fragment and since an intersect outputs only results which appear in both inputs, no fetch is needed anymore after the intersect. So no pull-up of the fetch was needed and since the intersect also does not require the name fragment to be evaluated, the fetch from the left input is not needed at all and can be removed.

In this phase we have to really carefully analyze where we should pull up a fetch, where we should leave it where it is, or where we can remove it. This analysis has to be done per fragment, because one fragment might be better fetched later, whereas another fragment must be fetched where it is now.

The Pull-Up Phase

During the pull-up phase we will have to decide whether a fragment can be pulled up or not. To do so we use a decision tree (Figure 4.20) and a table to look up information about logical node types (Table 4.1). A minus in the last column indicates, that this information is not needed.

For each fragment we want to pull up, we first look at the logical node above. If this logical node requires the fragment we are currently working on, then this fragment has to stay (upper left branch of decision tree). A fragment is required, if the logical node needs the fragment to be evaluated (e.g. filter).

If the fragment is not required, we look at all inputs of the logical node above. If they all fetch the fragment we are considering at the moment, then we follow the “yes” output in the middle of our decision tree, otherwise we follow the “no” output. As an example, a distinct on top of a fetch for fragment \(x\) will answer yes to the second question in the decision tree. See Figure 4.21.

If all inputs fetch the given fragment, then normally it would make sense, to pull up this fragment, but if the logical node above introduces new results (e.g. unnest), then we can safely remove the fetch, because whatever we fetched now, would not appear in these new results anyway (lower left branch in decision tree).

If we answered no to the second question in the decision tree, that is, if not all inputs of the logical node above fetch the current fragment, then we know, that at least one of the inputs provides that fragment.
4.4 Logical Fetch

Figure 4.20: Fragment pull-up decision tree

### Table 4.1: Logical node type categorization

<table>
<thead>
<tr>
<th>logical node type</th>
<th>#inputs</th>
<th>requiring</th>
<th>new results?</th>
<th>output?</th>
</tr>
</thead>
<tbody>
<tr>
<td>distinct</td>
<td>Λ</td>
<td>1</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>filter</td>
<td>σφ</td>
<td>1</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>intersect</td>
<td>∩</td>
<td>2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>union</td>
<td>∪</td>
<td>2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>minus</td>
<td>-</td>
<td>2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>θ-join</td>
<td>θφ</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Left outer join</td>
<td>⊓</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Right outer join</td>
<td>⊔</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Full outer join</td>
<td>⊐</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>choose</td>
<td>?&gt;1</td>
<td>&gt;1</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>shallow unnest</td>
<td>/</td>
<td>1</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>deep unnest</td>
<td>//</td>
<td>1</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>project</td>
<td>π_1</td>
<td>1</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>order by</td>
<td>α_1</td>
<td>1</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>group by</td>
<td>g_1</td>
<td>1</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 4.21: Pull-up example distinct
Now, if my input (the inputs where this fragment comes from) is not used in the output of the logical node above, then this fragment may be removed, otherwise it has to stay (lower right branch in decision tree). A “narrowing” logical node (e.g. intersect) does not need the data of all its inputs. If a logical node is not narrowing (e.g. union), then the fragment can not be removed, because some of its input may appear in the output of the logical node above and the fetch can also not be pulled up, since some of the output of the logical node may contain elements from the input which already provides that fragment. Figure 4.22 show this case where the logical node above is a union.

The implementation of the fetch pull-up algorithm is given in the appendix in Section C.2.

θ-Join Example

To clarify the categorization shown in Table 4.1, we show three more examples. First the θ-join.

A possible query plan to execute the join is shown in the upper left corner of Figure 4.23. If the black boxes provide no fragments, then after pushing down the fetch, we get the query plan shown in the upper middle. No fragment was removed, so after pulling up the fetches as much as possible, we get the query plan in the upper right corner.

It becomes more interesting, if one of the black boxes provides the fragment that is requested on top. This situation is shown in the lower left corner. In this case, after the pushdown, we get the figure shown in the lower middle. There the “name” fragment was removed from the fetch on the left, because the fragment is provided by the black box below. And finally after the pull-up, the name fragment on the right can also be removed, because the name fragment is already provided by the input on the left. So we get the result shown on the lower right.

Minus Example

To explain the behavior of logical minus, have a look at Figure 4.24. In this figure, we show all possible combinations of whether a black box provides a fragment or not. In the first row, the black boxes provide nothing. In the second and third row one of the black boxes provides a fragment. In the fourth row, both black boxes provide the requested fragment. This figure also shows cardinalities of outputs.

If no fragments are provided by the black boxes (1st row), then we must keep the fetch has high as possible. If both black boxes provide the fragment (4th row), then the fetch disappears already after the pushdown. Now if only one of the black boxes provides the fragment, it becomes more interesting. Referring back to table 4.1, we see that inputs below a minus only have to provide output if they are on the left side. Therefore if we have a fetch on the left, it has to be preserved, that is pulled up (3rd row). If on the other hand, we have a fetch on the right only, it can be removed (2nd row).

Combination of Intersect and Union

As a last example, we show that our pushdown and pull-up algorithm also works for combinations of different logical node types. Here we have choose intersect and union. See Figure 4.25.
Pushdown works fairly well. In the third query plan of Figure 4.25 we can see, that one fetch was correctly removed. During Pull-Up another fetch can be removed, because the other input already provides that fragment.

If you look closely at the final result shown in Figure 4.25, you will notice that now the fetch has to get data for 50 elements whereas it would have had to fetch only the data for 20 elements were it on the top. This shows a very important feature of the algorithm which we will further explore in Section 4.4.5.

### 4.4.5 Characteristics of Pushdown and Pull-Up

Our Pushdown and Pull-Up algorithm presented in Section 4.4.4 is correct in a sense, that it puts fetches only where they are really needed. Its limitation comes from the fact that a correct plan in that sense is not necessarily the best plan to execute.

To recognize this problem, have a look at Figure 4.25 and Figure 4.26.

The later example (Figure 4.26) highlights the problems of a query plan where fetches are introduced correctly. The query plan in the middle was generated using the pushdown and pull-up algorithm. At each level in the query plan, we know exactly which fragments are available, but we fetch $x$ for 30 resource views. The query plan on the right on the other hand, delays fetching $x$ until the end. We therefore have to fetch $x$ only 10 times and doing so, get a lower total cost. The only problem in this solution is that we loose the exact knowledge of which fragments are available where in the query plan and therefore estimating the cost of the upper most fetch becomes really difficult.

A nice feature of a query plan generated by our pushdown and pull-up algorithm is that it is straightforward to get an optimal plan for query execution with respect for fetch placement (as shown on the right of Figure 4.26). All you have to do is pull up all fetches until no more reducing logical nodes are above any fetch. It can therefore be seen as an algorithm which reduces the problem of distributing logical fetches in a logical query plan optimal for query execution to a very simple problem.
Figure 4.24: Pull-up minus example
Figure 4.25: Pull-up combination of intersect and union example

Figure 4.26: Pull-up combination of filter and union example
4.4.6 Logical Provide

Before we show a second way to remove unneeded fetches in a logical query plan, we introduce another logical node type - logical provide. This logical node type is currently only used to detect unneeded fetches. Its graphical specification can be found in Figure 4.27.

![Logical Provide](image)

Figure 4.27: Graphical specification of logical provide

A logical provide has a single input and a set of provide fragments.

4.4.7 Dead Fragment Detection

In this subsection an alternative recursive algorithm is presented to remove not needed fetches. We compare it with the algorithm presented in Section 4.4.4. To start with, we assume a single fetch is present on top of the query plan. This algorithm inserts required fetches as it proceeds.

As already mentioned in Section 4.4.3, this algorithm first pulls up what is provided by the black boxes, and afterwards pushes down remaining fetches as far as possible.

Introducing Logical Provide Nodes

To pull up fragments provided by black boxes, logical provide nodes (see Section 4.4.6) are used. As the first step, such logical provides are introduced above each black box which provide some fragments. An example is shown in the left two query plans of Figure 4.28.

The Pull-Up Phase

In the first phase of this algorithm, logical provides are pulled up as far as possible. Now “as far as possible” needs to be defined. As we have seen in Figure 4.28, if a logical provide reaches an intersect, it may go through it, whether the other input of the intersect also has a logical provide on the same fragment or not. Therefore logical intersect acts as a union for provided fragments:

\[ \cap := \cup^\text{Provided Fragments} \]  

(4.2)

![Dead Fragment Detection - Intersect](image)

Figure 4.28: Dead fragment detection - intersect
A logical union behaves exactly the other way around. It becomes an intersect for provided fragments:

\[ \cup := \cap_{\text{Provided Fragments}} \]  

(4.3)

An example of how union behaves in dead fragment detection is shown in Figure 4.29.

A complete list of how each logical node type behaves during the pull-up phase of dead fragment detection is given in table 4.2. The first three data columns (#inputs, requiring and new results?) are the same as in table 4.1.

The Pushdown Phase

In the second phase of this algorithm, remaining logical fetches are pushed down as far as possible. Again “as far as possible” has to be defined. As a rule of thumb, you may not push down fetches through logical nodes which decrease the number of elements (e.g. filter or distinct). A complete list of when it is allowed to push down a remaining fetch is given in table 4.2 (in the “down?” column).

It can be seen that pushing down a remaining fetch is only feasible for union, choose, project and order by. For union and choose this can reduce the number of elements which have to be fetched. For project

Table 4.2: Logical node type categorization - dead fragment detection

<table>
<thead>
<tr>
<th>logical node type</th>
<th>#inputs</th>
<th>requiring</th>
<th>new results?</th>
<th>up/down?</th>
</tr>
</thead>
<tbody>
<tr>
<td>distinct</td>
<td>1</td>
<td>no</td>
<td>no</td>
<td>yes/no</td>
</tr>
<tr>
<td>filter</td>
<td>( \sigma_{\phi} )</td>
<td>1</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>intersect</td>
<td>( \cap )</td>
<td>2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>union</td>
<td>( \cup )</td>
<td>2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>minus</td>
<td></td>
<td>2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>( \theta )-join</td>
<td>( \theta )</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Left outer join</td>
<td>( \geq )</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Right outer join</td>
<td>( \preceq )</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Full outer join</td>
<td>( \supseteq )</td>
<td>2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>choose</td>
<td>( ? )</td>
<td>&gt; 1</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>shallow unnest</td>
<td>/</td>
<td>1</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>deep unnest</td>
<td>//</td>
<td>1</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>project</td>
<td>( \pi_{l_{-1}} )</td>
<td>1</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>order by</td>
<td>( \alpha_{l_{-1}} )</td>
<td>1</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>group by</td>
<td>( g_{l_{-1}} )</td>
<td>1</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
and order by, it does not change anything, but also does not hurt, since the number of elements is the same below and above such a logical node.

An example of why it is needed to push down after pulling up provided fragments is shown in Figure 4.29. Here it is shown that since the provide may not go up through the union, the fetch has to go down to take advantage of the provided fragments.

The implementation of dead fragment detection is too long to be printed in this thesis. It can be downloaded from the homepage of iMeMex [15].

4.4.8 Pushdown and Pull-Up vs. Dead Fragment Detection

Both algorithms presented in this section, Pushdown and Pull-Up and Dead Fragment Detection, solve the same problem: They insert logical fetches into logical query plans at optimal places for later query execution.

Since they both solve the same problem, you would expect their output to be the same - would you not? Now this is not so, because an optimal logical query plan, with respect to query execution, is not unambiguous. A very simple example reveals that. See Figure 4.30.

The query plan on the left of Figure 4.30 would be produced by Dead Fragment Detection, where the plan on the right, would be one produced by Pushdown and Pull-Up. Dead Fragment Detection puts fetches as far down as possible compared to Pushdown and Pull-Up which pulls fetches as high up as possible.

Complexity Analysis

There is not much to say about the runtime complexities of either Pushdown and Pull-Up or Dead Fragment Detection, but still it might be interesting to note, that both of them are linear in time and space.

Since both algorithms traverse the tree only two times, they are linear in time. Since none of the algorithms introduces more than one additional logical node per existing logical node, they are linear in space:

\[ C_{FetchPlacement} := O(2 * n) \] (4.4)

Where \( n \) is the number of nodes in the input (the query plan to insert the fetches). Times two reflects two traversals of the logical node tree.

4.4.9 Fetch vs. Projection

The logical fetch is different from the logical projection. Projection restricts access to certain fragments where fetch only makes sure these attributes are eagerly fetched.
4.5 Black Box

A black box is a new type of logical node which may appear as leaves in a logical query tree. It not only is a general way to express access to data, it also features distributed query planning and sharing responsibilities between query planners that want to work together.

They are first mentioned in the Master’s thesis of Olivier Girard [9], where one possible use case, the pushdown of a logical query plan from the logical to the physical layer, is explained. Here we would like to continue and extend the usage of these black boxes.

4.5.1 Black Box Specification

Besides providing an output operator, black boxes also store information, which allows you to integrate them into your own cost-based query planning. Below all information stored on a black box are explained. A graphical summary can be seen in Figure 4.31.

Estimated Cardinality

Cost-based decisions often depend on the knowledge of how many results will be produced by the underlying operator. If one operator knows how expensive it is to process a single element, he will need to know how many results there are.

Cost

A price in units of the cost model of the vendor of this black box. The ability to specify its own price in its own cost model simplifies the cost calculation for the lower query planners. A query planner which integrates black boxes of other query planners into its own planning will have to translate from the former cost model into its own cost model.

More details on how this price should be understood in our own cost model and how to translate one cost model into another is given in Section 5.1.

Execution Time

Time required to execute this black box. It can be seen as a price in milliseconds. This “cost model” can be understood by all query planners, but might be more difficult to provide. If a cost in milliseconds is provided, it will be combined with the cost calculated in higher query planners.

On a black box arbitrary combinations of cost and execution time are allowed.
Provided Fragments

The data model used by iMeMex, iDM [5], calculates its data in a lazy manner - it is intensional. As a consequence, without special effort, you do not know, how expensive a call to retrieve on of the components of a resource view might be. Therefore the knowledge of which fragments of a resource view are already fetched by this black box is needed to insert logical fetches in the remainder of the logical query plan. See Section 4.4. Logical fetches in turn may be used to estimate the cost to fetch the data required by the operator on top.

Order

The order in which resource views are returned from this black box. This information may be needed during physical query planning to choose between different implementations of a logical operator. An example would be an intersect, which could be implemented using a hash-based or a merge-based approach.

In short, this information allows us to make use of interesting orders [21].

4.5.2 Generalization of Accessors

To execute a query most of the time you will need some data to evaluate it against. To get the data you will normally have to specify where from to get it and often you are also allowed to specify some predicate to filter the data before you get it.

In a relational database for example, all data is kept in tables. If you want some data, you will have to specify from which table and often a relational database management system also allows you to specify a filter criterion.

These two pieces of information are sometimes called “accessors” and ways to access the data “access paths” as mentioned in [2] and [21]. We now generalise this concept of accessors and call them “black boxes”.

A black box is a very powerful data accessor, because it does not only support selection over the data, but any kind of query supported in our logical algebra. A black box can also be seen as a high-level data accessor, because it may make use of a complete system beneath it to get the data. In Figure 4.32 we show that a black box can be used as a standard accessor, but also may be used replace complex queries. The black box introduced for the left branch may for example be an access to the filesystem, where the black box introduced for the branch on the right may be created by a relational database.

---

**Figure 4.32: Generalization of accessors**

---
4.6 Logical Measure

4.5.3 Physiological Query Plans

In Section 4.2 is explained how query plans are pushed down from higher to lower query planners. The returned type of this pushdown is still a logical query plan and not yet just a physical pull operator to get the results. This has to be so, because we do not require lower query planners to plan everything of a given logical query plan and therefore the returned results may still contain logical nodes.

What is needed is a way to combine the physically planned parts of a query plan with the not yet physically planned parts. A black box does exactly this. Since a black box is still a logical node, it can be used in a logical query plan and since a black box stores a physical output operator, it combines the planned with the not yet planned parts of a logical query plan.

A higher query planner collects the results of its lower query planner and combines them in a single query plan. Such a plan may then contain black boxes as leave nodes and if so is partly planned or physiological.

Every query plan containing black boxes is an example of a physiological query plan, but since most of the time we do not show what is hidden behind a black box, have look at Figure 4.33 which shows the inner life of a black box.

4.5.4 Distributed Query Planning

Since a black box can not only be used to express selections, but may represent large parts of a logical query plan, it can also be used to send a query to some other powerful query planners and afterwards combine their results.

Sharing Responsibilities

If a logical query plan is given to another query planner and this query planner replaces parts of it with black boxes, you actually trust that query planner, that whatever comes out of these black boxes is true. On the other side as a query planner who receives a logical query plan and replaces parts of it with a black box, you believe, that the remaining parts of the query plan can be handled by a higher query planner. Therefore by using black boxes, you share responsibilities between different query planners to successfully execute a query.

4.6 Logical Measure

A logical measure is a logical node which has a single input. Its purpose is to pinpoint places in a logical query plan where measurements should be performed.
During experiments it is important to be able to insert measurements at several places in the physical execution plan. To do so in iMeMex, a developer may configure a logical node pattern on top of which physical measurement operators are inserted. A logical node pattern describes the structure of a query subplan. A Filter node on top of an All node, for example, would be written as \texttt{filter(all)}.

Measure is a logical node, because we want to insert such nodes in the logical query plan. This allows us to pushdown the information where to perform measurements to lower query planners.
Chapter 5

Physical Query Planning

In this chapter we explain how our logical algebra and the logical query plans generated therefrom are transformed into physical execution plans. In Section 4.5 black boxes were introduced. In this chapter they are used as the input nodes during physical query planning.

Black boxes are still logical nodes which appear in a logical query plan, but they represent a physically planned part of the complete query plan. The inputs to this remaining physical planning phase are therefore not just logical plans, but physiological plans. These physiological plans consist of leaf nodes which are physically planned and the remaining logical nodes which still have to be physically planned.

The remainder of this chapter is organized as follows: Section 5.1 explores the opportunity to have different cost models on each black box. How to translate from one cost model to another is explained in Section 5.2. Then our cost-based query planners are introduced in Section 5.3.

5.1 Cost Models

The input given to physical query planning in iMeMex is a logical query plan which has all its leaves replaced by black boxes. The black boxes, even though they are logical nodes, encapsulate physical operators, which produce the results for that part of the logical query plan which is covered by the black box. The estimated cost to execute a black box are delivered by the black box.

Each physical query planner needs a way to estimate the cost of the execution plans it generates. To accomplish this, it has to have a knowledge of a cost model. That is, for all the operations it performs, it should be able to estimate its cost. Now a cost, or a price you have to pay to run a certain operation, always needs a unit, or in money terms, a currency. One property of currencies is that each currency may be exchanged into another currency. In iMeMex, this is important, because each black box may express its cost by its own currency. After this little excursion into cost models, we get:

- A black box provides its results in terms of an operator.
- To get the result, a price has to be payed. This cost is written on the black box and its unit can be any currency chosen by the query planner that generated the black box.

5.1.1 Multiple Cost Models

Each producer of black boxes may choose to specify the cost of a black box in terms of his own cost model. This is desirable, because for most query planners which produce black boxes, it is much easier to use some special cost model than to use a cost model which is used all over the system.

A materialized view manager for example might know how many entries are stored in each materialized view. Therefore a simple but sufficient cost model for a materialized view manager is to assign one cost unit for reading one entry. Depending on the size of the materialized view reading that materialized view may be more or less expensive. Using such a cost model, the materialized view manager does not need
to know how expensive it is to read one entry compared to other operations in the system, because all he needs to tell to the upper query planner is “executing that black box costs X units in my currency”.

The translation from the currency (or cost model) used by the materialized view manager to a currency which can be compared to other costs in the system is left to the upper query planner. How currencies are translated from one to another is explained in Section 5.2.

A logical query plan using multiple currencies is shown in Figure 5.1. In this figure, the black boxes in Swiss Francs (“Fr.”) could have been produced by the materialized view manager. The black box which states its price in Euro might be a full text index on a slow hard disk whereas the black box in US-Dollars might be an index in a relational database. From this figure we can see that as a higher-level query planner, we may get a logical query plan as input which contains several currencies.

5.1.2 A Common Cost Model

In the previous section we showed that it might be useful to have several different cost models during a single query planning phase. We therefore enabled a black box to not only store the cost as a number, but also remember its currency.

Since we might have several stacked query planners, a single currency might not be enough. Assume we have three layers of query planners. The lowest produces a black box using cost model A. This black box is used during query planning in the middle layer. The middle layer in turn wants to use its cost model B. There are now two options. Either the middle layer translates the cost of the black box from the lowest layer to its own cost model, or we allow to store two (or more) cost models per black box. Figure 5.2 shows this situation.

Instead of storing the cost of a black box in many different currencies, we only allow to store the cost in one cost model of choice plus another cost model which can be commonly used, i.e., execution time. The reason why time can be used by every query planner is that above each received black box an execution-time measuring operator can be inserted which translates from the received cost model to time. Translating from one cost model to time is performed by multiplying the cost with the exchange rate from cost model to time. This will be explain in more detail in the following subsection. We make the assumption that cost is proportional to time. Further load on the system changes that proportion.

The cost of a black box is stored in a class which can hold the cost in the cost model of the query planner plus the cost in time. This class is shown in Listing 5.1.

5.2 Translating Between Different Cost Models

In the previous section we saw that a black box stores two costs: one in a cost model of the black box producer’s choice, another in time (milliseconds). In this section, we explain how to translate from one cost model to another.
5.2 Translating Between Different Cost Models

Figure 5.2: Stacked query planners

```
/**
 * Cost which can be written on a black box.
 */
public class Cost {

    /* query planner which produced this black box */
    public String vendor;

    /* cardinality */
    public long numResults;

    /* cost in units of the query planner's cost model */
    public double totalCost;

    /* cost in time (milliseconds) */
    public double totalCostInTime;
}
```

Listing 5.1: Cost class
A cost model, or currency in general, is abstract and we know nothing about it. All we can do is to measure how a currency can be exchanged to time. To do so, we insert execution time measuring operators on top of black boxes. A measuring operator stores the estimated cost which was provided by the query planner of the black box. During execution of the query the measuring operator measures the real execution time needed to execute the black box. Given these two numbers an exchange rate can be calculated:

\[ \text{exchangeRate} = \frac{\text{measuredTime}}{\text{estimatedCost}} \]  \hspace{1cm} (5.1)

The process to insert execution time measuring operators and how planning continues is show in Figure 5.3.

Every higher query planner needs to find the exchange rates of its lower query planners. Finding and updating exchange rates is explained in Section 5.2.1. The information about all these exchange rates is stored by an exchange officer, see Section 5.2.2.

### 5.2.1 Exchange Rate Update Algorithms

The first time a lower query planner is encountered by a higher query planner, that is the first time a black box produced by a lower query planner has to be used in the query planning of a higher query planner, no exchange rate is known. To start with, we assume the exchange rate from the cost model of the lower query planner to time is 1.0 - which is often wrong.

The second time a black box from the same lower query planner is used in a higher query planner, we already know what the execution time of the previous black box was (if a measuring operator was placed on top of the black box). From that we have calculated a first estimate of an exchange rate. This exchange rate is now used during the second planning.

This second execution again provides the higher query planner with a measurement about the exchange rate from cost model of the lower query planner to time. This information should now be combined with the old information. We call a way to combine such measurements an exchange rate update algorithm. We have studied several algorithms to perform this, each having different characteristics.

#### Latest Measurement Only

The most basic approach is to ignore earlier measurements and always set the current exchange rate to the latest measurement. Such an implementation reacts very fast to changes of the exchange rate, but does not ignore peaks or bursts at all.

\[ ER' = \frac{\text{measuredTime}}{\text{estimatedCost}} \]  \hspace{1cm} (5.2)
5.2 Translating Between Different Cost Models

Average of Two

Considering the last exchange rate and taking the average with the latest measurement we get:

\[ ER' = \frac{ER + (\text{measuredTime}/\text{estimatedCost})}{2.0} \]  

This approach does not react to changes as fast as the latest measurement only approach, but it lowers the failures which occur during peaks and bursts.

Lowest of Three

Instead of looking at only the latest measurement or combining it with the previous measurement, we could keep a longer history. This update formula always keeps the last three estimated exchange rates. From these, the lowest is chosen.

\[ ER_3 = \frac{\text{measuredTime}}{\text{estimatedCost}}; \quad ER' = \text{smallest}(ER_1, ER_2, ER_3) \]

This approach ignores peaks of up to length two. Its disadvantage is that it needs three measurements to adapt to a new exchange rate if the exchange rate is going up, otherwise it would adapt immediately.

Average of Three

As for the lowest of three approach, we keep the last three estimated exchange rates. From these we take the average:

\[ ER' = \frac{ER_1 + ER_2 + ER_3}{3.0} \]

Peaks are not ignored as well as in lowest of three, but adapting to a new exchange rate happens slightly smoother.

Highest of Three

As for the lowest of three approach, we keep the last three estimated exchange rates. From these we take the highest:

\[ ER' = \text{highest}(ER_1, ER_2, ER_3) \]

Peaks are treated very badly. Adaption to a new exchange rate happens immediately if the exchange rates increases, otherwise its delayed by three.

Long Term Average

Another possibility would be to take a long term average of all ever estimated exchange rates.

\[ ER' = \frac{\text{average} \times \text{numElements} + (\text{measuredTime}/\text{estimatedCost})}{\text{numElements} + 1} \]

Depending on how many elements are used to calculate the long term average, peaks can be ignored quite well, but adapting to a new exchange rate might take very long.

Look Back Average

This is the most complicated update algorithm we present. It keeps a long term average plus a few of the last measurements. To remove bursts or peaks, this algorithm looks at the last few measurements. If the beginning and the end are similar to the long term average, then everything in between is considered a burst if it is not similar to the long term average and is removed from the list of kept measurements. Looking at the beginning and the end is important to recognize real changes in the exchange rate. Consider Figure 5.4. The long term average is shown as a dark magenta line. The range where measurements are still recognized
as being similar to the long term average is shown in bright magenta. The upper image shows a peak which
gets removed. In the lower image the peak is not removed, because the last of the kept measurements is
not similar to the long term average. Finally the exchange rate is calculated as the average of the remaining
kept measurements.

Comparison of Exchange Rate Update Algorithms

In this section we show a short tabular summary of the exchange rate update algorithms presented above.
See Table 5.1. The results for burst recognition were confirmed with an experiment (which we do not show
here). The results for change adaption have been derived from the implementation. As we can see from the
table, the look back average update algorithm performs best considering both burst recognition and change
adaption.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Burst Recognition</th>
<th>Change Adaption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latest Measurement Only</td>
<td>very bad</td>
<td>very good</td>
</tr>
<tr>
<td>Average of Two</td>
<td>bad</td>
<td>good</td>
</tr>
<tr>
<td>Lowest of Three</td>
<td>good</td>
<td>bad</td>
</tr>
<tr>
<td>Average of Three</td>
<td>bad</td>
<td>good</td>
</tr>
<tr>
<td>Highest of Three</td>
<td>good</td>
<td>bad</td>
</tr>
<tr>
<td>Long Term Average</td>
<td>good</td>
<td>very bad</td>
</tr>
<tr>
<td>Look Back Average</td>
<td>very good</td>
<td>good</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of exchange rate update algorithms

5.2.2 Exchange Officers

Every query planner which has to work with more than its own cost model, that is which receives black
boxes from lower query planners or which has to output cost in time, needs an exchange officer. An
exchange officer knows about exchange rates from currency to time. An exchange officer implements one
of the exchange rate update algorithms from Section 5.2.1. The interface of an exchange officer is shown
in Listing 5.2.
Listing 5.2: Exchange officer interface

5.3 Cost-Based Query Planners

In the previous two sections of this chapter we saw how we can use multiple cost models and explored algorithms to calculate an exchange rate from one cost model to another. In this section we study three approaches to physically plan the logical operators which remain in the query plan above the black boxes. The first approach works bottom-up and eagerly chooses the cheapest subplan. The second approach also works bottom-up, but keeps different interesting subplans. The third approach, dynamic programming, generates optimal query plans in terms of estimated execution cost, but is exponential in the number of joins in the plan.

5.3.1 Eager Choose Planner

The Eager Choose planner works bottom-up and traverses the query tree node by node. The first type of logical nodes it encounters are the black boxes which are the only allowed leaf nodes at this stage of query planning.

Black Box Translation

Black boxes may have a cost which is stored in a different cost model than the one we are using. Since we have no understanding of this cost model, we exchange the cost to time. The simplified code to plan a black box is shown in Listing 5.3.

It is important to notice that after exchanging the cost on the black boxes from their own cost model to time, the remaining cost-based operator planners can work completely in their own cost model or time.

Common Operators

After planning the leaf nodes, we plan the logical nodes right on top of the leaf nodes. We continue like this until we reach the top of the query plan or until we encounter a choose operator. In the meantime we may need to plan some common operators. The cost model used to plan them is shown in Table 5.2 where $n$ is the input cardinality and $c$ the cost of the input.

Eager Choose

When we reach a logical choose node we have to choose one of its alternatives. This Eager Choose planner eagerly chooses the cheapest alternative. The Eager Choose planner is explained in Listing 5.4. Before we can choose one of the available alternatives, all of the alternatives have to be planned (lines 3-8). The costs
public BlackBox planToCost(BlackBox bb) {
    // get estimated cost in lower query planner’s cost model
    Cost cost = bb.getCost();

    // put estimates into metadata object
    OperatorMetaData meta = new DefaultOperatorMetaData(cost.vendor,
                                                        cost.totalCost, cost.numResults);

    // insert execution time measuring operator which translates
    // from one cost model to time
    ExchangeOfficer officer = planner.getExchangeOfficer();
    PullOperator measurer = new ExchangeRateMeasuringPullOperator(
        bb.getPullOperator(), officer);

    // store estimated cost on operator to calculate exchange rates later
    measurer.setMetaData(meta);

    // replace physical operator in black box
    // and exchange cost to time
    bb.setPullOperator(measurer);
    officer.exchangeToTime(cost);

    return bb;
}

Listing 5.3: Black box planner

<table>
<thead>
<tr>
<th>Operator</th>
<th>Output Cardinality (n’)</th>
<th>Cost (c’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>distinct</td>
<td>Λ</td>
<td>n</td>
</tr>
<tr>
<td>filter</td>
<td>σ&lt;sub&gt;ϕ&lt;/sub&gt;</td>
<td>n * 0.1</td>
</tr>
<tr>
<td>intersect</td>
<td>∩</td>
<td>n&lt;sub&gt;left&lt;/sub&gt; ≤ n&lt;sub&gt;right&lt;/sub&gt; : n&lt;sub&gt;left&lt;/sub&gt; + n&lt;sub&gt;right&lt;/sub&gt;</td>
</tr>
<tr>
<td>union</td>
<td>∪</td>
<td>n&lt;sub&gt;left&lt;/sub&gt; + n&lt;sub&gt;right&lt;/sub&gt;</td>
</tr>
<tr>
<td>shallow unnest</td>
<td>/</td>
<td>n * 10.0</td>
</tr>
<tr>
<td>deep unnest</td>
<td>//</td>
<td>n * 1000.0</td>
</tr>
</tbody>
</table>

Table 5.2: Cost model of eager choose planner
of the alternatives may now be partly in our own cost model and partly in time. To be able to compare the
cost, the part of the cost which is stored in our own cost model has to be exchanged into time (lines 16-19).
Finally the cheapest alternative is choosen (lines 21-24) and returned.

```java
public BlackBox planToCost(Choose choose) {

    // plan children
    List bbs = new ArrayList();
    for (LogicalNode child : choose.getChildrenReadOnly()) {
        BlackBox bbChild = planner.planToCost(child);
        bbs.add(bbChild);
    }

    // eagerly choose one alternative according to lowest cost
    BlackBox bbChoosen = null;
    double minCostMs = Double.MAX_VALUE;
    for (BlackBox bb : bbs) {
        Cost cost = bb.getCost();
        // get cost in milliseconds
        Cost copy = cost.copy();
        ExchangeOfficer officer = planner.getExchangeOfficer();
        officer.exchangeToTime(copy);

        if (copy.totalCostInTime < minCostMs) {
            minCostMs = copy.totalCostInTime;
            bbChoosen = bb;
        }
    }

    return bbChoosen;
}
```

Listing 5.4: Eager choose planner

**Final Translation to Time**

After everything is planned we put another execution time measuring operator on top. This operator allows
us to derive an exchange rate from our own cost model to time. The implementation is similar to inserting
measuring operators on top of black boxes as shown in Listing 5.3.

**5.3.2 Global Effects**

The Eager Choose planner presented in Section 5.3.1 has one severe limitation: it does not consider global
effects. This limitation appears because eagerly choosing the cheapest alternative at one point is a local
decision which can only consider what is below that point in the query tree. Therefore, eagerly choosing
the cheapest alternative is not always the best option when looking at a query plan as a whole, because
there are global effects that cannot be decided locally.

**Interesting Order**

One such global effect is called *interesting order* [21]. To consider interesting order means to make use of
the ordering which exists in the inputs to speed-up otherwise expensive operations. An example is a join
which can be quite expensive if executed as a nested loop join, but becomes very efficient, if its inputs are
ordered on the join attribute, because then it can be executed as a merge sort.
Figure 5.5: Interesting order example

Figure 5.6: Join order example

Figure 5.5 shows an example where interesting order should be considered. The black box on the right is much more expensive than the black box on the left, but since its output is ordered by A the join on top might be much cheaper to execute. Therefore the choose should keep both alternatives.

Join Order

Another global effect is join order. The order in which joins are executed plays a significant role for query performance, because one join may reduce the cardinality of the output where another may increase it. Therefore it is important to order joins in a way which leads to lowest total cost. Finding the best order is clearly not a local decision.

Figure 5.6 shows two query plans that produce the same output. From the cardinalities you can see that the plan on the left is almost twice as expensive to execute as the one on the right (assuming cardinalities translate directly to cost).

5.3.3 Choose Pull-Up Planner

The Choose Pull-Up planner is also a bottom-up planner which traverses the query plan node by node. Unlike the Eager Choose planner, this planner considers one global effect: interesting order. This is achieved by keeping all alternatives of a logical choose that provide different configurations related to order and available fragments (see Section 4.4).

This cost-based planner works similarly to the Eager Choose planner except for the handling of a logical choose. To plan a choose, we pull the choose upwards. This allows us to keep several alternatives instead of choosing a single one now.

Figure 5.7 shows the choose pull-up process. From state 1 to state 2, the choose on the lower right is pulled up planning the filter above. Processing the filter naturally costs something. In this example, we assume the cost to process this filter on top of the left black box is 4 Fr., to process it on the right black
Figure 5.7: Choose pull-up
box costs 2 Fr. The order of the right black box, A, is kept while processing the filter. From state 2 to state 3 the two chooses are combined by simply adding all alternatives. The next step, from state 3 to state 4, shows that only the cheapest alternative from several which have the same configuration (same provided fragments and same order) must be retained.

5.3.4 Dynamic Programming Planner

The two query planners shown in Sections 5.3.1 and 5.3.3 are both based on traversing the query plan node by node in a bottom-up fashion. As we have seen they both cannot find the best possible join order, because they do not change the structure of the query plan. A completely different approach to finish query planning is plan enumeration.

Many plan enumeration algorithms have been proposed in the literature. The most popular one is dynamic programming. This algorithm was first used in IBM’s System R project [21]. A good description of the dynamic programming algorithm for query planning can be found in [17].

Dynamic programming can be used to find an optimal query plan in terms of cost. It automatically considers interesting orders and join orders. Its only problem is that it is exponential in time and space complexity. To circumvent this problem, an extension known as iterative dynamic programming was proposed in [18].
Chapter 6

Bulk Processing Resource Views

Executing a query in general means to fetch data, process it, and provide a result to the client. To process the data, most state-of-the-art query execution engines (see [17] for a good overview) are based on the iterator model [12]. In such a model, operators are implemented as iterators. Since all iterators have the same interface, all operators can be plugged together.

iMeMex also uses the iterator model and resource views are the basic processing unit. Operators therefore process one resource view after another. Since iMeMex is implemented in Java, which is an object-oriented language, each resource view is an object. An operator calls a function like `next()` to get the next resource view to process.

In this short chapter we present a first approach to process resource views in bulk. Section 6.1 shows reasons why bulk processing resource views might be useful. In Section 6.2 the design and implementation is explained. The experiments we performed are presented in Section 6.3 and Section 6.4 gives a conclusion.

6.1 Motivation for Bulk Processing

The basic iterator model used in an object-oriented language requires to have one object per processing unit. In iMeMex this means that we have to instantiate one object per resource view which we want to process. If we are working on large data sets, millions of resource views have to be processed and therefore millions of objects have to be instantiated.

If creating an object was for free, this would not be a problem, but since creating an object requires time (CPU) and memory, instantiating one object per resource view could become a bottleneck. One possible solution to this is to use a larger processing unit which is not as fine granular as a single resource view, but a group of resource views.

Besides requiring one object per resource view, the basic iterator model also requires us to execute one function call for each next resource view. Again when having millions of resource views to process, one function call per resource view might already be too expensive.

6.2 Design and Implementation

As explained in the motivation, an object grouping several resource views together could be used as a coarser granular processing unit. This approach has the nice property, that we still can use the iterator model to implement our operators. The only change is that the processing unit is no longer a single resource view, but a group of resource views.

In this section we first look at one possible implementation of a group of resource views, OID arrays. Then we show operators that are able to process such arrays.
6.2.1 OID Array

Grouping several resource views together could be done in a java.util.List, but doing so would neither reduce the number of objects nor the number of function calls to get the objects. An array on the other hand is a single object and no function calls have to be executed to get the data stored in the array.

An array in Java could store objects, but since we want to avoid having one object per resource view, we could store primitive types in the array. A java.util.List does not allow this.

If we want to store primitive types in the array, we have to find a mapping from resource view to primitive types. In iMeMex every resource view which is processed above the resource view manager, has a unique object identifier (OID). This OID is a long, which is a primitive type, and can therefore be used for our purposes.

The data structure we use to group several resource views is an OIDArray. Its implementation is shown in Listing 6.1.

```java
public class OIDArray {

    /* data storage/transport area (implicit capacity) */
    public long[] oids;

    /* number of elements which can be put into this OID array */
    public final int capacity;

    /* current reading position */
    public int readPos;

    /* current writing position */
    public int writePos;

    /**
     * Constructor.
     * @param capacity capacity to use for this array.
     */
    public OIDArray(int capacity) {
        this.capacity = capacity;
        oids = new long[capacity];
        readPos = 0;
        writePos = 0;
    }
}
```

Listing 6.1: OIDArray Class

As you can see an OIDArray only stores an OID for each resource view. Therefore operators which need other data, e.g. the name or the tuple component, to be evaluated would have to fetch this data. Fetching the data per resource view would be at least as big an overhead as processing each resource view as an object.

A possible extension of an OIDArray could also store several other arrays holding more data about a resource view.

6.2.2 OIDArray Operators

Given OID arrays to process groups of resource views, as shown in Section 6.2.1, restricts the possible operators that can be executed without having to lookup additional data. Only operators which can be executed by working with just OIDs can be used.

We have implemented the following operators:
6.3 Experiments

In the following experiments we want to find the impact on query execution performance by varying the capacity of the used OID arrays.

Our expectations are the following: Executing a query using an OID array which can store a single OID (capacity=1) should have a similar execution time as using the basic iterator model. If we increase the capacity of the used OID arrays, execution time should go down.

We executed the query plans shown in Figure 6.1. \( n \) is the number of resource views produced. Experiment 4 uses two times the query plan shown in experiment 3. We used the following hardware: Lenovo Laptop T60, Intel Core Duo 2.0 GHz processor, 2 MB cache, 1 GB main memory. The size of the generated input, the number of resource views, was \( n := 2^{19} = 524288 \). The results of the experiments are shown in Figures 6.2 to 6.5.

- **Merge**: Merge any number of inputs. The output is ordered by increasing OID if all inputs are ordered by increasing OID. That is the order is preserved while merging.

- **Concatenate**: Concatenate any number of inputs. This is similar to the merge operator with the slight difference, that it does not create an ordered output even if the inputs are ordered.

- **Forward**: Forward a single input. This does the same as the concatenate operator with a single input, but its implementation does not have to consider multiple inputs and is therefore optimized.

- **Intersect**: Intersect two inputs. This implementation assumes that the inputs are ordered in increasing order.

- **Generator**: Artificially generate OID arrays. This input operator is used for the experiments shown in Section 6.3.

The implementation of these operators is done such that processing an OID array can be performed without function calls or instantiating new objects. Therefore the number of function calls and object instantiations is linear to the number of OID arrays to process. This is important during the following experiments, because we want to vary the capacity of an OID array and measure its impact.

![Figure 6.1: Query plans used for experiments](image-url)

- **Experiment 1**
- **Experiment 2**
- **Experiment 3**
- **Experiment 4**
Figure 6.2: Experiment 1

Figure 6.3: Experiment 2

Figure 6.4: Experiment 3

Figure 6.5: Experiment 4
6.4 Conclusion

Bulk processing resource views, that is processing several resource views together, is feasible as we have seen from the experiments in Section 6.3. Even though it provides a small benefit in terms of execution time, we did not further investigate it. Using OID arrays was only able to process the input two to three times faster than processing resource views one after another. Creating large amounts of objects and performing many function calls does not hurt performance as much as we expected. Java seems to do a very good job here. It is also important to notice that the time needed to process resource views often is small compared to reading the input. Methods to increase processing speed, like OID arrays, only make sense if reading the input is fast and large amounts of resource views have to be considered. An example for such a scenario is streaming sensor data.

The increase in performance, even though it is not that large, could be considered. The materialized view manager 3.4 would be a good starting point to further investigate bulk processing. In the materialized view manager we directly read OIDs from disk and therefore have good control over what is happening at lower levels.
Chapter 7

Future Work

In this chapter we present open questions related to query optimization and show possible extensions to iMeMex, which are related to query processing and this Master’s thesis in general. Query optimization has already been studied for a long time (especially for relational databases). The dataspace abstraction introduced in [8] opens new research challenges.

7.1 Merging Alternative Query Plans

After pushing down a query the returned physiological query plans have to be merged. An unsolved problem appears when a lower query planner wants to return several (maybe structurally changed) query plans. Which or how many query plans should be returned?

If we allow lower query planners to return structurally changed query plans, then the algorithm presented in Section 4.3.6 will have to consider that. A possible solution would be to merge the returned query plans based on query containment. Such a merge algorithm would be much more expensive to execute than the presented algorithm, because query containment is expensive to use. A structurally changed query plan is shown in Figure 7.1.

7.2 Estimating Cardinalities

In many cost models the cost to process a single element is known and the total cost is calculated by multiplying the cost to process a single element times the number of elements:

\[ TC := C_1 \times n \] (7.1)

Cardinalities are therefore essential to estimate the total cost to execute an operator. If cardinalities are not available in a high-level query planner, then statistics could be used instead.

A better approach would be to make sure cardinalities are always computed in low-level query planners. Since low-level query planners are close to the physical data it is their main responsibility to get good cardinality estimates. If cardinalities are badly calculated at this level, cost estimation becomes very difficult in higher levels of the query plan. Whenever possible, information provided by lower level query planners should be exploited, otherwise statistics could be kept.

7.2.1 Estimating Cardinalities of Unnest Operators

Our unnest operators, shallow- and deep unnest, produce resource views which do not appear in their input. Output cardinalities can therefore not be derived from input cardinalities. To estimate output cardinalities either statistics or an index on the group component could be used to provide this information.
7.3 Estimating Selectivities

Selectivities are needed to estimate the number of results produced by operators which reduce or filter their input. A few example operators are: Distinct, Filter, Intersect, Join.

Low-level query planners which want to perform more than just returning scans over their data need to know selectivities. Since they are close to the physical data, they could try to keep extensive information about the data. Statistics or Histograms would be an option. If a low-level query planner uses a library to access the data, then whenever possible, provided selectivities (or statistics in general) should be reused. For example a full-text index using Lucene should try to use selectivities which are provided by Lucene. If no additional information about the data is provided by a library, then histograms or statistics could be kept.

After the low-level query planners have planned as much as they could, the high-level query planner has to merge the returned physiological query plans and finish the physical planning. To do so, estimating selectivities (e.g. in a filter) might be neccessary. Currently a static selectivity factor of 0.1 is used. Statistics or histograms on this level could be an alternative to find better selectivities depending of the query that is being executed. Since each query planner is responsible itself for providing good estimates, better estimates could be incrementally provided.

7.4 Join and Intersect Order

The order in which joins are executed is important to consider during physical query planning. For instance, reducing joins should be executed before expensive, less reducing joins. Currently neither the Eager Choose Planner 5.3.1, nor the Choose Pull-Up Planner 5.3.3 consider join order. The best join order could be found with dynamic programming. See [17] for possible implementations.

7.5 Queries Spanning Multiple Data Sources

Joins spanning multiple data sources are currently executed above the data sources. This has the disadvantage that a huge amount of data has to be transfered from the data sources to iMeMex. Since a join will filter the received data, it would make sense to pushdown that filter. This approach is called bind join and is described in [14].
7.6 Physical Choose

While executing a query in a distributed system, data sources might breakdown or one part of the system might suddenly be disconnected. It would be nice if the query processor could react to such events and choose a different alternative query plan which does not rely on the system component that failed. A physical choose [13] would keep several alternative query plans which can be exchanged at runtime. Another usage of a physical choose would be to use parallelism during execution of a query [11]. Several alternative query plans could be executed in parallel and their results be intersected.

The logical choose presented in Section 4.3 pinpoints a place where a physical choose could be inserted. Given an implementation of a physical choose, inserting it in the physical execution plan would be straightforward.

7.7 Physical Fetch

iDM is a lazy data model (see Section 2.1). Therefore data does not have to be present during query execution when resource views are given from one physical operator to the next. A physical fetch could ensure that the data is really available before resource views are passed on.

Physical fetches could be inserted in the physical query plan where a logical fetch 4.4 is in the logical query plan.

7.8 QoS Oriented Query Optimization

A nice feature that could be studied is Quality of Service (QoS) oriented query optimization.

A possible approach is the following: Our black boxes could, besides the information they transport now, provide QoS related information like precision and recall. The lower query planners would then not only return a single black box for some subplan, but many each having different QoS guarantees. The optimizer finally would choose the query plan which comes closest to the QoS requirements of the client.
Chapter 8

Conclusion

Cost-based query optimization is a large field and has been studied for a long time. Especially in the area of relational databases a lot of interesting solutions have been found. Recently new research opportunities related to dataspaces have been discovered.

In this Master’s thesis we further explored this area towards integrating heterogeneous and distributed query planning and query optimization in a service-oriented architecture. A query processing model based on pushing down the query to data sources and indexes has been presented and was implemented in iMeMex. One important building block of this pushdown model are black boxes. A black box is part of a logical query plan and represents a subplan which is already physically planned. These black boxes are introduced in the query plan by different query planners. This in fact allows to distribute query planning and query optimization among several heterogeneous query planners.

This new query processing model calls for algorithms to combine the work performed by different query planners. It also requires query planners which finish the physical planning of a partly planned query plan. During this thesis one algorithm to merge partly planned query plans has been presented. To finish physical query planning of a partly planned query plan, one algorithm has been implemented and the design of a second has been described.

Besides providing a way for different query planners to work together, this thesis also studied how to work with multiple cost models. Since many query planners may participate in planning and optimizing a query, different cost models might be used. To enable cost-based decisions in a system where multiple cost models are used, being able to translate from one cost model to another is essential.

To make this query processing model work, the system architecture of iMeMex had to be changed. Query pushdown is now possible to data sources as well as to indexes and the materialized view manager. During this thesis, the materialized view manager, a new powerful data source plugin and one index which spans several components of a resource view have been added to iMeMex. In the course of this thesis we have also continued the work to use a service oriented architecture based on OSGi. This now allows to deploy iMeMex in different deployment scenarios which range from mobile phones to workgroup servers.

These changes to the iMeMex system architecture not only allow to experiment with a new query processing model, but also prepare an interesting basis for further research topics.

During this thesis a first approach to bulk processing resource views has been studied and a few experiments have been run. The experiments showed a possible gain of factor two to three in query execution performance. Further research in this direction should be done.
Chapter 9

Acknowledgments

Many thanks:
First to my supervisors Dr. Jens-Peter Dittrich, Marcos Antonio Vaz Salles, and Shant Kirakos Karakashian for reading very rough drafts of this thesis and all their support and discussions.
To Olivier René Girard for his Master’s thesis which provided a nice basis for the layout of this text. To the DBIS group for its pleasant working atmosphere. And finally to my family and friends for being around.
Appendix A

Joins in iMeMex

A.1 Joins in General

In this section, we present generally known join types. The most well known joins and join-like operators are shown in Figure A.1. An example of a natural join is given in Figure A.2.

![Figure A.1: Join types](image1)

<table>
<thead>
<tr>
<th>Employee</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>EmpId</td>
</tr>
<tr>
<td>Jens</td>
<td>1234</td>
</tr>
<tr>
<td>Marcos</td>
<td>1235</td>
</tr>
<tr>
<td>Shant</td>
<td>1236</td>
</tr>
<tr>
<td>Lukas</td>
<td>1237</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee ⋈ Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Jens</td>
</tr>
<tr>
<td>Marcos</td>
</tr>
<tr>
<td>Shant</td>
</tr>
<tr>
<td>Lukas</td>
</tr>
</tbody>
</table>

![Figure A.2: Natural join example](image2)
A.2 Joins in iMeMex

In this section, we look at how iMeMex supports joins. iMeMex supports four join types:

- inner join
- left outer join
- right outer join
- full outer join

The inner join is a $\theta$-join and therefore covers many other join types. A *Natural join* for example can be expressed as $\theta$-join. An *equijoin* is a $\theta$-join where $\theta$ is “$=$”. A *Semijoin* is a projected natural join. And finally an *Antijoin* is the complement of a semijoin.
Appendix B

Alternative Merge Simplifiers

B.1 Unneeded Unresolved Alternative Remover

Listing B.1 shows the implementation of the “unneeded unresolved alternative remover”, which is used as the first step during alternative merge simplification.

B.2 Trivial Choose Remover

Listing B.2 shows the implementation of the “trivial choose remover”, which is used as the second step during alternative merge simplification.
private boolean removeUnneededUnresolvedAlternatives(
    LogicalNode lognode) {

    // if no children are left, return true if leave is resolved
    if (lognode.nrOfChildren() == 0)
        return lognode.getType() == LogicalNodeType.BLACK_BOX;

    // recursively loop over ALL children (no early break)
    // and test if they resolve.
    // a logical node resolves, if all its children resolve
    // or if a single child can be resolved and the node's type is choose
    boolean childrenResolvedByChoose = false;
    List<LogicalNode> unresolvedChildren = new ArrayList<LogicalNode>();
    for (LogicalNode child : lognode.getChildren()) {
        boolean childResolved = removeNotNeededUnresolvedPossibilities(child);
        if (childResolved) {
            if (lognode.getType() == LogicalNodeType.CHOOSE)
                childrenResolvedByChoose = true;
            }
        }
    }

    if (childrenResolvedByChoose) {
        // if at least one child is resolved
        // and this logical node's type is choose
        // then remove unneeded unresolved children
        for (LogicalNode unresolvedChild : unresolvedChildren)
            lognode.remove(unresolvedChild);
        return true;
    } else {
        if (unresolvedChildren.size() == 0) {
            // if all children are resolved, keep all and return true
            return true;
        } else {
            // if some children are NOT resolved
            // and the type of this logical node is not choose,
            // then this logical node can NOT be resolved either
            return false;
        }
    }
}

Listing B.1: Unneeded unresolved alternative remover
public LogicalNode processTree(LogicalNode lognode) {

    // remove choose with only a single output
    if (lognode.getType() == LogicalNodeType.CHOOSE
        && lognode.nr0fChildren() == 1) {
        lognode = lognode.getChild(0);
    }

    // loop over children (getChildren() returns a shallow copy)
    for (LogicalNode child : lognode.getChildren()) {
        lognode.replace(child, processTree(child));
    }

    return lognode;
}

Listing B.2: Trivial choose remover
Appendix C

Fetch Pushdown and Pull-Up

C.1 Fetch Pushdown

Listing C.1 shows the first part of the implementation of the “fetch pushdown”, which is the first phase of removing unneeded fetches. This listing shows, what we do, if we encounter a logical node which is not of type fetch. Listing C.2 shows, what we do if we encounter a logical fetch.

```java
public LogicalNode processTree(LogicalNode node) {
    if (node.getType() == LogicalNodeType.FETCH) {
        // (handle logical fetch)
    } else {

        // introduce required fragments if needed
        Set<String> requiredFragments = getRequiredFragments(node);
        int numChildren = node.nrOfChildren();
        if (requiredFragments != null) {
            for (int i = 0; i < numChildren; i++) {
                LogicalNode child = node.getChild(i);
                Fetch newFetch = new Fetch(child,
                                       new HashSet<String>(requiredFragments));
                node.replace(child, newFetch);
            }
        }

        // recursively process children
        for (int i = 0; i < numChildren; i++) {
            LogicalNode child = node.getChild(i);
            node.replace(child, processTree(child));
        }
    }
    return node;
}
```

Listing C.1: Fetch pushdown - not fetch type
```java
public LogicalNode processTree(LogicalNode node) {
    if (node.getType() == LogicalNodeType.FETCH) {
        Fetch fetch = (Fetch) node;
        // get single child of fetch
        LogicalNode child = fetch.getChild();

        if (child.getType() == LogicalNodeType.BLACK_BOX) {
            BlackBox bb = (BlackBox) child;
            // remove all fragments from the fetch which are provided by
            // the black box
            Set<String> fragments = fetch.getFragments();
            fragments.removeAll(bb.getFragments());
            // if all fragments of fetch are covered by blackbox,
            // return the black box only (that is remove fetch)
            if (fragments.size() == 0) {
                return bb;
            } else {
                return fetch;
            }
        } else if (child.getType() == LogicalNodeType.SHALLOW_UNNEST
                   || child.getType() == LogicalNodeType.DEEP_UNNEST) {
            // don’t push fetch through shallow- or deep-unnest
            // still recurse, but return fetch
            processTree(child);
            return fetch;
        } else {
            // swallow fetches and required fragments found on the way down
            Set<String> requiredFragments = getRequiredFragments(child);
            if (requiredFragments != null) {
                fetch.getFragments().addAll(requiredFragments);
            }
            // push fetch down into each branch, that is
            // loop over all grand children
            int numChildren = child.nrOfChildren();
            for (int i = 0; i < numChildren; i++) {
                LogicalNode grandchild = child.getChild(i);
                // copy fragments and add to new grand children
                Set<String> copiedFragments =
                        new HashSet<String>(fetch.getFragments());
                Fetch newFetch = new Fetch(grandchild, copiedFragments);
                child.replace(grandchild, newFetch);
            }
            // and recurse
            return processTree(child);
        } else {
            .. (handle non-fetch type)
        }
    }
}
```

Listing C.2: Fetch pushdown - fetch type
C.2 Fetch Pull-Up

Listing C.3 and C.4 show the implementation of the “fetch pull-up”. Fetch pull-up is performed after fetch pushdown to bring fetches as high as possible in the query plan.

```java
public LogicalNode processTree(LogicalNode node) {
    // if we have no children, we just return ourselves
    int numChildren = node.nrOfChildren();
    if (numChildren == 0) return node;

    // traverse tree from the leaves to the root
    for (int i = 0; i < numChildren; i++) {
        LogicalNode child = node.getChild(i);
        node.replace(child, processTree(child));
    }

    // handle fetch
    if (node.getType() == LogicalNodeType.FETCH) {
        Fetch fetch = (Fetch) node;
        return fetch;
    }

    // find required fragments
    Set<String> requiredFragments = UnneededFetchRemoverHelper.getRequiredFragments(node);

    // find fragments that are fetched from all inputs
    Set<String> fetchedByAllFragments = null;
    for (int i = 0; i < numChildren; i++) {
        LogicalNode child = node.getChild(i);
        if (child.getType() == LogicalNodeType.FETCH) {
            Fetch fetch = (Fetch) child;
            if (fetchedByAllFragments == null) {
                // the first fetch sets the basis
                fetchedByAllFragments = new HashSet<String>();
                fetchedByAllFragments.addAll(fetch.getFragments());
            } else {
                // further fetches constantly drain the set
                fetchedByAllFragments.retainAll(fetch.getFragments());
            }
        } else {
            // not all inputs are fetches, so no fragment fetched by all
            fetchedByAllFragments = null;
            break;
        }
    }
}
```

Listing C.3: Fetch pull-up - part 1
(see first page)

// loop over all children
Set<String> fragmentsToPullUpAll = new HashSet<String>;
for (int i = 0; i < numChildren; i++) {
    LogicalNode child = node.getChild(i);
    if (child.getType() == LogicalNodeType.FETCH) {
        Fetch fetch = (Fetch) child;

        // collections for fragments which can be removed/pulled-up
        Set<String> fragmentsToRemove = new HashSet<String>;
        Set<String> fragmentsToPullUp = new HashSet<String>;

        // loop over all fragments
        for (String fragment : fetch.getFragments()) {
            // 1st decision of decision tree
            if(requiredFragments != null && requiredFragments.contains(fragment)){
                // stay
            } else {
                // 2nd decision of decision tree
                if (fetchedByAllFragments != null && fetchedByAllFragments.contains(fragment)) {
                    // decision 3.1 of decision tree
                    if (doesIntroduceNewResults(node)) {
                        // remove
                        fragmentsToRemove.add(fragment);
                    } else {
                        // pull-up
                        fragmentsToPullUp.add(fragment);
                    }
                } else {
                    // decision 3.2 of decision tree
                    if (outputRequired(node)) {} // stay
                    else fragmentsToRemove.add(fragment); // remove
                }
            }
        }

        // remove fragments from this fetch
        fetch.getFragments().removeAll(fragmentsToRemove);
        fetch.getFragments().removeAll(fragmentsToPullUp);

        // add pulled-up fragments of this fetch to collected set
        fragmentsToPullUpAll.addAll(fragmentsToPullUp);

        // remove fetch if no fragments left
        if(fetch.getFragments().size() == 0) node.replace(fetch, fetch.getChild());
    }
}

// introduce fetch on top of this node, // if there are fragments to pull up
if (fragmentsToPullUpAll.size() > 0) {
    Fetch newFetch = new Fetch(node, fragmentsToPullUpAll);
    return newFetch;
} else return node;

Listing C.4: Fetch pull-up - part 2
Bibliography


