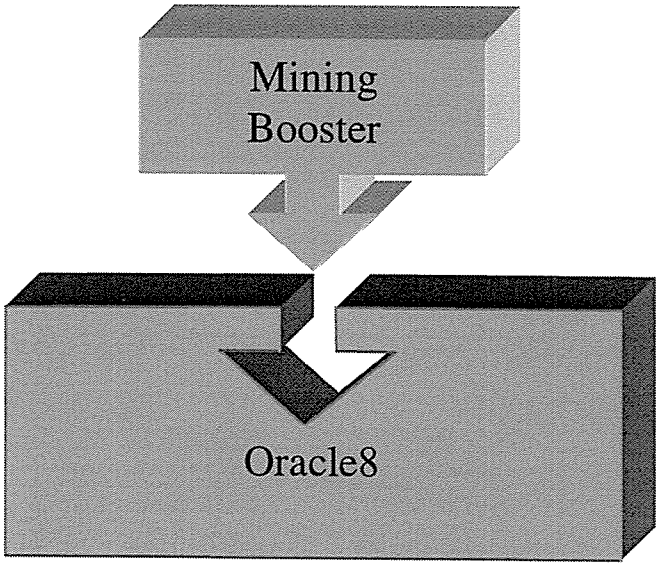
Project Report

Data Distilleries



Arjan van Muyen
Hio-Enschede

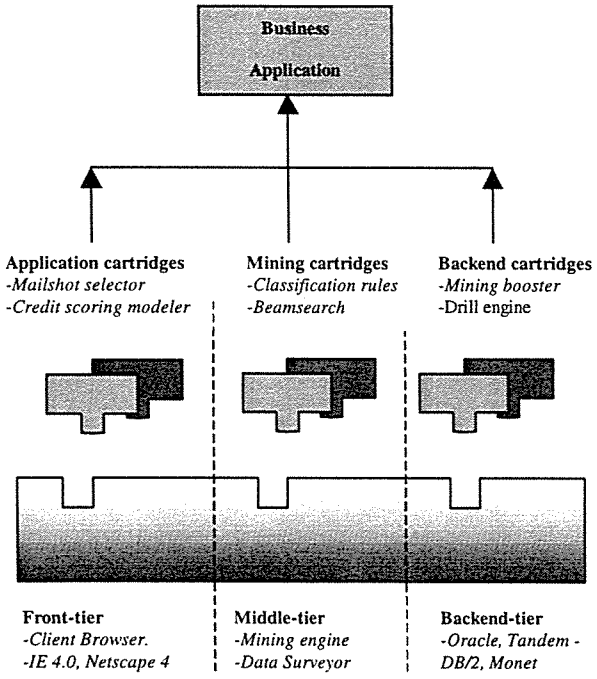
Company Confidential*

15/06/98

1. Introduction

Data mining applications cause very heavy and very specific query loads on a DBMS, that can be characterized as consisting of many small queries, involving full table scans, that zoom in into each time smaller subsets of objects, and use only a few attributes from a wide and deep mining table. Various vendors have tried to extend OLTP - databases to handle this type of query load. Multidimensional databases, pre-computed aggregations, SQL-Extensions like the Cube-operator, are attempts to boost the OLTP databases for this type of querying.

Researchers at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam have developed a totally different approach that is able to handle the data mining query load efficiently by using vertical table fragmentation and algorithms optimized for main memory execution. These techniques are implemented in a main-memory DBMS called Monet. Data Distilleries, a spin-off company of the CWI, develops a client-server data mining tool, Data Surveyor (DS), which offers a variety of powerful algorithms for interactive data mining on very large data sets.



The tool is designed in a three-tier architecture :

1. The front end cartridges, a 100% pure Java graphical user interface.
2. The middle layer, which implements the various pluggable mining-algorithms in a mining-engine
3. The backend, the Monet database system or any other SQL-speaking DBMS.

A business application consists of one cartridge out of each layer.

Many companies store its customer data inside the Oracle-DBMS. Currently, the data has to be duplicated towards Monet before the mining session with DS can start. This is not compliant with the major advantage of databases : central storage of data. To eliminate this nagging problem, Data Distilleries has launched the Mining Booster project.

1.1 Mining Booster

The main goal of the Mining Booster (MB) is to achieve a Monet-like performance on Oracle8. By using the Oracle data-cartridge mechanism, a kind of search accelerator structure called the Data Vector (DV) is introduced to Oracle. These data vectors can be created on any table column in Oracle, and have special operators defined on them for data cube construction and aggregation, that can be computed efficiently. This efficiency is gained in much the same way as Monet operations achieve efficiency. In fact, a data vector corresponds 1-1 with a Binary Association Table (BAT), the vertical fragment of Monet.

1.1.1 Objectives Mining Booster

The objectives to be met by the MB cartridge are:

- Performance
  the cartridge should enable Monet-like performance on Data Mining tasks like the DD Benchmark.
- Openness
  the cartridge should be reusable by data mining or OLAP tools other than Data Surveyor.
- Orthogonality
  it should be possible to apply the functions in the cartridge to any kind of Oracle table, containing any kind of data.
- Self-contained
  it should be easy to integrate the Booster cartridge in an Oracle system used for data warehousing. This means that no a priori requirements should be placed on the Oracle schema.
- Data Warehouse integration
  Changes in the data warehouse should not affect the mining booster functionality. This means that that it should be able to cope with updates occurring on the Oracle tables.

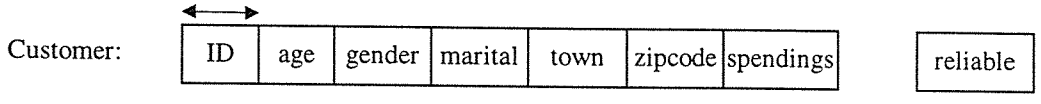
Technically, all this should be made possible by employing the new object-relational features of Oracle8.

1.2 DD Benchmark

To measure how well a DBMS is able to support data mining, Data Distilleries developed the Drill Down Benchmark (DDB). The DDB is a typical data mining-session with Data Surveyor in a real business case :

> *A mail-order warehouse company wants to optimize its mailing activities by using database marketing. In particular, it wants to make special offers to reliable customers. The company heard about good results achieved with data mining, and decided to do a pilot-project where it wants to develop a general profile of* ***reliable*** *customers.*

Goal of this project is to find out which client characteristics indicate a higher probability of becoming a reliable customer. The company owns a data warehouse with 100 attributes per customer, and wants to predict the value of the reliable-attribute from -combinations of- the six attributes listed below:



The DDB is distributed as an ASCII-dump of data, and a set of queries.

When distributing the benchmark to database vendors one does not want to distribute 1-4 G (needed for the most extreme performancetest) of bulk data. Therefore, the distributed bulk data consists of one table of 1 M tuples of seven attributes. The standard width of the mining table used in the buisinesscase is 100 attributes wide. These 100 attributes are obtained by horizontal concatenating the 6 attributes listed above: Hence, we obtain columns like age_0, age_1, age_2, .. etc..

1.2.1 Scaling

To enable the DDB to serve as a benchmark, the sizing of the mining table has to be formalized as well. This is achieved by introducing horizontal (*HF*) and vertical (*VF*) scaling factors. *HF (in {1, 10, 100})* applies to the number of columns, *VF (in {1, 10, 100, 1000}) applies* to the number of rows. Applying the scaling factors results in mining tables from 100 to 10.000 attributes and cardinalities from 1M to 1 G. Applying *HF* also multiplies the number of attributes taken into account by the same factor *HF*.

The DDB, with *HF* set to one, results in a 133 queries-of-interest generated by the DS-mining engine. How well a DBMS backend is able to support data mining is then expressed as a timescore on the DDB.

1.2.2 Implementations

The DDB was implemented for the Monet database and an anonymous RDBMS product. The testplatform was a Sun Ultra2 Creator 3D machine with :

- two 168 MHz CPU's
- 512 Mb of main memory
- 16 MB of L2 cache
- 12 MB/s real disk throughput.

Timescores are listed in the table below :

Summarized Results of the DD Benchmark

database size			platform		elapsed time					
	HF	VF	DBMS	#cpus	total	B_0	B_1	B_2	B_3	B_4
small (400MB)	1	1	relational	2	22m35s	1m22s	1m28s	7m48s	5m47s	3m9s
			monet	2	39s	4.6s	4.7s	17.9s	9.1s	2.3s
				1	1m10s	9.1s	8.9s	31.1s	17.0s	4.1s
big (4GB)	1	10	monet	2	8m32s	50.0s	54.0s	4m23s	1m52s	33s
				1	13m58s	1m27s	1m38s	6m43s	3m14s	57s
wide (4GB)	10	1	monet	2	8m46s	1m5s	53.0s	3m41	2m29s	37s
				1	15m25s	1m39s	1m38s	6m33s	4m29s	1m6s

2. Project Description

My project is to participate in the Mining Booster (MB) -project by designing the interface of the MB. As a data cartridge is defined in PL/SQL, implementing the MB interface will consist of expressing the data vector and its operations as a set of classes in PL/SQL, using Oracles object-relational features. Functional and technical specifications have to be defined and discussed.

Milestone in this project is to run the DD benchmark on Oracle8, and to achieve a Monet-like performance.

The following objectives of the overall Mining Booster project are covered in the interface:

- Performance
  The cartridge should enable Monet-like performance on Data Mining tasks like the DD Benchmark.

- Openness
  The cartridge should be reusable by data mining or OLAP tools other than Data Surveyor.

- Orthogonality
  It should be possible to apply the functions in the cartridge to any kind of Oracle table, containing any kind of data.

Implementations coming from my project are :

- An set of classes in PL/SQL, with member functions that enables the construction of data cubes using data vectors.
- A communications cartridge, in PL/SQL, that handles the communications between the Monet server and ORACLE

Overall position in the project:

Marco Fuykschot, an engineer sent on secondment from SuperFluid, is currently working on the implementation of the data vector. He is, supervised by Peter Boncz (Core backend developer at DD), designing the Oracle-representation of the data vector, optimizing storage-capacity in Oracle, minimizing memory usage, and the fastest way of transporting the data vectors towards the Monet-server. Most of his work is done in C and SQL. I will refer to this part of the MB as the MB engine. My task is to connect to this engine, invoke the commands that build the specified DVS, and generate optimal Monet-statements to process the DVS. We work separately, and will integrate out work at the end of the project

3. Context

3.1 Data mining

Data mining is a technique to discover strategic information hidden in very large databases. This information, e.g. trends and patterns, can be used to improve business decisions. The last few years data-mining has been used in an experimental setting, now we are at a stage it can be used on a day to day basis to support business professionals. Nowadays, most companies have to deal with fast moving and competitive markets. Customers preferences change easily and competitors are always there to look for better products, new niches and other ways to improve their marketshare.

For these companies, the crucial weapon is knowledge. Knowing more about the outside world, being able to detect trends earlier and to better predict customer's behavior are crucial for survival.

One of the ways to learn about the outside world is to analyze the wealth of data that is already available within companies. Companies collect a lot of data about the outside world, e.g. information about their customers, the products they buy and how they pay for these products. All this information is stored in large databases. Analyzing these data will learn them a lot and let them take actions to take full competitive advantage of this knowledge.

3.1.1 Data Warehousing

Customer data, product data and transaction data are stored in so-called production databases. Generally, this data is stored in a form very well suited for daily business, but not suited for analysis. For analysis, we often want to combine data from different sources, e.g. combine customer data with externally available data such as geodemographic data. Therefore, special databases are constructed called data warehouses, where data from various sources is combined, cleaned and stored in a form suited for analysis. A data warehouse is simply a database for analysis. These data warehouses may store very large amounts of detail data, up to Terabytes (thousand billion bytes).

3.1.2 Analyzing data

Building a data warehouse in itself is useless, as long as users don't have the powerful tools needed to analyze this data warehouse. In Data Distilleries' vision, not all business questions can be solved by one and the same tool. So rather than a single tool, the user should have a variety of tools, all geared towards particular information requirements. Different analysis techniques have been developed during recent years for these requirements:

1. quantify the data
2. explain behavior
3. discover hidden knowledge
4. predict behavior

3.1.2.1 quantify the data

Users often want quantitative information from their data, such as

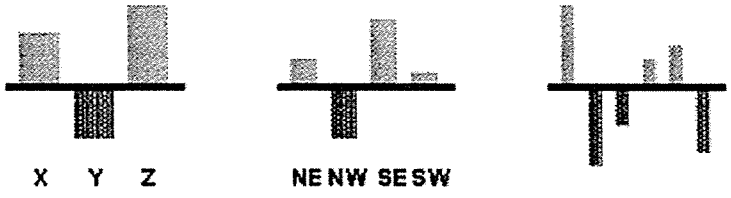
- how many products did I sell last month?
- how many customers reside in New York?

If this information need is stable over time, report generators can be used to generate a standard report for each month. Sometimes however, users need to have more flexibility to get answer to ad hoc questions. Here, standard query tools are used. These tools are also known as managed report or query environments.

3.1.2.2 Explain behavior

Sometimes, users want to have a better understanding of observed anomalies or behavior in their data. For example, if sales figures drop for a particular product Y (figure a), the user may want to find an explanation for this behavior. Knowing the explanation allows users to take appropriate actions. OLAP (On Line Analytical Processing) tools are user-friendly and flexible tools that allow users to interactively browse through the data. In the above example, the user may discover the reason for the dropped sales figure by looking at the data from different dimensions.

For example, the user may look at the sales figures for product Y for the different regions, and find out that say North-Western region is mainly responsible for the decreased sales (figure b). Next, the user can perform a drill down: zoom in on the North-Western region and find out which states contribute most to the dropped sales (figure c)



(a) sales for product x, y & z

(b) y for different regions

(c) sales for y for NW region

3.1.2.3 Discover hidden knowledge

The above techniques, how useful they may be, have one serious drawback. They expect the user to have suspicions of the knowledge hidden in the database. The above techniques test only the user's assumptions. In the above OLAP example, the user assumed that fallback of product X is caused by differences in sales in states.

Data warehouses combine data from various sources and therefore contain a lot of hidden knowledge that is not known to the user. For example, if you combine your customer data with geodemographic data, there may be a very interesting relation between your customer's willingness to donate to charity and their creditworthiness. Since this is an unexpected relation, you would probably not have formulated this as an explicit query. So you would never have found this useful knowledge.

Data mining allows the user to discover hidden information in large databases. Rather than relying on the user to formulate hypotheses, a data mining tool formulates ten thousands of these hypotheses itself, verifies them on the database, and returns the most interesting ones to the user. A data mining tool may discover relations and trends such as:

"young men driving expensive red leased cars have 27% probability of getting a car accident, which is significantly more than the average insurgent"

"85% of product X is sold to unmarried women, aged 45 - 55, living in suburbs and income > 50000"

These results, also known as explanatory or descriptive models, provide insight in e.g. the profiles of customers and allow you to target new niches and develop new products.

3.1.2.4 Predict behavior

The ability to predict future events allows us to anticipate these and make better decisions. For banks, it is very useful to predict whether a new applicant will default on a loan or not. Or for direct marketing, it may be useful to target only those prospects that are very likely to respond to the mailing.

Hence, we need a model that predicts customer behavior. A variety of techniques can be used to build this model, such as

- Neural Networks
- Statistical Techniques
- Data Mining

As described above, data mining techniques are used to discover hidden relationships in data. For example, this can be the relationship between customer characteristics and creditworthiness. Once this relationship has been discovered, it can be used to make predictions for new customers.
For example, the system could discover the following rule:

if age < 24 and gender = male and carprice > 40000 then accident probability is 24%

stating that young male drivers in expensive cars have a high probability of getting an accident.

3.1.3 Data Cubes and Cross-tables

Now, how are these hidden relations and probabilities detected ?

The answer is already mentioned, by aggregating on the desired attributes. Together with the aggregate function (count, average, min, max etc) so-called data cubes are constructed. Aggregating divides the data into groups with the same value. Let's for now assume the aggregate function is the count, and go back to the running example of the DD Benchmark, then we can count how many customers are 'male' and reliable. We also can count how many are 'male' and not reliable. From these two we can construct a probability of a male being reliable. We could also do this for all males living in Amsterdam. Then we compose a restricted data cube.

.

A grouping on two attributes (i.e. find all distinct combinations of these attributes), together with the aggregate function count, is called a Cross-table. When a different aggregate function is used the result is called a data cube. On speaking terms, these two are often considered equal.

	Gender	
Reliable	M	F
Yes	23	234
No	243	123

A data cube on gender and reliable

	Gender, town = Amsterdam	
Reliable	M	F
Yes	23	200
No	32	10

A data cube on gender and reliable, restricted for Amsterdam

3.2 Monet

Monet is a customizable database system developed at CWI and University of Amsterdam, intended to be used as the database backend for widely varying application domains. It is designed to get maximum database performance out of today's workstations and multiprocessor systems. It has already achieved considerable success in supporting a data mining application, and work is well under way in a project where it is used in a high-end GIS application. Monet is a type- and algebra-extensible database system and employs shared memory parallelism.

3.2.1 Data structures

Monet uses the Decomposed Storage Model to store its data : all relations are sliced per column, one column is the basis data element in Monet : the Binary Association Table.
This approach has a number of advantages : First, one column is much smaller in size then a whole table (of e.g. 100 attributes, a typical mining table), therefore it will fit into main memory much easier. Second, as the data is also stored per column on disk, IO is much more efficient compared to row-based storage when accessing only a few attributes. As a third, Monet has even more optimized memory usage by encoding data types to integers, and by mapping cell-ids to positional ids. It is because of these three strategies that Monet can aim for main memory execution.

One column of a relation is represented as a Binary Association Table (BAT) in Monet.
Relevant parts of the data structure are shown below :

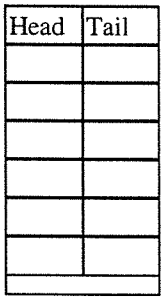


Fig : a BAT in Monet

In real, hash tables, binary search trees and heaps to store and search the data efficiently are also present, but they are not relevant here. Initially, the head-column contains a ID, and the tail column contains the corresponding data. Cells in different BATS are mapped to the same row through their ID.

3.2.2 BAT operations

To illustrate the effect of a number of BAT-operations we take two BATS, representing the column 'Gender' and 'Age' from table 'Customer'. In Monet, operations are expressed in the Monet Interaction Language (MIL). For each operation the MIL-code and its effect is demonstrated.

```
gender := new(oid, chr);
gender.insert (oid(1), 'f');
gender.insert (oid(2), 'm');
gender.insert (oid(3), 'f');
gender.insert (oid(4), 'm');
gender.insert (oid(5), 'f');
gender.insert (oid(6), 'f');
```

gender

1@0	f
2@0	m
3@0	f
4@0	m
5@0	f
6@0	f

OR, using @0-notation

```
age := new(oid,int);
age.insert (1@0, 20);
age.insert (2@0, 22);
age.insert (3@0, 20);
age.insert (4@0, 23);
age.insert (5@0, 24);
age.insert (6@0, 20);
```

age

1@0	20
2@0	22
3@0	20
4@0	23
5@0	24
6@0	20

Note that :

- in MIL-notation group(gender) is equivalent to gender.group()
- 1@0 and oid(1), mean OID 1, just 1 means integer 1
- cells in distinct columns are only related through having the same oid.

➢ Example one: GROUP BY gender

gender

1@0	f
2@0	m
3@0	f
4@0	m
5@0	f
6@0	f

`gender.group()` →

gender_ct

1@0	1@0
2@0	2@0
3@0	1@0
4@0	2@0
5@0	1@0
6@0	1@0

The operator

```
group (bat[oid,any]) return bat[oid,oid]
```

creates for each different value in the tail-column a new group-id. This group-id is the oid of the first occurrence of this value. Every next occurrence of this value is marked as member of this group.
An aggregate like Count (*) is then easily performed with a count of group members.

A listing of all different values can be obtained by the operator:

```
join(bat[any::1,any::2], bat[any::2,any::3]) return bat[any::1,any::3]
```

It takes two input-BATS, joins the head of the first with the tail of the second, and returns a BAT with their joint elements (based on the any::2 - column).

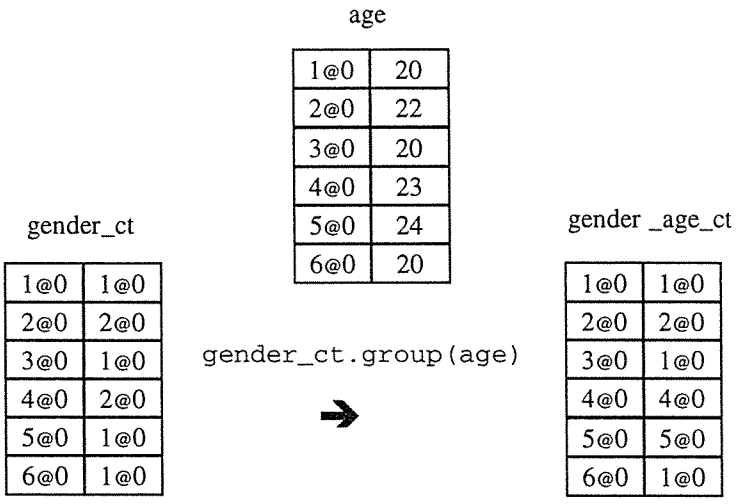
e.g.

```
C.histogram.reverse.join(age)
```

lists all different values with their counts. where

- histogram delivers a bat[oid, int] with the different group-ids and their counts
- reverse swaps the head- and tail column of a BAT

➢ Example two : GROUP BY gender, age



The operator

```
group(bat[oid,oid], bat[oid,any]) return bat[oid,oid]
```

is an overloaded version of the previous group-operator. Now it looks at all different combinations of values in the first and second argument. New group-ids for the discovered combinations are created.

➢ Note the reuse the former computed grouping on gender.

If we want to go back to retrieve all computed combinations and their aggregates, we again need the join-operator : res. with age and gender.

```
like :

histo := gender _age_ct.histogram
```

histo

1@0	3
2@0	1
4@0	1
5@0	1

age

1@0	20
2@0	22
3@0	20
4@0	23
5@0	24
6@0	20

gender

1@0	f
2@0	m
3@0	f
4@0	m
5@0	f
6@0	f

```
m_age:=histo.reverse.join(age);  m_gender:=histo.reverse.join(gender);
```

m_age

3	20
1	24
1	22
1	23

m_gender

3	f
1	m
1	m
1	f

Or the Print-function can print values right away.

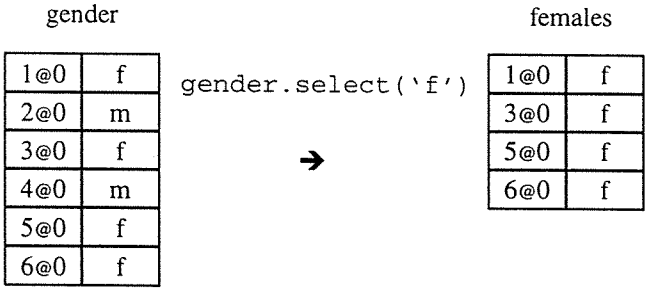
```
like :

gender_ct_age.histogram.print(age, gender)
```

"On screen "

20	f	3
24	m	1
22	m	1
23	f	1

➢ Example three : SELECTION



Note that an equivalent operation would be:

```
females := gender.semijoin (H);
```

with H a BAT like

H

1@0	any
3@0	any
5@0	any
6@0	any

The operator

```
semijoin (bat[any::1,any::2], bat[any::1,any]) return bat[any::1,any::2]
```

returns the original BAT, but only those elements with the same head as the second BAT: i.e. selecting one BAT with another BAT.

Example four: Grouping and selection

age

1@0	20
2@0	22
3@0	20
4@0	23
5@0	24
6@0	20

females

1@0	f
3@0	f
5@0	f
6@0	f

gender_ct

1@0	1@0
2@0	2@0
3@0	1@0
4@0	2@0
5@0	1@0
6@0	1@0

```
gender_ct.subgroup(age, females)
```

➔

fem_gender_age_ct

1@0	1@0
3@0	1@0
5@0	5@0
6@0	1@0

The operator

```
subgroup (bat[oid,oid], bat[oid,any], bat[oid,any]) return bat[oid,oid]
```

Performs two operations in one statement: first `bat[oid,oid]` is semijoined with the last argument, then the previously discussed operation `group(bat, bat)` is performed.

3.2.3 Application interfaces

Monet is an extendible database system. Below we see the complete architecture of Monet All its fromtends use TCP/IP to connect to the server. The interpreter can be extended with modules containing application-specific commands. The MB engine will be integrated with Monet by such an extension-module, containing commands to put- and fetch Oracle data.

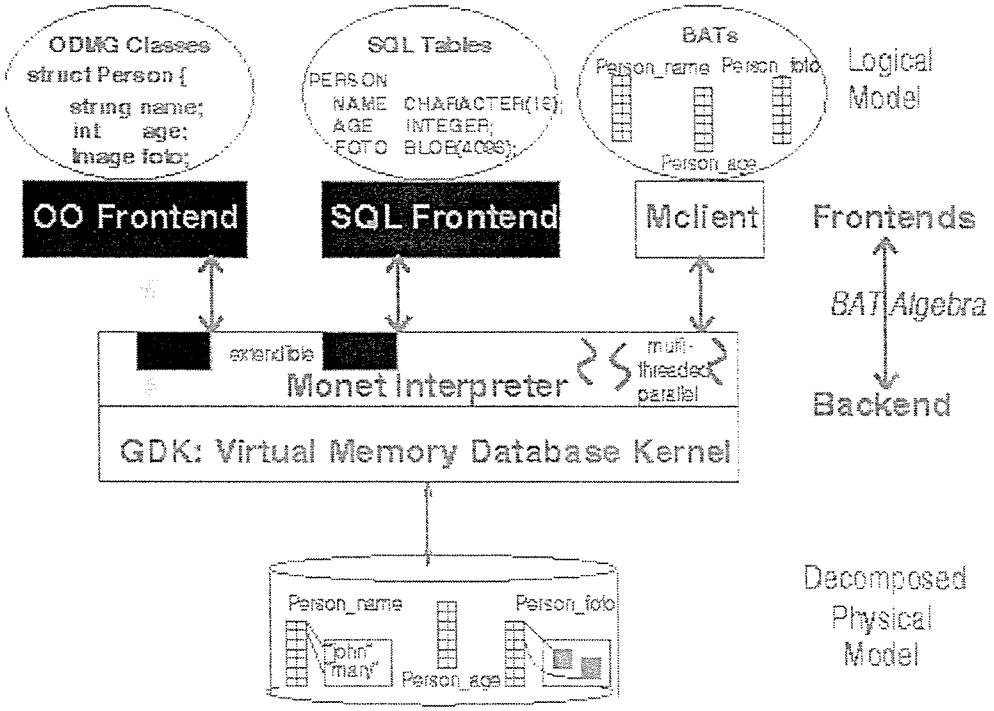


Figure: Monet architecture

3.3 The Oracle 8 Object-relational features

The interface of the MB will use the Object relational features of Oracle8 from PL/SQL. Therefore, it is meaningful to examine these new features to demarcate the toolkit.

3.3.1 Language elements

➢Definition

Objects in Oracle are defined in two steps :

1. define the public methods and attributes of the object
2. define the methods body and additional methods

```
Create type customer as object (
_age integer,
gender char,
MEMBER function age() return integer
);

Create type BODY customer as (
MEMBER function age() return integer is
begin
      return _age;
end;
```

➢Creation & Storage of Objects

In PL/SQL, objects can be created using :

```
DECLARE c Customer;
begin
      c := customer(23, 'm');
end;
```

The object c is valid inside this BEGIN/END scope, the constructor is system-generated and cannot be edited.

Objects must be explicitly stored in Object-Tables :

```
CREATE TABLE Customer_t OF Customer;
```

Or as columns in a regular table :

```
CREATE TABLE  Order_t (
      orderid     integer
      customer    Customer);
```

```
insert into Customer_t  values (c);

-or-

insert into Order_t  values ('E546', customer(20,'m'));
```

Objects stored in an object-table are called *row-objects*. They have a unique, immutable identifier called an *object identifier*. Objects appearing only in table columns or as attributes are called *column-objects*.

➢Collection types

These are special container-types which can hold sets of any user-defined type. We have two types : bounded ad unbounded collections.

1. `CREATE TYPE cust_col_fix as VARRY(20.000) of Customer`
2. `CREATE TYPE cust_col_var as TABLE of Customer`

The first can hold as many elements as specified on declaration, the second can grow 'unlimited'.

➢Comparison

The standard SQL comparison operators (=, <, >, <=, >=) can be overloaded to handle the defined object. Functions have to be implemented to enable Oracle to compare the objects.

➢Sharing objects

A built-in data type, called REF, allows to share an object. A REF is a pointer to an object.

```
CREATE TYPE  Order (
      orderid     integer
      customer    REF Customer);
```

it is possible that the object of a REF becomes unavailable, through deletion of the object or a change in privileges. This situation is called DANGLING and can be checked from SQL (IS DANGLING-predicate)

➢Inheritance

In this release, inheritance is not supported

➢Overloading

Overloading is defined here as distinguishing methods on the basis of their formal and actual parameter list. Functions and procedures with the same name and different parameter-lists are supported.

➢Limitations

- Private attributes are not supported.
- Private methods not are supported
- Creation of REFS is embedded in SQL:
  - `Select REF (p) into cref from customer_t p where p.id=id;`
- This implies that you can only have REF to an object stored in an object-table.
- Dereferencing REFS is also embedded in SQL
  - `Select DEREF(cref) into c from dual;`
- Nested collections are NOT supported. This means that whenever an object type has a collection-attribute, it cannot be stored in a different collection.
- Changes to a stored object are not reflected in the object-table until you use the update-statement.

```
➢ Update Order_t p
      Set set p.customer = cref
      Where p.order_id = 'E246'
```

3.3.2 Application Interfaces

There are various application-interfaces to Oracle : Oracle Call Interface (OCI), RPC from PL/SQL, and Pro*C- and Pro*Cobol- precompilers. Also there is a standard package called DBMS_PIPE for communication through named pipes. Maybe this can also be used to communicate with the Mserver. Regarding performance, only three of all interfaces will be discussed here : OCI, RPC from PL/SQ, and the DBMS_PIPE package.

➢RPC from PL/SQL

From PL/SQL it is possible to invoke procedures implemented in another 3GL language through a Remote Procedure Call (RPC). This procedure has to be stored in an external library. When an external procedure is called, PL/SQL alerts a listener process, which in turn launches a session-specific agent called extproc. The listener hands over the connection to extproc. PL/SQL passes to extproc the name of the shared library, the name of the procedure, and any parameters passed in by the caller. Then extproc loads the shared-library and runs the external procedure. Also, extproc handles service calls, (such as raising an exception) and callbacks to the Oracle-server. Finally, extproc passes to PL/SQL any values returned by the external procedure.

However, as we see, this communication generates a lot of overhead, which will affect performance.

➢DBMS_PIPE - package

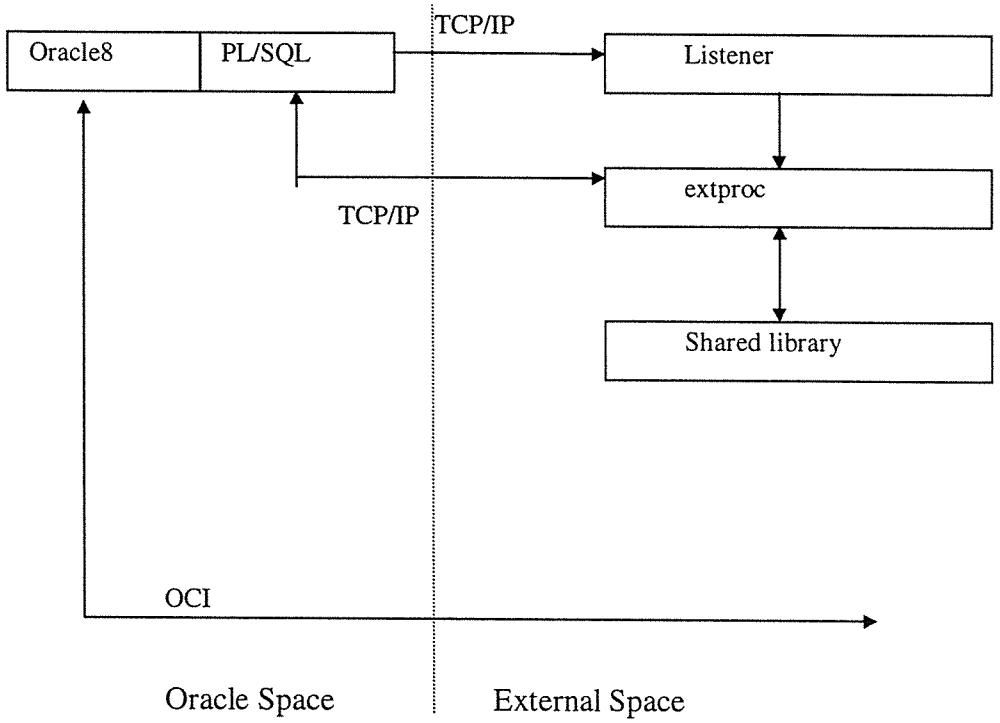
This pre-fabricated package from Oracle allows you to communicate with another process through a named pipe. However, this DBMS-PIPE -package is not implemented using the UNIX named pipe mechanism. The UNIX-Pipe could be used for communication with the Monet server. There could be a workaround (still use the UNIX Pipe) by doing a UNIX-system call, but I did only see a possibility doing this with the PRO*C-interface, not with PL/SQL. From PL/SQL, using the pipe-mechanism would result in an Rpc-call which of course can read and write from a named-pipe, but now we do not avoid the overhead with the extproc agent. Also, Monet has to be extended to handle named pipes.

➢OCI

The OCI is an Application Programming Interface(API) that allows to manipulate data and schemas in an Oracle database from outside Oracle. It is a C-interface, and it is the most direct way to communicate with Oracle : little overhead is generated. It is used for the creation and transport of the data vectors.

3.3.3 Oracle API-Overview

Below, the relevant Oracle API are shown together:



3.3.4 Data cartridges

Creating a data cartridge is using the object-relational features to add a new data type to the database.

4. Analysis

When designing an interface, it is useful to separate between functional behavior of the interface, and technically problems to solve considering the implementation details of the data vectors, Oracle limitations and the communication with the MB engine. Also, The queries of the DD benchmark will be characterized to gain more insight in the desired functionality. Decisions like the actual look of the interface on the level of individual functions and their effect will be made during implementation.

4.1 Functional analysis

4.1.1 Requirements

- Simple
  As a starting observation, I see that it is a big step from the Row-based Thinking (Oracle) to the column-based thinking in Monet. A part of the processing speed of Monet is smart programming in the Monet Interface Language (MIL) which requires a strong developed column-based point of view.

One requirement for dvml is hide unnecessary complications for the Oracle-programmer. To do this, we need in specific cases to design primitives with which you can do several MIL operations in one statement.

4.1.2 DD benchmark

The DDB is executed in 5 batches. After the first 2 batches, Data Surveyor starts interacting and formulating hypothesis (selections) to restrict the maximal width of the beamsearch.

Batch 0:

Basic groupings for each attribute, 6 groupings are computed

Batch 1:

Extend these groupings with a grouping on the target-attribute (reliable)
7 cubes (including aggregations) are computed on 100% of the database

➢ From now, we only deal with data cubes

Batch 2:

DS starts formulating hypothesis as selections on one attribute.

All promising cubes from batch 1 are restricted with the selections and re-computed.
50 cubes (including aggregations) are re-computed on 30% of the database

Batch 3:

First eliminate non-promising subgroups. The hypothesis is now a selection on 2 attributes, an extension (and) of the selections of Batch 2. The selection of Batch 2 is reused and joined with the extra selection. Batch 2 or 1 is not reused, Batch 0 is reused.
40 cubes (including aggregations) are computed on 15% of the database.

Batch 4:

Further extension of selections applied in Batch3. The selection of Batch 3 is reused and joined with the extra selection. CT's from Batch 2 or 1 are not reused, Batch 0 is reused.
30 cubes (including aggregations) are computed, on 8% of the database

Summary

- The groupings from batch 0 are not inspected, but they are reused
- After batch, we only have data cubes
- The ten most promising cubes from every batch are further restricted with selections (hypothesis)
- Only 2-dimensional cubes are constructed
- After the first grouping, the second grouping-attribute is always the same (reliable)
- a cube is never restricted (selected) with a grouping attribute

4.1.3 Functional requirements

From above I would like the following functionalities covered by the interface:

I would like to enable the programmer to:

- Create data vectors
- Construct data cubes
- Construct restricted data cubes
- Specify the aggregate-function used for the cube
- Observe the various aggregate-values of a data cube
- Store computed cubes

Things I would like to hide are

- Aspects considering the MB engine and the Monet-server.
- Oids, voids and enumeration (Monet-specific!)
- The path from a just computed cube [oid, int], to the actual values [any, int] with their aggregates I would like to present a data cube like a regular table.
- optimal processing considerations

4.2 Technical analysis

All actual processing of the data vectors will be done on a hidden Monet server. In Oracle, only optimized storage of the DVS will be implemented. On demand, from the interface, DVS will be created in Oracle and transported to Monet to be processed there. The interface will send MIL-commands that will manipulate the BATS there. These MIL-commands could be composed optimal. On demand, again from the interface, the processed DVS for inspection. Therefore, the mapping between the Oracle DV and Monet-BAT is important. Being able to do the above, administration is required. Monet will be extended with a transport module with functions like 'get_bat' and 'put_bat' to transport the DVS between the two processes. To enable Monet to resolve the grouped BATS ([oid, oid]!) to combinations of values additional administration is required. Another technically item could be ' Multi-batch query optimization '. This is complex optimization that relies on re-use of former computed cubes, over multiple batches.

4.2.1 Needs for Administration

- Creation of internal data vectors.

DVS are created on demand, and we want this to be transparent for the end user. This requires an administration of available internal DVS. This administration should be hidden behind the interface.

- Creation of interface-objects.

The objects created in the interface should be stored centrally as well, to know what we have, and to be able to share the objects.

- Path to a cube.

In MIL a BAT is meaningless if you do not know which operations are performed on it. Hence, all operations on a DV should be administrated in the interface to be passed to Monet the moment we want to store the cube.

4.2.2 Optimal processing

First we will show how DS delivers its hypothesis to MB interface, and what optimizations can be achieved here.

For example, suppose DS delivers the following request for the MB interface:

```
DS: Dear Mining booster, Please calculate for me:

      Cube(<no selection>, {age, gender}, count);
      Cube(town=amsterdam, {age, gender}, count);
      Cube(town=rotterdam, {town, gender}, count);
      Cube(town=rotterdam && age <40, {town, gender}, count);
      Cube(town=amsterdam && age <35, {zipcode, gender}, count);
      Cube(town=rotterdam && age < 60 and age > 35, {zipcode, gender},
      count);
```

All of these cubes are normally computed with MIL statements like :

```
age_ct   := age.group();
females := gender.select('f');

alt 1 : fem_age_ct := age_ct.semijoin(fem);
        fem_age_ct_reliable := fem_age_ct.group(reliable);

alt 2 : fem_age_ct_reliable := age_ct.subgroup(reliable,females);
```

where alternative 2 compresses all two statements of alternative 1 into one.

In the DD benchmark, `age_ct` is already computed in batch 0, and therefore can (and is) reused using alternative 2 from above. However, `age_ct_reliable` is computed (batch1) ! It is tempting to introduce a third alternative

```
alt 3 : fem_age_ct_reliable := age_ct_reliable.semijoin (females)
```

Just semijoining with females means not having to recompute the second grouping on reliable! Lets see what happens if we do this for just one grouping:

```
age_ct := age.group ( ) :
```

age

0	20
1	22
2	20

age_ct

0	0
1	1
2	0

```
fem := gender.select('f ') ;
```

gender

0	m
1	f
2	f

fem

1	f
2	f

```
fem_age_ct := age_ct.semijoin(fem);
```

fem_age_ct

1	1
2	0

Remember that:

- Grouping creates group-ids in the tail-column
- Semijoining returns those elements from the first BAT with the same head as the second BAT
- Resolving group-ids to values is achieved by use of the join-operator

If we resolve the group-ids (tail-column) of fem_age_ct with the original gender-bat we discover that a person of 22 years has changed its sex! Reason for this is that oid (0) was (and always is) an element of the original grouping (age_ct) We also see that groupid (0) appears at position 2 as well. But oid (0) is not female ! Thus, when resolving using the join-operator, we corrupt the data !
Of course, problems like this are solvable in MIL :

```
fem_val := fem_age_ct.join (fem_age_ct.reverse.join(gender) );
```

however, the DS mining engine, in general knows which hypothesis it generates, therefore it does not need to resolve this dangerous values. For the interface, this means additional administration.

Note that, when speaking of performance, the group-operation also counts the members of each group. Hence, when using the count(*) aggregate function, the counting is done while grouping. When semijoining a grouping, the registered counts do not longer correct: the count requires an extra computation. Tests have showed that recomputing the grouping is little faster then first semijoining and

then computing counts of group members. However, when using a different aggregate the semijoin is *much* faster. Therefore, the most generic solution might be the semijoin.

4.2.2.1 Multi batch query optimization

MBQO is a complex area, which is relatively new, also with DD.

Decisions for reuse are distinctive in two ways:

1. When can a cube be reused, and when not
2. Which cubes are useful to keep, and which are not

Considering point 1:

When computing N-dimensional cubes, one would be looking for cubes that are grouped on a subset of the grouping-attributes of the cube to compute. Then, if a match is found, one could re-use this cube.
For example :

> Suppose we have cube_*n* available
> grouped on gender, age, town
>
> And we get the request for cube_*m*
> grouped on : town, age, zipcode, gender, and reliable
>
> then we could
> group cube_*n* on zipcode and reliable to get the desired result

I see the optimizer here as a process, scanning available cubes for their components.

Point two is more difficult:

I would think that the lowest-dimensional cubes are the ones that are the most frequently re-used. When a cube is restricted, its reusability lowers, but its size (and its memory-load) also. The question for reusing this cube also depends on the mining-algorithm used : *what is the probability that a cube-request, similar than this one will occur in the future ?*

- When the optimizer is embedded in the MB, the administration has to be done in PL, as this is the place where all information is available. The scanner will be using Oracle tables and Oracle data type-functions for comparing attribute-values. As the number of available cubes can be quite large, I assume that the administration also will affect performance.

The administration needed for MBQO can now be characterized as follows

- which operations are performed on a cube.
- which cubes are available in Monet for re-use

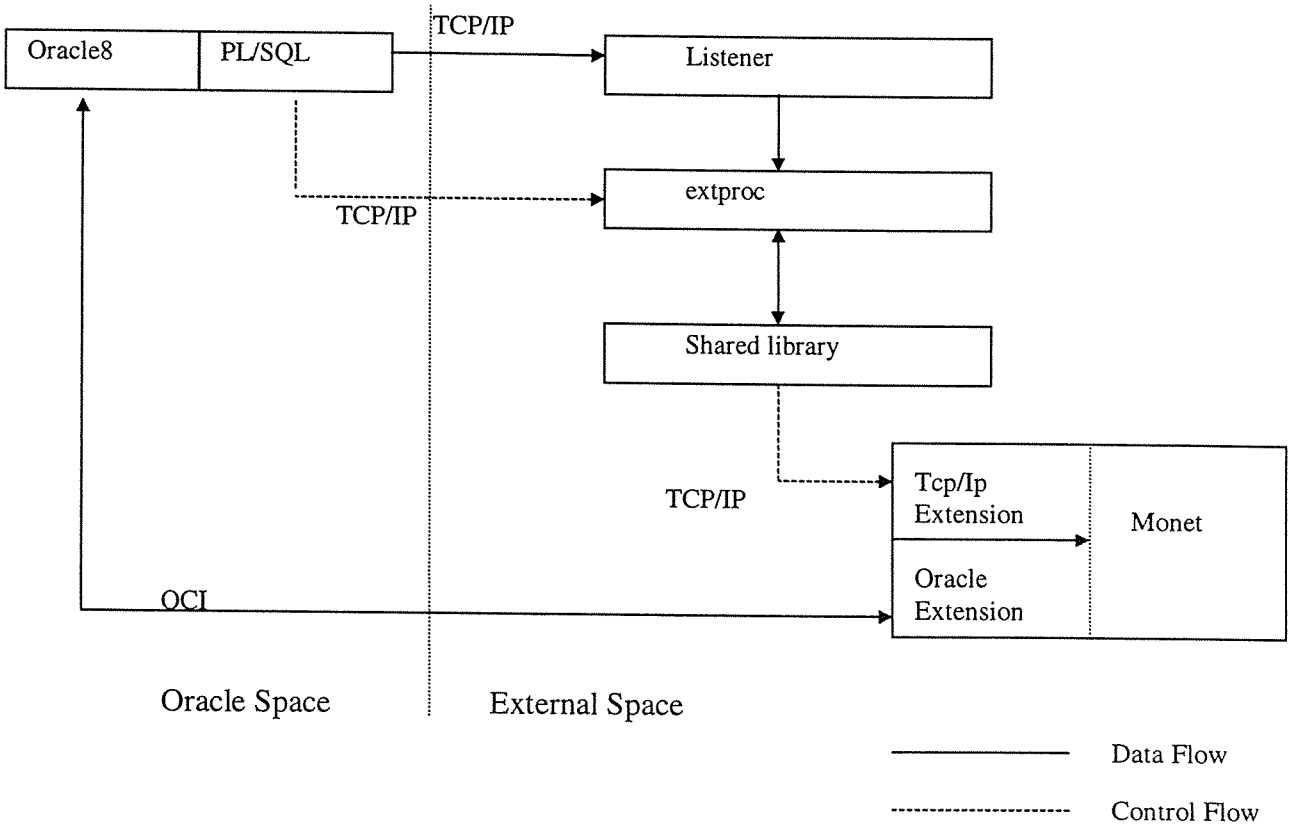
4.2.3 Communication architecture

Communication consists of a control flow and a data flow. The interface will use the control flow, where the engine will utilize the data flow. These two differ in magnitude of data-traffic : The control flow requires few bandwidth where the data flow is a typical bulk operation. As seen in chapter 3, both Oracle and Monet have various interfaces to the outside world. This paragraph is about choosing the right channel for the right flow.

As the OCI interface is the most efficient, it will be used for the bulk operations. Monet will be extended with functions like 'GetBat" and 'PutBat' which will use the OCI API towards Oracle. For the control flow, OCI is out of the question. It is a C-interface, therefore it cannot be used directly from PL. It is also not appropriate, as OCI is defined as a set of functions to access Oracle-data from outside Oracle. Passing commands does not require access to Oracle tables. The named pipe would be an appropriate channel for the control flow, unfortunately is was not implemented using the UNIX pipes but by shared memory in the Oracle memory space.
Therefore, we are forced to use the extproc-agent for passing commands. As Monet already has a TCP/IP extension, TCP/IP is the most logical entrance to Monet.

Putting all channels together in one picture results in :

4.2.3.1 Overall communication diagram of the Mining Booster



5. Design

Requirements and a set of technically challenges have been inventoried in the previous chapters, but what the interface actually looks like is not clear at all at this moment. I think this is a normal stage in the design-process as the actual look of this interface is determined by

1. Functional requirements
   - the functional requirements are clear, this is not the problem
2. Technical possibilities / limitations of the MB Engine.
   - due to the overall development stage many implementation details are unknown
3. Object-oriented possibilities / limitations of PL/SQL
   - Having no experience available in developing with Oracle8 objects throughout Europe, here also many things are uncertain.

5.1 design process

When I look back in the design process, I see several considerations and choices I have made. In order for the reader to have a view on the process as a whole, I discuss my considerations for some of the above points. At a certain point, I started implementing, and this gave a lot of input too. I finalize all this into the implemented version of dvml. After this version, a version from DD is presented, which reflects a greater overview of the whole MB project and its technical implications. However, from my point of view I think I can improve this version on some points. This last – not implemented – version, is the one I am the most satisfied with. In chapter 6, I will present an implementation proposal for this version, based on my experiences with Oracle objects.

Let us recall the functional requirements from chapter 4, and then discuss alternatives for realizing these:

Functional requirements :

- Creation of data vectors
- Construction of data cubes
- Computation of restricted data cubes
- Specify the aggregate-function used for the cube
- Observe the various aggregate-values of a data cube
- Store computed cubes

Details to hide:

- Aspects considering the MB engine and the Monet-server.
- oids, voids and enumeration (Monet-specific !)
- The path from a just computed cube [oid, int], to the actual values [any, int] with their aggregates I would like to present a data cube like a regular table.
- optimal processing considerations

5.1.5 Implementation break

Below, we see the first version being implemented

```
mb          := MBTable('demo', 'customer_big2',0,0,0,1);
batch       := batch(1);

town        := mb.GetDV ('town_0');
rel         := mb.GetDV ('reliable_0');

mb.build();

batch.addcube (age, rel, cnt);
batch.addcube (gender, rel, cnt);

if batch.exec then
      age_rel      := batch.getcube (age, rel, cnt);
      gender_rel   := batch.getcube (gender, rel, cnt);
end if;

females     := gender.MBselect('=', 'f');
adam        := town.MBselect('=', 'Amsterdam');
fem_adam    := females.MBintersect (adam);

batch.addcube (age, rel, cnt, fem_adam);

if batch.exec then
      age_rel      := batch.getcube (age, rel, cnt, fem_adam);
end if;
```

5.1.5.1 Discussion

Looking at the interface we see a new function with the MBTable : Build ( ) . Due to an implementation novelty of the MB engine, creating data vectors batch-wise gained performance. Placing this function with the MBTable is a bit of an inconsistency, as optimal batch-wise processing is placed within the batch-object. On the other hand, DVS are created through the MBTable. This inconsistency is not solvable at the moment, but we will get back to this item later on this chapter. As we can see from the script, it is possible to creating an intersection between two selections that are not yet computed in Monet. Releasing the synchronicity with the selections did not raise any implementation difficulties, whatsoever. On the contrary, DVS are also now also asynchronous, therefore why make any fuss with the getcube-function ?

```
fem_age_rel       := batch.addcube (age, rel, cnt, females);
fem_town_rel      := batch.addcube (town, rel, cnt, females);

batch.exec
```

can be realized using the same implementation as with the selections.

5.1.6 New requirements, features

Implementation raised the need for additional attributes for the IO's. This has as a consequence, that on creation of these objects, these parameters have to be set appropriately. The constructor of an object is system-generated and cannot be edited. To set these technical attributes appropriate I see two alternatives:

1. allowing creation of objects in the interface using the constructor. After calling the constructor, one has to call an init-function to set the attributes appropriately.
2. only allow the MBTable and batch to be created using the constructor. The rest of the objects should be created as a result of a function-call.

Alternative two has my preference, as it is already the case in the interface. But I think it is sensible to establish it as a need here.

Implementation also learned that administering the information for the Multi-query-optimization inside the batch-object is not that handy. It is more useful, and object-oriented, to administer the elements which construct the object within the object itself. Then the optimizer could scan the already created objects searching for parts it can reuse in the construction of a new object.
Having this in mind, one could register computation of a restricted cube like

```
fem_age_rel        := batch.addcube (age_rel, cnt, females);
```

Due to time limitations, this is not implemented. Moreover, a better solution for this will evolve from the final version of dvml.

5.2 DVML v 1.1

Now we are at the stage to present the fully-functional version of dvml. Note that this version allows only execution of the DD benchmark, nothing more.

This means:

- only the "count"-aggregate is supported
- only the intersect-operation is implemented
- only two-dimensional cubes can be computed

```
begin

mb          := MBTable('demo', 'customer_big2',0,0,0,1);
badge       := batch(1);
cnt         := aggregate('count');

if mb.logon('obsidian', 12084) then

-- *********** Batch0 ***************

      age               := mb.GetDV ('age_0');
      gender            := mb.GetDV ('gender_0');

      mb.build();

-- ********** Batch1 ****************
      age_rel           := badge.addcube (age, rel, cnt);
      gender_rel        := badge.addcube (gender, rel, cnt);

      badge.exec

      age_rel.store('age_rel');
      town_rel.store('town_rel');

-- ********** Batch2 ****************

      females := gender.MBselect('=', 'f');

      fem_age_rel       := badge.addcube (age, rel, cnt, females);
      fem_town_rel      := badge.addcube (town, rel, cnt, females);

      badge.exec

      fem_age_rel.store('fem_age_rel');
      fem_town_rel.store('fem_town_rel');

--
---- ********** Batch3 ****************

      adam              := town.MBselect('=', 'Amsterdam');
      fem_adam          := females.MBintersect (adam);
      fem_adam_age_rel  := badge.addcube (age, rel, cnt, fem_adam);

      badge.exec

      fem_adam_age_rel.store ('fem_adam_age_rel');

--
---- ********** Batch4 ****************
--
      young             := age.MBselect('between', '20', '35');
      young_fem_adam    := young.MBintersect (fem_adam);

      married_fem_adam_age_rel :=
            badge.addcube (spendings, rel, cnt, young_fem_adam);

      badge.exec
```

```
        young_fem_adam_rel.store('young_fem_adam_rel');
end if;
mb.logoff();
end;
/
```

5.2.1 Discussion

Let us check coverage of the functional requirements from the analysis

- ✓ Creation of data vectors
  Yes
- ✓ Construction of data cubes
  Yes
- ✓ Computation of restricted data cubes
  Yes
- ✓ Specify the aggregate-function used for the cube
  Yes
- ✓ Store computed cubes
  Yes
- ❖ Observe the various aggregate-values of a data cube
  It can be done, but the user has to go on manual by first storing the cube as an Oracle table, And then use SQL to retrieve the subset of the cube it is interested in. E.g.

  ```
  Select * from age_rel where rel = 'yes'
  ```

  Thus, it is possible, but it is not in the interface.

We proceed, with the details to hide :

- ✓ Aspects considering the MB engine and the Monet-server.
  The awareness of remote interaction is reduced to logging on and off to a specific host/port, and a 'Build'-function, which builds the DVS all at once.
- ✓ oids, voids, and enumeration (Monet specific !)
  These technical issues are not present in the interface
- ✓ The path from a just computed cube to the actual values with their aggregates
  The user is not aware of getting back to values. It simply calls the 'store'-function
- ✓ I would like to present a data cube like a regular table.
  Achieved.
- ✓ optimal processing considerations
  optimal processing is reduced to usage of the batch-object. I consider this minimal enough.

```
qb.cache (10, age 'between', '32', '24')
qb.cache (10, zipcode '<', '6000')
```

5.4 DVML version 1.2

Comments from the previous paragraph can be collected in a interface-proposal.

During design, I had growing difficulties with the batch-object. Creating the objects via the batch-object is not very intuitive. Adding elements to it feels clumsy to me. Also when checking the NIAM one can see I is not very integrated with the rest of the interface. Technical reasons proposed the solution with the batch object and announcing computation or storage of a cube to the batch object. After all, Administration is required to create an overview of which objects have to be created, in able to do optimizations. At the end of this project, I see an implementation possibility of removing the batch-object as an explicit object, to which a computation has to be announced, and still keep the possibility of doing the multi-query optimizations. The implementation possibilities will be presented at the end of chapter 6, for now I only present the look of this version, and will discuss its functionality.

```
begin

mb  := MBooster ('scott', 'customer', 0, 0, 0, 0);
cnt := MB_aggr('count', NULL);
avg := MB_aggr('avg', 'age_0');

if mb.logon (obsidian, 12084) then

      age          := mb.GetDV('age_0');
      gender       := mb.GetDV('gender_0');
      town         := mb.GetDV('town_0');
      martial      := mb.GetDV('marital_0');
      reliable     := mb.GetDV('reliable_0');

      mb.build ( );

      age_rel := age.combine(reliable);

      age_rel.aggregate ('table', 'age_rel_counts', cnt);
      age_rel.aggregate('file', '/var/tmp/age_rel_avg.asc', avg);

      age_marital_rel  := age_rel.refine (marital )
      age_marital_rel.agregate('table', 'age_marital_rel_counts', cnt);

      mb.cache (10, reliable, '=', 'y')
      mb.process ( );
      -- when mining on-line: inspect the results, generate the next
      lines of code

      adam              := town.MBSelect ('amsterdam');
      married           := marital.MBSelect ('married');
      married_adam      := adam.intersect (married);

      adam_married_age_rel := age_rel.restrict( married_adam );
      adam_married_age_rel.aggregate
            ('table','adam_married_age_rel_counts', cnt);
      adam_married_age_rel.
            aggregate('table', 'adam_married_age_rel_avg', avg);
```

```
        adam_married_age_rel.aggregate
               ('table','adam_married_age_rel_counts', cnt);
        adam_married_age_rel.
              aggregate('table', 'adam_married_age_rel_avg', avg);

        mb.cache (10, reliable, '=', 'y')

        mb.process ( );
        -- inspect the results
end if;
mb.logoff ( );
end;
```

5.4.1 Discussion

In my opinion, the overall look of the interface is more consistent. The object is told what to do, and the rest is handled by the MBooster. Creating DVS, building and also processing them. The build-function is maintained because it increases the size of the database. We can create our objects in any order the syntax allows us, and in the quantity only limited by the size of our main memory! With an extra cache-option - 'no cache'- even this restriction could disappear. After storage of a cube, it can be destroyed instantly. Note that checking this option will disable reusability of cubes. We have the flexibility of reusing the same grouping for different aggregates visible at the outside, which will inspire people to explore this feature: it increases flexibility. Maybe this interface could even be connected with a (to develop) mining cartridge, which can generate new DVML-statements after execution of each batch. Then caching could also be regulated from there.

However, the latter subjects are out of the scope of my project. My project was defining and implementing an interface, using Oracle8 objects, for bringing Monet-performance inside Oracle.

Milestone was running the DD Benchmark. It is achieved, including recommendations for a next version of the interface.

6. Implementation

Here the implementation from DVML v1.1 will be described as this is the fully-functional version of dvml. This chapter is best read, accompanied by the Mx-document supplied with this report. Mx is a code-documentation-tool used at DD. It enables, by usage of tags, source-code and technical documentation to placed in one file. This file can then be exported to e.g. HTML or Latex – format to be printed or browsed. This chapter starts, with the data model, followed by a paragraph on mapping to Monet and integration with the MB engine. Hereafter, the PL/SQL-part of the interface is discussed. We proceed with a paragraph on the developed communication cartridge, which enables the communication with Monet and the MB engine. We conclude this chapter with an implementation proposal for an 'sellable' version of DVML.

6.1 Mapping & Interaction

As the Oracle internal storage of the DVS is quite complex (due to optimizations) I choose to separate the 'interface' data vector from the internal data vector. This means that the interface DV is a separate object from the internal DV. This approach also enables me to work separately from Marco and implement integration later. Definition of a solid mapping between IO's and MO's has priority one at this moment, accompanied by a description of the interaction with the MB engine.

Cube-computation is achieved by sending MIL-code to the Monet server. To enable this, a communication cartridge is developed. Because the MILstrings are composed inside Oracle, it is here where the names of the used Monet-variables should be known. As the IO's itself also need a unique identifier, I chose to use this same identifier, a unique name, to be the mapping. As I keep inside each IO an administration of all IO's involved in its creation, composing the MILstrings now simplifies to pasting the names of the involved IO's into the MIL-syntax.

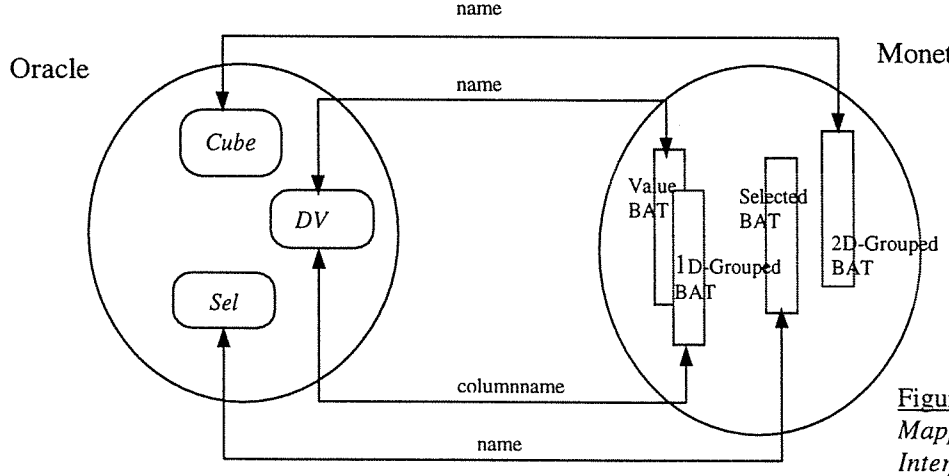


Figure:
Mapping between Interface-objects & Monet BATS

Now only remains the challenge of transporting data vectors to Monet, and transporting freshly computed cubes back to Oracle to be inspected. Solution for this challenge came from the MB engine.

During development, a monet-extensionmodule is developed containing the following functions :

```
logon ( ..parameters ..)    : session
-- logs on to the mining booster for the specified database & mining
table

fetchbats ( .. column-list ..) : BAT
-- creates and fetches all specified data vectors

put_bat_booster(..<ownername>,<tablename>,<src-BAT><involved BATS > .. )
-- src BAT is the cube
-- involved BATS are the value-DVs the cube is grouped on
-- the function-call results in an Oracle-table which lists all
discovered combinations, with their aggregate-values
```

Having this available, all communication reduces to sending the correct MILstrings at the right moment.

6.2 PL/SQL part

Starting with conclusions on possibilities of object oriented programming in Oracle8, an overall code-setup overview will be presented. Hereafter for each object, the technical implementation possibilities and choices will be discussed.

6.2.1 Objects in Oracle8

a) In Oracle8 you can create 'objects' containing attributes and member functions.
b) You can access this attributes and members by 'dot' notation.
c) Creation of an object is achieved by calling a system-generated constructor-function, who simply takes all attributes as a parameter. This constructor cannot be edited.
d) List attributes are possible through collection types. One can define a VARRAY of <object-type> or a TABLE OF <object-type> as a separate type, and then declare an attribute of this type. Unfortunately, these collections cannot be nested, which reduces their usability a *lot*
e) Storing objects is possible in so called object-tables
f) Keeping the objects up-to-date with the object-table is left to the programmer (UPDATE-statement)
g) Sharing objects (REF-type) means sharing the table entry of the object
h) Dereferencing REFS is embedded in SQL (VALUE, REF, and DEREF-operators), not by 'dot' notation

in this release,

- private members or attributes are not supported
- inheritance is not supported

Now, how to deal with this 'object-relational' approach:
Experimenting with point f) from above, it became clear to me that Oracle8 objects were an interface on top of a regular RDBMS-approach. Sharing objects is not implemented by shared memory. Now I can see 3 possibilities of dealing with this:

1. Do not use REFS, or objects as attributes of an object, use unique identifiers instead.

2. Use REFS. All of its syntactic ugliness of can be hidden behind the interface as long as every object containing REF-attributes is created as a result of a functioncall. Take the select-statements as it is.
3. Do not use REFS, but use object-attributes instead. If the objects are kept small, the size of an object with object-attributes will not increase too much. And one can use dot-notation.

Each of the above alternatives requires use of get/set-functions to access and change attributes that change value during the lifetime of an object. These functions can then encapsulate the select and update-statements. They all also have their individual advantages/disadvantages:

6.2.1.1 Alternative 1

Advantage:

I enables you to identify an attribute of an object without explicit (and implicit) use of SQL. When the object is needed, you can use SQL explicitly. This sounds good considering performance.

Disadvantage :

It shuts you of from future improvements in the syntax. One should also rewrite the sourcecode if objects suddenly are implemented using shared memory. However, I consider the latter unlikely. (e.g. being able to use objects is an option when starting Oracle)

6.2.1.2 Alternative 2

Advantage:

One profits from future implementation improvements of the selects needed when dereferencing the REFS.

Disadvantage:

I see many, many select-statements passing by when derferencing REFS. It cannot perform. However, I have not tested it exhaustive. Also the difficult creation of REFS obliges you to use functions.

6.2.1.3 Alternative 3

Advantage:

It produces good sourcecode. Using dot-notation. One should use as much functions as possible (instead of storing data), there a function-definition is stored only once: with the object-definition.

Disadvantage:

One cannot have mutual dependencies between objects.

I do not discuss hybrid constructions of these three alternatives, but I want to point out that still many administration in a ' bit of an' object-scheme has to be done in global accessible SQL-tables, as collections cannot be nested. Hence, safe object-oriented programming is out of the question.

In my implementation, for safety reasons, I decided to use the object-features just to give the interface an object-feel. Inside the interface, I decided to use PL/SQL where it is designed for, to manipulate SQL-tables in a procedural approach. After

<create selections> <create cube> <make cube persistent>

Example :
Let's suppose :

- the cube (town, reliable, "count", old_females) is added to the batch
- the grouped version of town is called dv1
- the selection 'old' is already present in Monet, and is called sel1
- the new name of the cube is cube5

Then, the MILcode consists of the following parts :

<create selections>

```
-- sel1(old people) is already present in Monet, thus not computed again
sel2 := gender_0.uselect('f');  -creationstring of females
sel3 := sel1.semijoin(sel2); -- creationstring of old females
```

<create the cube>

```
cube5 := dv1.subgroup(reliable_0, sel3);
-- this is the creationstring of cube5
-- dv1 is the grouped DV from the column town_0
-- reliable_0 is the value DV from the column reliable_0
```

<make the cube persistent>

```
-- this is only done if mbtable.debug is set to one
cube5.rename("cube5"); -- to materialize the mapping between the IO and the BAT
cube5.persists(true);
```

6.3.5 MBTable

Central name-generation

The function new_name takes a prefix as an input-parameter, and returns this same prefix, with a number attached to it. Allowed prefixes are 'sel', 'cube', and 'dv'.
Depending on the prefix, the corresponding counter is incremented, and this update is of course prolonged to the object-table.

Build

This procedure calls the multi-bat-creation function get_booster_bats from the Module Booster in Monet. This function needs some identification parameters and all columnnames from which we want to create data vectors. These columnnames are available from the DV-object. Result of the get_booster_bats function is creation and transport of the specified data vectors, and a mapping-bat called 'all_bats'. (When transported to Monet, the data vectors still have names like tmp_142, tmp_135 etc. The mapping bat contains the mapping between the tmp-names and the columnnames)
Thus, to materialize the intended mapping between Oracle- data vectors, and the Monet-bats, some action is required. I want :

- the value-BAT to be renamed to <columnname>
- the value BAT to be grouped to a BAT named DVname

To achieve this I collect the renamestring and the groupingstring from all created DVS, and sends them to Monet just after the call to get_booster_bats. These grouping- and renamestrings are set in the function GetDV( ).

GetDV
This function does a check if a data vector with the same columnname is already created in the interface if not, it creates the data vector-object. As pointed out above, when transported to Monet, this DV needs to renamed to a BAT named <columnname>and grouped to a BAT named <DVname> to ensure consistent mapping between the IO and the Monet-objects. Seen from Oracle, we only know the name ("all_bats") of the mapping –BAT which contains the mapping between the columnname, and the BAT with the data. I, of course want to rename and group the BATS by sending MILstrings. Monet is equipped for such a problem. In the module BAT we encounter the perfect function for this:

```
find (BAT src, str name) : BAT
```

which searches in the head-column of the BAT src for the string name and returns the BAT from the tail-column. Apparently is leaving mapping-BATS a standard mechanism from Monet. Knowing of the find-function, the MILstring is easily composed to :

```
all_bats.find("<columnname>").rename("<columnname>");
```

and after this, the groupingstring is even simpler

```
<DVname> := columnname.CTgroup( );
```

After composing these MILstrings, the DV-object is created, and asked to insert itself into the data vector-table. Hereafter, the IO is returned as a functionresult.

6.4 the communication cartridge

All interaction with Monet and the MB engine is initiated by sending MILstrings to the Monet server. To send MILstrings, I chose to develop a simple communication-object, containing one function : sendcommand (<commandstring>)
Embedding this function is a separate communication-object enlarges of course its usability in the interface. First I will describe implementation choices for the function sendcommand. Hereafter I construct the object to be functional in the interface.

6.4.1 sendcommand

As pointed out in the analysis, the communication channel should be TCP/IP. Within TCP/IP there are two choices.

1. Monet provides the extension-module tcpip. This module is intended for Monet-Servers to communicate, and exchange data. sending MILstrings is possible, but syntactically different from 'normal' MILcode

2. When a Monet-server is running, it listens on a standard port for clients to connect. For each client connecting to this port a separate context is created, thus freeing it from all concurrency-problems. The Mclient-interface is used by for the Mclient program to open a console-session on the Monet-server. Mclient provides a more user-friendly interaction with facilities like command-history etc... The protocol for a session like this is of course regular MIL followed by a newline-character.

The second choice is the most suitable for me to use.

After thorough investigation, the complete protocol for the Mclient interface turned out to be :

1. logon with <username>\n
2. send MIL-command(s) \n
3. read the response from the Monet –interpreter from the same channel (terminated by '\001')
4. read the prompt Monet presents to the Mclient-program. (terminated by '\001')

Now we have established the communication channel, chose the most suitable interface, and figured out the exact protocol, we can point out the necessary steps for sending one MILstring to Monet.

1. open the socket to the specified host and portnumber
2. write the command in the socket
3. read the response from the socket
4. read the prompt from the socket
5. .....???

Ending up at point 5 we encounter a problem in the Monet-communication :once we have send a command to Monet, we are through for this functioncall. What happens next? The connection with Monet should remain open in order to preserver the current context.

How does Oracle handle functioncalls to an external library exactly? Let us recall some information from the context chapter.

When calling an external procedure, the listener activates a session-specific agent called extproc.. This agent. loads the specified library from disk, and executes the function. After the function terminates, extproc hands over the control back to Oracle. ***Extproc remains active throughout the Oracle session.*** *When the Oracle session terminates, extproc is killed*

One could expect to find all variables unchanged when recalling the external function. In this case there would be no problem and we could write the same socket again. Tests showed that indeed, the extprocagent remained active after the PL/SQL program was finished. The Monet-connection also remained open after execution of the PL/SQL program. Unfortunately, all variables were re-initialized upon each functioncall. Apparently the library is refreshed, or re-initialized when called again.

The positive point is that the Monet-connection remains open after the external function terminates, otherwise an old facility of Monet had to be re-installed which enables terminating, and re-entering a Monet-session. For now, a sufficient solution is to transport the socketnumber back to Oracle, and with the next functioncall pass it as a parameter. With the first command (mlogon) a zero could be passed. This construction enables re-use of the socket.

6.4.2 Structuring the communication cartridge

After this technical issue, let us assemble the parts together to a usable communication cartridge.

An external function is never a member of an object. Oracle does not allow it. An external function is installed under a user-id. The communication cartridge is user of this function. Its memberfunction sendcommand calls the external function with the appropriate parameters. My idea is to store the socket number, along with hostname and portnumber inside the communication object. Then when sendcommand calls the external function, it could pass "self.hostname", "self.portnumber", "self.socketnumber" as a parameter. The socketnumber, which is returned by the external function, could then be updated in the communication-object to be re-used with the next functioncall.

In my opinion, the communication-object is now fully customizable on creation, and ready for usage in the interface.

6.4.3 Integration in the interface

In the logon-function of MBTable, the communication-object is created. Hostname and portnumber can be intuitive parameters for the logon function of MBTable. Now, initiating the contact with Monet is reduced to calling 'mbtable.logon(hostname, portnumber)';

6.5 debugging facilities

The debugging facilities consist of :

- PL/SQL output, which gives information on the objects being created
- ASCII-output on std-out on the oracle-server, which gives information on the MIL-code being executed, and the response of the Mserver to it.
- All created objects can be saved to be inspected from the console

All these three options are triggered by one attribute of MBTable. PL/SQL-output can be disabled manually by putting a statement ' DBMS_OUTPUT.DISABLE' after the logon function.
Sending the MIL-code that saves the objects cannot be disabled manually, but it is not tightly integrated in the code.

6.6 Implementation proposal DVML 1.2

In this chapter I will describe the implementation possibilities of DVML 1.2. I will take the implementation of v1.1 as departure, describe how this can be improved, and fitted into the new look of version 1.2.

Advantage of the current implementation is that every object in the interface knows how to create itself in Monet. This makes the interface very flexible in the sense of 'loose' order of creation, because every object also knows weather it is available in Monet.

6.6.1 Process-function

Considering announcement of the cube for processing in the next batch, we have two options:

1. We could add for each object the knowledge if it has to be processed in the next batch, a `'compute_on_next_call'`-attribute. Then the process function needs a view on the object-table with all cubes to compute.
2. Also we could maintain the current insert of an identifier in a separate table, and hide it behind the interface.

Order of execution is not an issue. If one of the components of a cube is not yet created, the Booster just takes the components creationstring and creates the object on-the-fly ! (and sets the presence-attribute to one, to make sure the component is not created twice) This can all be achieved using the same algorithm currently implemented in the function 'getcreationstrings', located in the batch-object. The optimizer could also set these creationstrings appropriate for the current batch- and cache population.

6.6.2 Cache-function

Choosing between the above two alternatives will also influence the implementation of the cache-function: It needs, after execution of the batch, knowledge about which cubes to examine for aggregate-values. Then it is able to leave the top *N*, and destroy the rest. This can be achieved with both alternatives of the previous paragraph. Examining the cube-tables can be done with SQL. The aggregated cube-tables will in general not be that large, but this depends on the data. Otherwise one should think of returning a sort of ranking-score from Monet.

6.6.3 Aggregate-function

Being able to read from the cube which aggregate-function to use, and to which destination the aggregated cube should be stored, obliges us to extend the cube with an extra attribute : the aggregate.

extra NIAM voor cube met aggregate-extensie (type, name)

6.6.4 Optimizer

Considering the optimizer, I think the choice between REFS and names is won by the REFS . REFS can be dereferenced from SQL. Hence, when scanning the objects for parts to reuse, we could use SQL. How this performs, depends on the implementation of Oracle on this, but a BIG toolkit is available for tuning this process. To implement the optimizer in PL, one needs unique variable names to map to in Monet. E.g. suppose we need to create :

1. Cube(age, gender, zipcode, reliable) and
2. Cube(age, gender, zipcode, town)

And we have thrown away

3. Cube (age, gender, zipcode)

Which could reuse

4. Cube (age, gender), which is still available

It makes sense to first create 3 from 4 and then use 3 to create 1 and 2.
This could be hacked in the MILcode as well, but this would be less controllable than this solution.
One could even think of creating a shadow-administration, in the case the user only computes cubes not that suitable for re-use (selected on many attributes)

Note that creationstring could be stored more efficient, by returning it as a function-result. This is because function-definitions are stored only once, in the object-definition, and no with each object again, like attributes. This string is not trivial in all cases, certainly not if the optimizer will be implemented as setting these strings based on the current batch-and cache-population.

7. Conclusions

Version 1.1 of dvml is implemented, which provided me with sufficient background to propose version 1.2, which absract further from the original MILcode. For my own feeling the demanded functionality is present, and more. We can speak of cubes, the terminologie of the field, and this will lower the learning curve users of this interfece have to go through. Also, the possibility of connecting to another data cartridge, a mining cartridge, is mentioned, and, in my opinion, very implementable. This ensures tight integration in the current software-architecture of the Data Surveyor suite, which is a good result. I am sure this was the purpose of this cartridge, but discovering it myself at the end of the project was a big pleasure for me. This was a hard project, the learning curve with DD is steep and long, but I finish it with a feeling of satisfaction.

Thanks to:

Fred Kwakkel,

, for his excellent help on writing this report

Peter Bonz,

, for his help on Monet

Marco Fuykschot

, for on-line support,
(*any targets in this Makefile* ?)
and a good time

Donald Kwakkel

, for his help on the 'sneaky' tricks of MIL

8. Glossary

8.1 List of abbreviations:

DD	*Data Distilleries*
DDB	*Drill Down Benchmark*
DS	*Data Surveyor*
CWI	*Centrum voor Wiskunde en Informatica*
RDBMS	*Relational Database Management System*
OLTP	*On Line Transaction Procesing*
SQL	*Structured Query Language*
PL/SQL	*Procedural Structured Query Language*
OLAP	*On Line Analytical Processing*
MB	*Mining Booster*
BAT	*Binary Association Table*
MIL	*Monet Interaction Language*
ID	*IDentifier*
DVML	*Data Vector Manipulation Language*
DV	*Data Vector*
DVS	*Multiple Data Vectors*
IO	*Interface Object*
MO	*Monet Object*

8.2 Used, not-trivial MIL-operators:

```
group (bat[oid,any]) return bat[oid,oid]
```

- It creates for each different value in the tail-column a new group-id. This group-id is the oid of the first occurrence of this value. Every next occurrence of this value is marked as member of this group

```
group(bat[oid,oid], bat[oid,any]) return bat[oid,oid]
```

- This is an overloaded version of the previous group-operator. Now it looks at all different combinations of values in the first and second argument. New group-ids for the discovered combinations are created.

```
join(bat[any::1,any::2], bat[any::2,any::3]) return bat[any::1,any::3]
```

- It takes two input-BATS, joins the head of the first with the tail of the second, and returns a BAT with their joint elements (based on the any::2 - column).

```
semijoin (bat[any::1,any::2], bat[any::1,any]) return bat[any::1,any::2]
```

- returns the original BAT, but only those elements with the same head as the second BAT: i.e. selecting one BAT with another BAT.

9. Reference

- MIL primitives for Quering A Fragmented World

  Peter A.Boncz &
  Martin L. Kersten,
  University of Amsterdam, CWI

- Data Mining and Knowledge Discovery

  Usama Fayyad,
  Microsoft Research, USA
  Hekkiki Manilla,
  University of Helsinky
  Gregory Piatetsky,
  Knowledge Stream, USA

- Oracle8 documentation CD

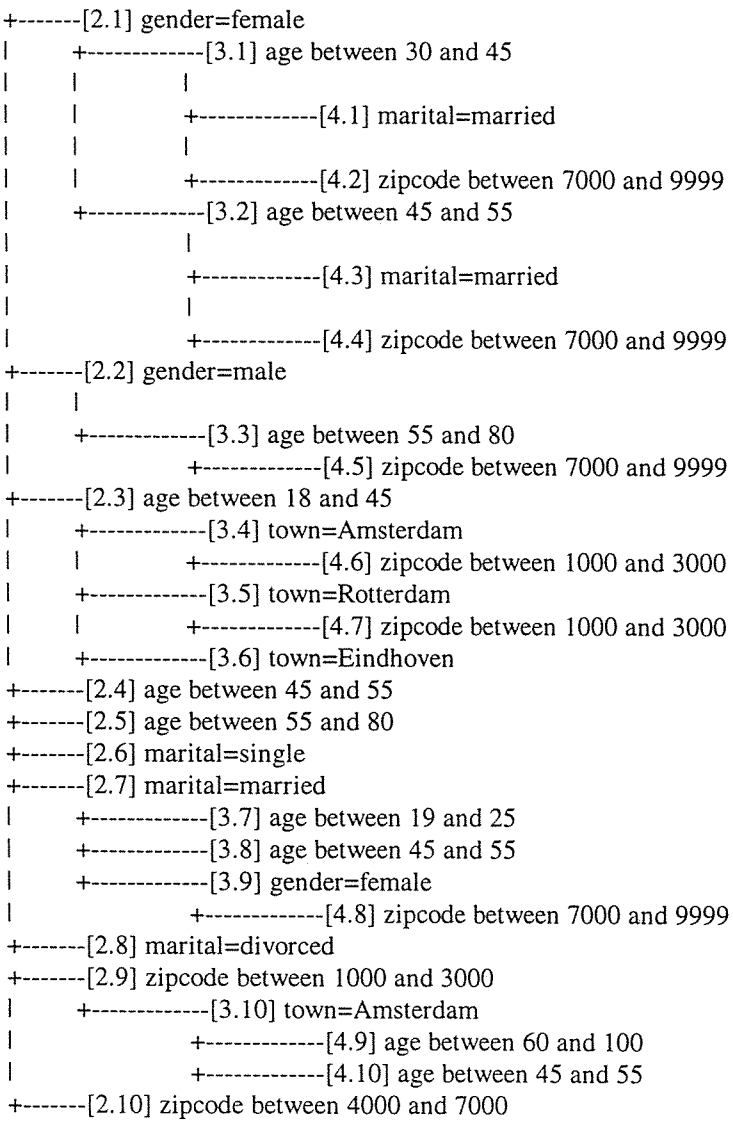
  Oracle corporation

- Oracle server

  Newsgroup

10.2 the DD benchmark search space

This search space definition considers batch 3-5 of the DD benchmark where the DS mining engine applies the beam search-strategy. It shows all branches of the beamsearch(10,3), and which branches the DS mining-engine decides to expand further, and which not.



batch 0:
7 times SELECT..FROM..WHERE..GROUPBY on 100% of customer
batch 1:
6 times SELECT..FROM..WHERE..GROUPBY (i.e. 7-1) on 100% of customer
batch 2:
50 times SELECT..FROM..WHERE..GROUPBY (i.e. (6-1)*10) on 30% of customer

batch 3:
40 times SELECT..FROM..WHERE..GROUPBY (i.e. (5-1)*10) on 15% of customer
batch 4:
30 times SELECT..FROM..WHERE..GROUPBY (i.e. (4-1)*10) on 8% of customer

9. Reference

- MIL primitives for Quering A Fragmented World

  Peter A.Boncz &
  Martin L. Kersten,
  University of Amsterdam, CWI

- Data Mining and Knowledge Discovery

  Usama Fayyad,
  Microsoft Research, USA
  Hekkiki Manilla,
  University of Helsinky
  Gregory Piatetsky,
  Knowledge Stream, USA

- Oracle8 documentation CD

  Oracle corporation

- Oracle server

  Newsgroup

10. Appendix

10.1 NIAM : sentences for the MB-interface

Name <N> is a datavector
columnname <CN>is a datavector

Datavector <DV> has a groupingstring <GS>
Datavector <DV> has renamestring <RS>
Datavector <DV> has debugstring <DS>
Datavector <DV> belongs to MBTable <MB>
Datavector <DV> communicates through Mconnection <MC>

Name <N> is a Selection

Selection <S> is available in Monet
Selection <S> has creationstring <CS>
Selection <S> has debugstring <DS>
Selection <S> belongs to MBTable <MB>
Selection <S> communicates through Mconnection <MC>

Selection <S> is a Value-selection
Selection <S> is a Composite-selection

Value-selection <S> is a selection on datavector <DV>
Value-selection <S> has selectionvalue <SV>
Value-selection <S> has second selectionvalue <SV>
Value-selection <S> has comparison-operator <CO>

Composite-selection <S> has as left Selection <S1>
Composite-selection <S> has as right Selection <S1>
Composite-selection <S> has concatenation operator <CO>

Name <N> is a Cube

Cube <C> has source-datavector <DV>
Cube <C> is grouped on Datavector <DV>
Cube <C> is selected with Selection <S>
Cube <C> has MBaggregate <MBA>
Cube <C> is processed in Monet
Cube <C> has creationstring <CS>
Cube <C> has debugstring <DS>
Cube <C> belongs to MBTable <MB>
Cube <C> communicates through Mconnection <MC>

Functionname <F> with DV <D> is an MBaggreagate

ownername <N> with tablename <TN> is an MBTable

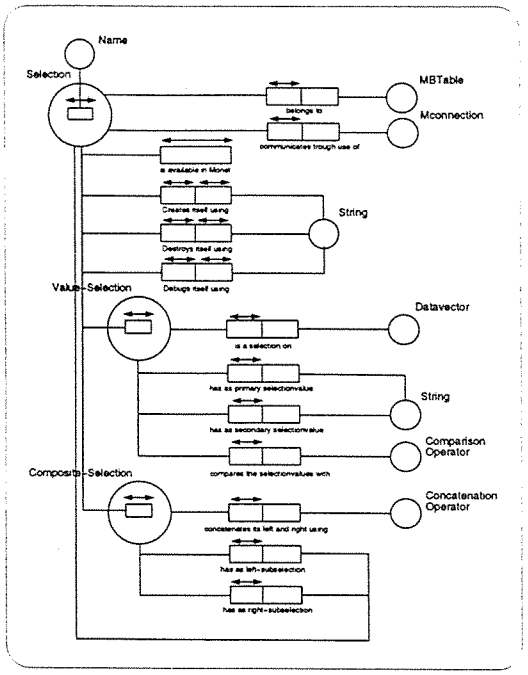
MBTable <MBT> has datavector <D>
MBTable <MBT> has selection <S>
MBTable <MBT> has cube <C>
MBTable <MBT> communicates through Mconnection <MC>

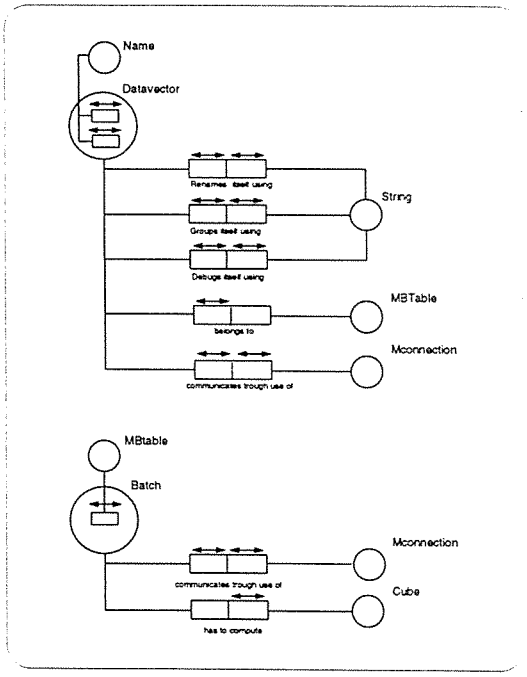
Number <N> is a BATCH

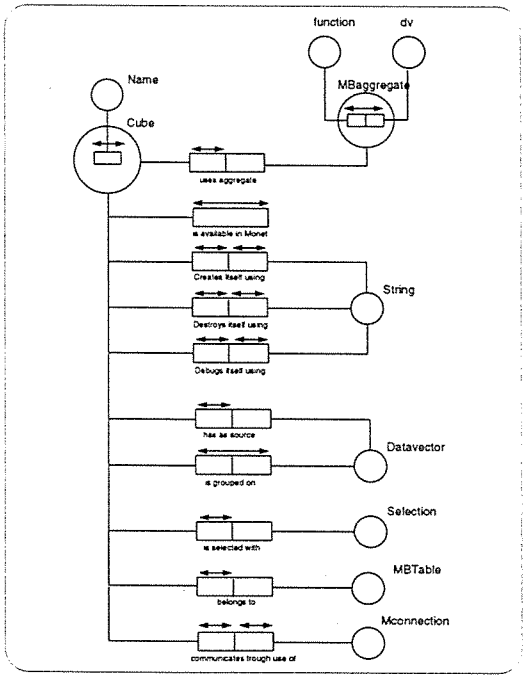
Batch <B> has to compute cube <C>
Batch <B> communicates through Mconnection <MC>

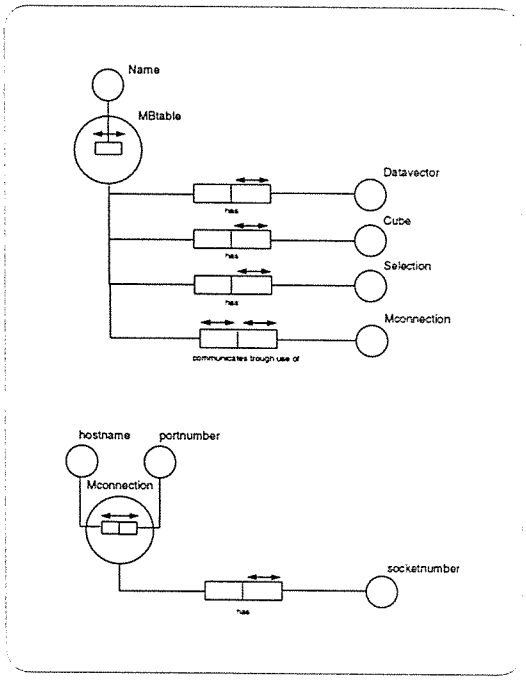
Hostname <N> with sockernumber <SN> is an Mconection
M connection <MC> uses socket with number <N>

10.1.1 NIAM Scheme



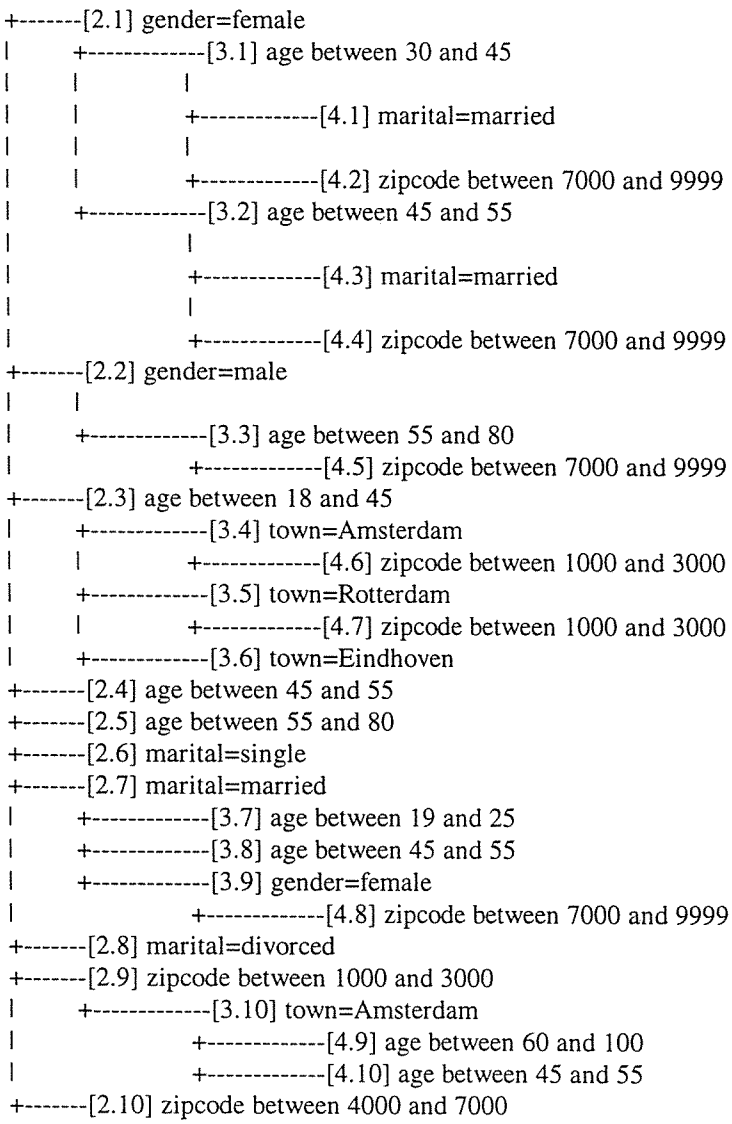






10.2 the DD benchmark search space

This search space definition considers batch 3-5 of the DD benchmark where the DS mining engine applies the beam search-strategy. It shows all branches of the beamsearch(10,3), and which branches the DS mining-engine decides to expand further, and which not.



batch 0:
7 times SELECT..FROM..WHERE..GROUPBY on 100% of customer
batch 1:
6 times SELECT..FROM..WHERE..GROUPBY (i.e. 7-1) on 100% of customer
batch 2:
50 times SELECT..FROM..WHERE..GROUPBY (i.e. (6-1)*10) on 30% of customer

batch 3:
40 times SELECT..FROM..WHERE..GROUPBY (i.e. (5-1)*10) on 15% of customer
batch 4:
30 times SELECT..FROM..WHERE..GROUPBY (i.e. (4-1)*10) on 8% of customer