EOSCUBE: A Constraint Database System for High-Level Specification and Efficient Generation of EOSDIS Products

Final Report for Phase 1: Proof-of-Concept

Alexander Brodsky ‡†§
Victor E. Segal ‡†

Consultants:
Jia Chen ‡
Menas Kafatos §
Owen Kelly §
Larry Kerschberg ‡
Samuel Varas ‡
Ruixin Yang §

†CESDIS, Goddard Space Flight Center (GSFC)
NASA

‡Department of Information and Software Engineering (ISE) and Center for Info. Systems Integration and Evolution (CISIE)
School of Information Technology and Engineering (IT&E)

§Center for Earth Observing and Space Research (CEOSR)
Institute for Computational Sciences and Informatics (CSI)

George Mason University
March 15, 1999
## Contents

I  EOSCUBE: Executive Summary  

II  Productivity and Feasibility Study  

1  Studied Domains  

2  EOSCUBE Database of AVHRR Data  

3  Products and EOSCUBE Programs  

3.1  NDVI Computation over Arbitrary Areas  

3.2  Computing Areas with SST and Clouds Satisfying Conditions  

3.3  Computing Area with SST Changes over Time  

3.4  Time Series of Spatial Correlation between Shifted Areas - Version 1  

3.5  Time Series of Spatial Correlation between Shifted Areas - Version 2  

3.6  Time Series of Spatial Correlation between Shifted Areas - Version 3  

3.7  Time Series of Spatial Correlation between Shifted Areas - Version 4  

3.8  Temporal Correlation  

3.9  Spatial Aggregation  

3.10  NDVI Animation Movie  

3.11  Temporal Correlation and SST Anomaly  

3.12  Color World Map  

3.13  Coloring Counties based on NDVI Means  

3.14  Counties and NDVI Coverage  

3.15  Sort in Drop Order  

3.16  NDVI Composition Algorithm  

III  EOSCUBE Features and Language  

4  Introduction  

5  CST Objects and EOSCUBE Queries by Example  

5.1  Constraints, CST objects and Schema by Example  

5.2  EOSCUBE Queries by Example
VI Conclusions and Suggested Future Directions

4
Part I
EOSCUBE: Executive Summary

The EOSCUBE constraint database system is designed to be a software productivity tool for high-level specification and efficient generation of EOSDIS and other scientific products. These products are typically derived from large volumes of multidimensional data which are collected via a range of scientific instruments.

Main Objectives for Phase 1 (Proof-of-concept):

- To demonstrate that EOSCUBE can provide considerable savings in development time of EOSDIS and other scientific products
- To demonstrate that product generation by EOSCUBE from real data sets is feasible.

Ultimate Goals (beyond Phase 1):

Productivity gain: EOSCUBE will allow Earth scientists to compactly specify data products concentrating on their scientific domains, while being relieved from a considerable programming effort.

Interleaved and Optimized Production: EOSCUBE will provide interleaved pipelined evaluation of a series of inter-related products, automatically optimizing dataflow control, buffer management, and materialization supporting clustering and indexing.

Platform Independence: EOSCUBE will support hardware/software platform independence, so that platforms’ change would only require changing a small number of interface methods, while leaving products generation software unchanged. It is planned that EOSCUBE will support a mix of underlying object managers, databases, mass storage systems, or just file systems in a very flexible way.

Easy Integration: EOSCUBE is used from within a C++ program and allows to use existing C/C++ code, without the need to translate data types and formats.
Accomplishments in Phase 1:

- Development of the EOSCUBE proof-of-concept prototype based on the CCUBE constraint object-oriented database system
- Specifying in EOSCUBE a range of scientific products, and actually generating a number of them using real input data sets.
- Preparing reports, within this final report, on:
  - Feasibility and productivity study, which contains EOSCUBE specification of a number of scientific products, and test cases run on real data sets
  - Specification of EOSCUBE features and language
  - Architecture and implementation of the EOSCUBE prototype
  - Work in progress on optimizing multi-product generation workflow
  - Recommended course of action

Main Conclusions in Phase 1:

- EOSCUBE has the potential for significant productivity gain in specification and generation of EOSDIS and other scientific products
- Generation of scientific products from real data sets is feasible using the EOSCUBE prototype
- An industrial-strength EOSCUBE implementation will be necessary for deployment and massive use of the system.
- The EOSCUBE language should allow incremental extensions, which are unavoidable in diverse scientific domains
- The overall evaluation model should also support data-flow processing (i.e. pipeline evaluation), in addition to query processing.
- The main aspects of global optimization should deal with interleaved pipelined evaluation of series of inter-related products, and concentrate on optimizing throughput via data flow control, buffer management, and materialization supporting clustering and indexing.
Future Action Paths for EOSCUBE:

We elaborate on recommended activities in Section VI. Below is a summary of main paths of action that will have to be carefully discussed and planned with EOSDIS.

Research Path, including local and global optimization, spatio-temporal indexing and clustering, and GIS constraint algebras

Industrial-strength implementation path, including high-performance EOSCUBE kernel, pipeline evaluation model, ODBC and platforms support, and GIS integration.

Collaborative work with Earth scientists on a specific set of new products, and continued customization of EOSCUBE for them. This will also used as a leverage for later massive deployment of EOSCUBE.

Deployment of EOSCUBE to Centers and Technical Support
Part II
Productivity and Feasibility Study

1 Studied Domains

We have studied the following three domains of processing in order to select products on which to exemplify EOSCUBE. We next briefly elaborate on each.

Geolocation data processing

This domain includes such procedures as pixel scanning algorithms, radar geometry, and systems of coordinates. The algorithms used in this domain primarily consist of sequences of vector and matrix manipulations, calculations of function values, long summations, and other arithmetic operations. Specifically, we studied the following documents in regards to this domain:

- MODIS Level 1A Earth Location ATBD
- TSDIS algorithm descriptions and equations for combined Geolocation, L1B VIRS and L1B TMI

Conversion of radar measurements into geophysical quantities

This domain directly deals with the radar measurement science. Consequently, data transformations used by the products are mostly large formulas involving mathematical analysis entities such as integration over time and space, the gamma function, trigonometric $\mathcal{A}$ transformations, limits etc. Specifically, we studied the following documents in regards to this domain:

- R. Meneghini and T. Kozu, Spaceborne Weather Radar
- M. Marzoug etc., 'A Class of Single- and Dual- Frequency Algorithms for Rain-Rate Profiling from a Spaceborne Radar. Part I' (TRMM)
- T. Iguchi etc., 'Intercomparison of Single-Frequency Methods for Retrieving a Vertical Rain Profile from Airborne or Spaceborne Radar Data' (TRMM)
- TSDIS Levels 1,2,3 data file specifications
- TSDIS Toolkit User's Guide
- TSDIS Software Design Specifications
Adjustments of geophysical quantities and their aggregations over space and time

This domain involves analysis of streams of measured quantities over time intervals, calculation of more precise values using compositions, adjustments of quantities in regards to other factors present during measuring, standardization of quantities, spatial aggregation and generation of grids of various resolutions. The transformations in this domain generally start from level 3 and go up to the data levels suitable for end-user presentation.

Specifically, we studied the following documents in regards to this domain:

- MODIS Vegetation Index ATBD
- MODIS Land Cover ATBD
- MODIS Infrared Sea Surface Temperature Algorithm
- MODIS Cloud Top Properties and Cloud Phase

We chose primarily this domain to exemplify the use of EOSCUBE. For the reasons of data availability, we decided to use data obtained by the Advanced Very High Resolution Radiometer (AVHRR), which was provided to us by the CEOSR center at GMU. Next we discuss the EOSCUBE databases that was created to store AVHRR data, and which facilitates clustering and secondary storage random access.

2 EOSCUBE Database of AVHRR Data

In the following sections we demonstrate how EOSCUBE can be used to query and produce products from the following geophysical parameters: vegetation index, cloud amount and sea surface temperature. The background for calculation of these parameters has been obtained by studying the documents mentioned above. However, in this section we are not concerned with how the parameters have been computed from the lower-level products. Most queries in the next sections deal with performing computations such as:

- computing parameters for arbitrarily complex areas
- describing spatial and temporal characteristics of areas where parameter(s) satisfy a certain condition
- performing spatial aggregations
showing area changes over time

A number of products (see details in the next sections) have been actually run on real-data sets (in the AVHRR database) and produced output. The actual data used to run those queries has been collected from several missions using the Advanced Very High Resolution Radiometer (AVHRR). Below are the references to the data sets descriptions:


The database that we have constructed represent essentially a grid of cells which provide Earth spatial coverage. The grid is a three-dimensional vector with the dimensions representing latitude, longitude, and a time-point (in months), correspondingly. Apart from regular C++ class methods such as constructor and destructor, the grid also contains the following fields and methods:

- **Resolution** is the grid’s resolution, 1 degree in our case
- **Low_Lat** is the latitude value corresponding to the (0,0) grid element
- **Low_Lon** is the longitude value corresponding to the (0,0) grid element
- **Lat_Ind_Range(C3_C.RANGE)** gives the range of the latitude index that corresponds to a given rectangle
- **Lon_Ind_Range(C3_C.RANGE)** gives the range of the longitude index that corresponds to a given rectangle
- **Lat_Ind_Upb** is the latitude index upperbound
- **Lon_Ind_Upb** is the longitude index upperbound
Each spatial cell in the grid represents a time series (a vector) of three parametric values:

- NDVI stands for vegetation index, a real number between 0.05 and 0.65. If the value is missing for a particular space-time point, the value of -99 is used instead.

- SST stands for sea surface temperature measured in Kelvin. The value of 0 is used for land surfaces.

- CLD stands for cloud amount, a real number between 1 and 100 which has been computed as the average frequency of cloudy pixels. The value of -99 is used for missing data.

There is also a temporal vector of SOI (Southern Oscillation Index) values. Those values do not depend on spatial location and are single values for each time point. Excerpts from the corresponding EOSCUBE declarations appear in Figure 1. The datasets used adhere to the following assumptions:

**Spatial Coverage** is global. Data in the grid is ordered from North to South and from West to East beginning at 180 degrees West and 90 degrees North. Point (0,0) represents the grid cell centered at 89.5 N and 179.5 W.

**Spatial Resolution:** The data are given in an equal-angle lat/long grid that has a spatial resolution of 1 X 1 degree lat/long.

**Temporal Coverage:** January 1987 through December 1988

**Temporal Resolution:** Monthly mean.

In the products descriptions in the next sections, we will use the following notation:

- **EOSCUBE database of AVHRR data** will denote the input grid, a C++ variable Grid of the type EOS3.Earth.Grid

- **clin_area** will denote a C++ variable of type C3.C.LIN, representing an arbitrary polygon in CST variables lat, lon

- **dcrange.area** will denote a C++ variable of type C3.DC.RANGE, representing a union of rectangles in CST variables lat, lon

- **range_area** will denote a C++ variable of type C3.C.RANGE, representing a rectangle in CST variables lat, lon
class EOS3_Cell
{
public:
    //...
    C3_Double NDVI;
    C3_Double CLD;
    C3_Double SST;
};

class EOS3_Earth_Grid : public C3_Vector<C3_Vector<C3_Vector<EOS3_CELL>>> {
public:
    //...
    C3_Int32 Lat_Ind_Upb;
    C3_Int32 Lon_Ind_Upb;
    C3_Double Resolution;
    C3_Double Low_Lat;
    C3_Double Low_Lon;
    C3_Range_Monoid Lat_Ind_Range( C3_C_RANGE);
    C3_Range_Monoid Lon_Ind_Range( C3_C_RANGE);
    //...
};

class County
{
public:
    //...
    C3_String Name;
    C3_DC_LIN extent;
    //...
};
%

Figure 1: EOSCUBE Declarations for AVHRR Grid Schema
month will denote a C++ variable of type C3_Int32, which represents a time point, a month index ranging from 0 to 23 (for the two-year period).

Result will denote a C++ variable representing the result for each select query. The type of the variable will be different for each product.

MBR stands for Minimum Bounded Rectangle.

SOI_Time_Series is a variable of type C3_Vector<C3_Double> which represents the SOI temporal vector.

The datasets were provided to us in the form of ASCII files and then imported into EOSCUBE with ObjectStore OODB as underlying storage manager. Below is a sample data fragment representing cloud amount coverage for Jan 1987, extracted from the corresponding file:

```
... ...
58.000 58.000 58.000 58.000 58.000 58.000 58.000
58.000 58.000 58.000 62.000 66.000 66.000 66.000
66.000 66.000 66.000 66.000 66.000 66.000 66.000
66.000 66.000 70.000 70.000 70.000 70.000 70.000
70.000 70.000 70.000 70.000 70.000 70.000 70.000
70.000 70.000 70.000 70.000 70.000 70.000 70.000
70.000 70.000 70.000 70.000 70.000 70.000 70.000
73.250 76.500 76.500 76.500 76.500 76.500 76.500
76.500 76.500 76.500 76.500 76.500 76.500 76.500
76.500 76.500 76.500 76.500 76.500 76.500 61.500
61.500 61.500 61.500 61.500 61.500 61.500 61.500
61.500 61.500 61.500 61.500 61.500 61.500 61.500
61.500 61.500 61.500 61.500 61.500 61.500 61.500
61.500 61.500 61.500 61.500 61.500 61.500 61.500
72.500 72.500 72.500 72.500 72.500 72.500 72.500
72.500 72.500 72.500 72.500 72.500 72.500 72.500
72.500 72.500 72.500 72.500 49.000 49.000 49.000
49.000 49.000 49.000 49.000 49.000 49.000 49.000
49.000 49.000 49.000 49.000 49.000 49.000 49.000
49.000 49.000 64.250 79.500 79.500 79.500 79.500
79.500 79.500 79.500 79.500 79.500 79.500 79.500
79.500 79.500 79.500 79.500 79.500 79.500 79.500
... ...
...
3 Products and EOSCUBE Programs

3.1 NDVI Computation over Arbitrary Areas

Calculate NDVI for an arbitrarily input polygon area and a given month.

Input

- EOSCUBE database of AVHRR data
- clin_area over which NDVI is to be computed
- month for which NDVI is to be computed

Output

Result, of type C3.Double, a real value representing the vegetation index for area at time point tp, calculated using the mean average

Description

An MBR for clin_area is computed, then iteration over the grid's window corresponding to the MBR is performed using the Lat.Ind.Range and Lon.Ind.Range methods. For each cell the TV predicate checks if the cell is actually inside the area. Each C-RANGE is then checked on intersection with clin_area. The NDVI for the time point month is inserted into result, which is of the special averaging primitive monoid type.

Sample Input

```c
C3_C_LIN clin_area = ( 3 * lon - lat >= 51 )
&& ( lat - lon <= 70 )
&& ( lon <= 65 )
&& ( lat >= 110 );
```

C3_Int32 month = 0;

Output For The Sample Input

See Appendix A.
EXEC COMPREH
SELECT { cell.NDVI } INTO { C3_Avg<double>} Result
  DEFINE area_mbr AS { clin_area.Get_MBR() }
FROM {Grid.Lat_Ind_Range(area_mbr )} AS {C3_Int32} lat_ind
FROM {Grid.Lon_Ind_Range(area_mbr )} AS {C3_Int32} lon_ind
  DEFINE delta AS { (Grid.Resolution)/2.0 }
  DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta}
  DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta}
WHERE { clin_area.TV( (lat/(center_lat), lon/(center_lon) ) ) }
  DEFINE cell AS { Grid[lat_ind][lon_ind][month] }
WHERE { cell.NDVI >= (C3_Double)0 }
;
3.2 Computing Areas with SST and Clouds Satisfying Conditions

Compute the spatial specifications of SST and CA data which satisfy the condition $290K \leq SST \leq 300K$ AND $CA \geq 50$

Input

- EOSCUBE database of AVHRR data
- cld.lwb, sst.lwb, and sst.upb, of type C3_Double
- month

Output

Result, of type C3_DC_RANGE an area at time point month, with $sst.lwb \leq SST \leq sst.upb$ AND $CA \geq cld.lwb$

Description

Iteration over the whole grid is performed. If the condition on SST and CLD is satisfied for a cell, the C_RANGE corresponding to the cell is constructed and added to the Result; the latter is of type C3_DC_RANGE which represents a union of rectangles.

Sample Input

C3_Int32 month = 0;
C3_Double cld.lwb = 70.0;
C3_Double sst.lwb = 200.0;
C3_Double sst.upb = 300.0;

Output For The Sample Input

See the output image in Appendix A.
EXEC COMPREH
SELECT { square } INTO { C3_DC_RANGE } Result
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb,1) } AS {C3_Int32} lat_in
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb,1) } AS {C3_Int32} lon_in
WHERE { Grid[lat_ind][lon_ind][month].CLD >= cld_lwb }
WHERE { Grid[lat_ind][lon_ind][month].SST >= sst_lwb }
WHERE { Grid[lat_ind][lon_ind][month].SST <= sst_upb }

  DEFINE delta AS { (Grid.Resolution)/2.0 }
  DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta}
  DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta}
  DEFINE square AS { (center_lat - delta <= lat <= center_lat + delta)
     && (center_lon - delta <= lon <= center_lon + delta) }

END COMPREH
3.3 Computing Area with SST Changes over Time

For two given months, find the spatial area where SST has increased by sst_delta.

Input

- EOSCUBE database of AVHRR data
- month_1
- month_2
- sst_delta, of type C3_Double

Output

Result, of type C3_DC_RANGE, an area where the SST has increased at least by sst_delta from time point month_1 to time point month_2

Description

Iteration over the whole grid is performed. If the condition on SST for the two given months is satisfied for a cell, the C RANGE corresponding to the cell is constructed and added to the Result; the latter is of type C3_DC_RANGE which represents a union of rectangles.

Sample Input

C3_Int32 month_1 = 0;
C3_Int32 month_2 = 1;
C3_Double sst_delta = 1.0;

Output For The Sample Input

See Appendix A.
EXEC COMPREH
SELECT \{ square \} INTO \{ C3_DC_RANGE \} Result
FROM \{ C3F_Range_Monoid\:<C3_Int32\>(0, Grid.Lat_Ind_Upb,1) \} AS \{ C3_Int32 \} lat_in
FROM \{ C3F_Range_Monoid\:<C3_Int32\>(0, Grid.Lon_Ind_Upb,1) \} AS \{ C3_Int32 \} lon_in
WHERE \{ Grid[lat_ind][lon_ind][month_2].SST - \\
    Grid[lat_ind][lon_ind][month_1].SST >= sst_delta \}
    DEFINE delta AS \{ (Grid.Resolution)/2.0 \}
    DEFINE center_lat AS \{ Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta\}
    DEFINE center_lon AS \{ Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta\}
    DEFINE square AS \{ (center_lat - delta <= lat <= center_lat + delta) \\
        && (center_lon - delta <= lon <= center_lon + delta) \}
;
END COMPREH
3.4 Time Series of Spatial Correlation between Shifted Areas - Version 1

For a selected a spatial region and a latitude/longitude shifts, (with the shifts, another region is defined; the two regions have the same shape and same size), compute a time series, that, for each month, will hold the spatial correlation (point to point) between SST over region 1 and NDVI over region 2.

Input

- EOSCUBE database of AVHRR data
- clin_area
- lat_shift, of type C3.Double
- lon_shift, of type C3.Double

Output

Result, of type C3.Vector<C3.Double>, a vector of real values representing spatial correlation for each month.

Description

The main query iterates over time first and uses a custom spatial correlation function, area_corr, which is implemented here.

Sample Input

N/A

Output For The Sample Input

N/A
Code

EXEC COMPREH
SELECT { area_corr( area, month ) } INTO { C3_Vector<C3_Double>} Result
FROM { C3F_Range_Mon(0, Max_Month-1)} AS {C3_Int32} month
;END COMPREH

This computation uses the function area_corr, which is defined next:

C3_Double area_corr ( C3_C_LIN area, C3_Int32 month ){
  sprod = sum_ndvi = sum_sst = sum_ndvi_sq = sum_sst_sq = 0;
  EXEC COMPREH
  PERFORM { sprod += cell.NDVI * new_cell.SST;
    sum_ndvi += cell.NDVI;
    sum_sst += cell.SST;
    sum_ndvi_sq += cell.NDVI* cell.NDVI;
    sum_sst_sq += cell.SST* cell.SST;
  }
  FROM { C3F_Range_Mon(0, Grid.Lat_Ind_Upb)} AS lat_ind
  FROM { C3F_Range_Mon(0, Grid.Lon_Ind_Upb)} AS lon_ind
  DEFINE lat_val AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind}
  DEFINE lon_val AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind}
  WHERE {area.TV(lat/lat_val, lon/lon_val)}
  DEFINE new_lat_ind AS {lat_ind + lat_shift/Grid.Resolution}
  DEFINE new_lon_ind AS {lon_ind + lon_shift/Grid.Resolution}
  DEFINE cell AS { Grid[lat_ind][lon_ind][month] }
  DEFINE new_cell AS { Grid[new_lat_ind][new_lon_ind][month] }
  END COMPREH
  C3_Double n = (Grid.Lat_Ind_Upb+1) * (Grid.Lon_Ind_Upb+1)
  C3_Double ndvi_avg = sum_ndvi/n;
  C3_Double sst_avg = sum_sst/n;
  double corr = ( sprod/n - ndvi_avg*sst_avg )/( sqrt
    ( sum_ndvi_sq/n-ndvi_avg*ndvi_avg)*sqrt
    ( sum_sst_sq/n-sst_avg*sst_avg))
  return (corr)
}
3.5 Time Series of Spatial Correlation between Shifted Areas - Version 2

This is another (number 2) implementation of the spatial correlation product.

Input
- EOSCUBE database of AVHRR data
- clin_area
- lat_shift, of type C3.Double
- lon_shift, of type C3.Double

Output
Result, of type C3.Vector<C3.Double>, a vector of real values representing spatial correlation for each month.

Description
Iteration over time goes first and materializes correlation sequences for each iteration. A generic correlation function, correl, is implemented here.

Sample Input
N/A

Output For The Sample Input
N/A
EXEC COMPREH
SELECT { correl(corr_seqs) }
INTO {C3_Vector<C3_Double> Result
FROM { C3F_Range_Mon(0, Max_Month-1)} AS {C3_Int32> month
DEFINE corr_seqs AS { area_corr_seqs(area,month) }
;
END COMPREH

C3_Vector<C3_Pair<C3_Double,C3_Double>>
area_corr_seqs( C3_C_LIN area, C3_Int32 month )
{
EXEC COMPREH
SELECT { C3_Pair<C3_Double,C3_Double>(cell.NDVI, new_cell.SST)
INTO {C3_Vector<C3_Pair<C3_Double,C3_Double>>} Result
FROM { C3F_Range_Mon(0, Grid.Lat_Ind_Upb)} AS lat_ind
FROM { C3F_Range_Mon(0, Grid.Lon_Ind_Upb)} AS lon_ind
DEFINE lat_val AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind}
DEFINE lon_val AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind}
WHERE {area.TV(lat/lat_val, lon/lon_val})
DEFINE new_lat_ind AS {lat_ind + lat_shift/Grid.Resolution}
DEFINE new_lon_ind AS {lon_ind + lon_shift/Grid.Resolution}
DEFINE cell AS { Grid[lat_ind][lon_ind][month] }
DEFINE new_cell AS { Grid[new_lat_ind][new_lon_ind][month] }
;
END COMPREH
return Result;
};

C3_Double correl( C3_Vector<C3_Pair<C3_Double,C3_Double>> corr_seqs )
{
  xysum = xsum = ysum = x2sum = y2sum = 0;
  PERFORM {xysum += x*y;
    xsum += x;
    ysum += y;
    x2sum += x*x;
    y2sum += y*y;
}
FROM { corr_seqs} AS pair
    DEFINE x AS pair.First
    DEFINE y AS pair.Second
END COMPREH

C3_Double n = corr_seqs.Count();
C3_Double xavg = xsum/n;
C3_Double yavg = ysum/n;
double corr = (xysum/n - xavg*yavg) /
              (sqrt(x2sum/n-xavg*xavg)*sqrt(y2sum/n-yavg*yavg))
return ( corr )

};
3.6 Time Series of Spatial Correlation between Shifted Areas - Version 3

Input
- EOSCUBE database of AVHRR data
- clin_area
- lat.shift, of type C3_Double
- lon.shift, of type C3_Double

Output
Result, of type C3_Vector<C3_Double>, a vector of real values representing spatial correlation for each month.

Description
Same as the previous product, but correlation sequences are not materialized; rather special monoid is used which incrementally implements correlation.

Sample Input
N/A

Output For The Sample Input
N/A
EXEC COMPREH
SELECT { area_corr(area, month) }
INTO { C3_Vector<C3_Double> Result }
FROM { C3F_Range_Mon(O, Max_Month-1) AS { C3_Int32 } month }
END COMPREH

C3_Double area_corr ( C3_C_LIN area, C3_Int32 month )
{
EXEC COMPREH
SELECT { C3_Pair<C3_Double, C3_Double>(cell.NDVI, new_cell.SST) }
INTO { C3_Correl<C3_Double> Result }
FROM { C3F_Range_Mon(O, Grid.Lat_Ind_Upb) AS lat_ind }
FROM { C3F_Range_Mon(O, Grid.Lon_Ind_Upb) AS lon_ind }
DEFINE lat_val AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind }
DEFINE lon_val AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind }
WHERE {area.TV(lat/lat_val, lon/lon_val)}
DEFINE new_lat_ind AS { lat_ind + lat_shift/Grid.Resolution }
DEFINE new_lon_ind AS { lon_ind + lon_shift/Grid.Resolution }
DEFINE cell AS { Grid[lat_ind][lon_ind][month] }
DEFINE new_cell AS { Grid[new_lat_ind][new_lon_ind][month] }
;
END COMPREH
return Result;
}
3.7 Time Series of Spatial Correlation between Shifted Areas - Version 4

Input

- EOSCUBE database of AVHRR data
- clin_area
- lat_shift, of type C3_Double
- lon_shift, of type C3_Double

Output

Result, of type C3_Vector<C3_Double>, a vector of real values representing spatial correlation for each month.

Description

This more efficient version iterate over cells first and computes all correlation values simultaneously.

Sample Input

C3_C_LIN clin_area = ( 100 <= lat <= 110 && 50 <= lon <= 60 ); // part of No
C3_Double lon_shift = -60 ; // into Pacific ocean
C3_Double lat_shift = -20 ;

Output For The Sample Input

See Appendix A.
EXEC COMPREH
    DEFINE area_mbr AS { clin_area.Get_MBR() }
FROM {Grid.Lat_Ind_Range(area_mbr )} AS {C3_Int32} lat_ind
PERFORM { C3M_TRACE(lat_ind) }
FROM {Grid.Lon_Ind_Range(area_mbr )} AS {C3_Int32} lon_ind
    DEFINE delta AS { (Grid.Resolution)/2.0 }
    DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta }
    DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta }
WHERE { clin_area.TV( (lat/(center_lat), lon/(center_lon) ) ) }
    DEFINE new_lat_ind AS {lat_ind + lat_shift/Grid.Resolution}
    DEFINE new_lon_ind AS {lon_ind + lon_shift/Grid.Resolution}
    DEFINE cell_ts AS { Grid[lat_ind][lon_ind] }
    DEFINE new_cell_ts AS { Grid[new_lat_ind][new_lon_ind] }
SELECT {C3_Pair<double,double> (cell_ts[month].NDVI, 
                        new_cell_ts[month].SST)}
INTO correl
FROM {C3F_Range_Monoid<C3_Int32>(0,Max_Months-1,1)} AS {C3_Int32} month
    DEFINE correl AS { Result[month] }
;
3.8 Temporal Correlation

For time periods when SOI < 0 (or >= 0), compute the spatial region over which the temporal correlation between NDVI and SOI is equal to or greater than a given value tc_lwb.

Input

- EOSCUBE database of AVHRR data
- tc_lwb, of type C3_Double

Output

Result, of type C3_Double RANGE, an area where the temporal correlation between SOI and NDVI over the 24 months is at least tc_lwb

Description

straightforward

Sample Input

C3_Double tc_lwb = 0.6 ;

Output For The Sample Input

See the outpour image in Appendix A.
EXEC COMPREH
SELECT { square } INTO { C3_DC_RANGE } Result
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb, 1) } AS { C3_Int32 } lat_i:
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb, 1) } AS { C3_Int32 } lon_in:
    DEFINE cell_ts AS { Grid[lat_ind][lon_ind] }
    SELECT { C3_Pair<double,double>( cell_ts[month].NDVI, SOI_Time_Series[month] ) } INTO { C3_Correl<double> } correl
    FROM { C3F_Range_Monoid<C3_Int32>(0, Max_Months-1,1) } AS { C3_Int32 } month:
    WHERE { cell_ts[month].NDVI >= 0.0 }
    WHERE { SOI_Time_Series[month] >= 0.0 }
    WHERE { (double)correl >= tc_lwb }
    DEFINE delta AS { (Grid.Resolution)/2.0 }
    DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta }
    DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta }
    DEFINE square AS { (center_lat - delta <= lat <= center_lat + delta) 
        && (center_lon - delta <= lon <= center_lon + delta) }
;
END COMPREH
3.9 Spatial Aggregation

Produce a new grid which is $N \times N$ times more coarse for a given month.

**Input**
- EOSCUBE database of AVHRR data
- $N$, of type C3\_Int32
- month

**Output**
Result, of type EOS3\_Earth\_Grid, a new coarse grid with the resolution $N$ the input grid resolution (at time point month), and NDVI values averaged for each coarse cell over finer cells inside it.

**Description**
Iteration over coarse cells goes first, then corresponding finer cells are visited and NDVI is averaged using the averaging monoid.

**Sample Input**
C3\_Int32 month = 0;
C3\_Int32 $N = 10$;

**Output For The Sample Input**
See Appendix A.
Code

EOS3_Earth_Grid New_Grid( 360/N, 180/N );
New_Grid.Size_All();

EXEC COMPREH
PERFORM
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb/N, 1) }
    AS {C3_Int32} new_lat_ind
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb/N, 1) }
    AS {C3_Int32} new_lon_ind

    SELECT { cell.NDVI }
    INTO {C3_Avg<double>} avg_ndvi
    FROM { C3F_Range_Monoid<C3_Int32>(new_lat_ind*N, (new_lat_ind+1)*N-1, 1) }
        AS {C3_Int32} lat_ind
    FROM { C3F_Range_Monoid<C3_Int32>(new_lon_ind*N, (new_lon_ind+1)*N-1, 1) }
        AS {C3_Int32} lon_ind

    DEFINE cell AS {Grid[lat_ind][lon_ind][month]}
    WHERE { cell.NDVI >= 0.0 }
;
    PERFORM {New_Grid[new_lat_ind][new_lon_ind][month].NDVI = (C3_Double)avg_ndvi }
;
END COMPREH
3.10 NDVI Animation Movie

Produce a sequence of areas with NDVI at least ndvi_lwb for 24 months.

Input

- EOSCUBE database of AVHRR data
- ndvi_lwb, of type C3.Double

Output

Result, of type C3.Vector<C3.DCLIN>, representing a collection of areas with NDVI equal or greater than ndvi_lwb

Description

Iteration over all time points is performed, then NDVI.LESS(ndvi_lwp) is called on each iteration

Sample Input

C3_Double ndvi_lwb = 0.4;

Output For The Sample Input

See Appendix A + animation in the demo.
EXEC COMPREH
SELECT { frame } INTO { C3_Vector<C3,DC,RANGE> } Result
FROM { C3F.Range_Monoid<C3_Int32>(0,Max_Months-1,1) } AS { C3_Int32 } month
    SELECT { square } INTO { C3_DC_RANGE } frame
    FROM { C3F.Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb, 1) }
        AS { C3_Int32 } lat_ind
    FROM { C3F.Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb, 1) }
        AS { C3_Int32 } lon_ind
WHERE { Grid[lat_ind][lon_ind][month].NDVI >= ndvi_lwb }
    DEFINE delta AS { (Grid.Resolution)/2.0 }
    DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta }
    DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta }
    DEFINE square AS { (center_lat - delta <= lat <= center_lat + delta)
    && (center_lon - delta <= lon <= center_lon + delta )
    ;
    END COMPREH

30
3.11 Temporal Correlation and SST Anomaly

Compute the spatial area in which the temporal correlation between SST Anomaly and CA is greater than or equal to tc_lwb.

Input

- EOSCUBE database of AVHRR data
- tc_lwb

Output

Result of type C3.DC.RANGE which represents the computed area.

Description

Note, anomaly here is a difference between a SST value and SST 2-year average.

Sample Input

N/A

Output For The Sample Input

N/A.
EXEC COMPREH
SELECT { square } INTO { C3_DC_RANGE } Result
FROM { C3F_Range_Mon<Int32>(0, Grid.Lat_Ind_Upb,1)} AS {Int32} lat_ind
FROM { C3F_Range_Mon<Int32>(0, Grid.Lon_Ind_Upb,1)} AS {Int32} lon_ind
SELECT { Grid[lat_ind][lon_ind][month].SST }
INTO {C3_Avg<double> } sst_avg
FROM { C3F_Range_Mon<Int32>=>(0, Max_Month-1,1)} AS {Int32} month
WHERE { cell_ts[month].SST >= 0 }
;
SELECT { C3_Pair<double,double>( cell_ts[month].CLD, sst_anomaly ) }
INTO {C3_Correl<double> } correl
FROM { C3F_Range_Mon<Int32>=>(0, Max_Month-1,1)} AS {Int32} month
WHERE { cell_ts[month].CLD >= 0 }
WHERE { cell_ts[month].SST >= 0 }
    DEFINE sst_anomaly AS { Grid[lat_ind][lon_ind][month].SST-sst_avg}
;
WHERE { (double)correl >= correl_lwb }
    DEFINE delta AS { (Grid.Resolution)/2.0 }
    DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta}
    DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta}
    DEFINE square AS { (center_lat - delta <= lat <= center_lat + delta)
        && (center_lon - delta <= lon <= center_lon + delta)}
;
END COMPREH
3.12 Color World Map

Create three spatial areas: green, yellow and red, classified by NDVI ranges, per given month

Input
- EOSCUBE database of AVHRR data
- month

Output
- green_area of type C3_DC_RANGE
- yellow_area of type C3_DC_RANGE
- red_area of type C3_DC_RANGE

Description
Straightforward.

Sample Input
N/A

Output For The Sample Input
N/A
Code

C3_DC_RANGE green_area, yellow_area, red_area

EXEC COMPREH
SELECT square(lat_ind, lon_ind, Grid.res)
INTO {C3_DC_RANGE} green_area
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb,1)} AS {C3_Int32} lat_in
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb,1)} AS {C3_Int32} lon_in
WHERE {Grid[latt_ind][lon_ind][month].NDVI <= green_NDVI_bound}
;

SELECT square(lat_ind, lon_ind, Grid.res)
INTO {C3_DC_RANGE} yellow_area
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb,1)} AS {C3_Int32} lat_in
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb,1)} AS {C3_Int32} lon_in
WHERE {Grid[latt_ind][lon_ind][month].NDVI > green_NDVI_bound &&
 Grid[latt_ind][lon_ind][month].NDVI <= yellow_NDVI_bound}
;
SELECT square(lat_ind, lon_ind, Grid.res)
INTO {C3_DC_RANGE} red_area
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb,1)} AS {C3_Int32} lat_in
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb,1)} AS {C3_Int32} lon_in
WHERE {Grid[latt_ind][lon_ind][month].NDVI > yellow_NDVI_bound}
;
END COMPREH

C3_C_RANGE square(int lat_ind, int lon_ind, double res);
{C3_CST_Var lat, lon;
return(Grid.lat_low + lat_ind*res <= lat &&
 Grid.lat_low + (lat_ind + 1)*res &&
 Grid.lon_low + lon_ind*res <= lon &&
 Grid.lon_low + (lon_ind + 1)*res
}
3.13 Coloring Counties based on NDVI Means

Color USA counties by mean NDVI values

Input

- EOSCUBE database of AVHRR data
- EOSCUBE database of USA counties
- month

Output

Result of type C3_Vector<C3_Pair<C3_String, C3_String>> to hold pairs of county name and the associated color, for each county.

Description

See the description of the USA database of counties in the EOSCUBE Language And Features section

Sample Input

N/A

Output For The Sample Input

N/A
EXEC COMPREH
SELECT { C3_Pair<C3_String,C3_String>(county.name, Color(ndvi_val))} INTO { C3_Vector<C3_Pair<C3_String,C3_String>> } Result
FROM { all_counties } AS (County) county
  SELECT { cell.ndvi } INTO { C3_Avg<double> } ndvi_val
  DEFINE area_mbr AS { county.extent.Get_MBR() }
  FROM {Grid.Lat_Ind_Range(area_mbr ) AS {C3_Int32} lat_ind
  FROM {Grid.Lon_Ind_Range(area_mbr ) AS {C3_Int32} lon_ind
    DEFINE delta AS { (Grid.Resolution)/2.0 }
    DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta }
    DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta }
    DEFINE square AS { (center_lat - delta <= lat <= center_lat + delta)
      && (center_lon - delta <= lon <= center_lon + delta))
WHERE { area.TV( lat/lat_val, lon/lon_val) }
    DEFINE cell AS { Grid[lat_ind][lon_ind][month] }
WHERE { cell.NDVI >= 0 }

END COMPREH

C3_String Color( C3_Double value )
{
  if ( (value <= 0.3 ) )
  { return "green";
  }
  else if ( (value <= 0.5 ) && (value >= 0.3 ) )
  { return "yellow";
  }
  else if ( (value >= 0.5 ) )
  { return "red";
      }
3.14 Counties and NDVI Coverage

Find the counties where areas of NDVI coverage of ndvi_lwb are at least the given percentage of the county's total area.

Input
- EOSCUBE database of AVHRR data
- EOSCUBE database of USA counties
- month
- ndvi_lwb
- percentage

Output

Description
straightforward

Sample Input
N/A

Output For The Sample Input
N/A
EXEC COMPREH
SELECT { square } INTO { C3_DC_RANGE }  ndvi_coverage
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lat_Ind_Upb,1) } AS {C3_Int32} lat_in
FROM { C3F_Range_Monoid<C3_Int32>(0, Grid.Lon_Ind_Upb,1) } AS {C3_Int32} lon_in
WHERE { Grid[lat_ind][lon_ind][month_1].NDVI >= ndvi_lwb }
    DEFINE delta AS { (Grid.Resolution)/2.0 }
    DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta }
    DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta }
    DEFINE square AS { (center_lat - delta <= lat <= center_lat + delta) 
                    && (center_lon - delta <= lon <= center_lon + delta) }
;
END COMPREH

EXEC COMPREH
SELECT { county.Name }
INTO { C3_Vector<C3_String> } Result
FROM { all_counties } AS {County} county
    DEFINE isec AS { county.extent && ndvi_coverage }
WHERE { SQ(isec)/SQ(county.extent) >= percentage }
;
END COMPREH

38
3.15 Sort in Drop Order

Sort the counties in the order of NDVI drop from the period of the first 12 months to the period of the last 12 months.

Input

- EOSCUBE database of AVHRR data
- EOSCUBE database of USA counties

Output

Result of type `Sorted_Vector<String, Double>` to hold county names sorted in the drop order.

Description

straightforward

Output For The Sample Input

See Appendix A.
EXEC COMPREH
SELECT {C3_Pair<C3_String, C3_Double>(county.Name, avg_drop) }
INTO {Sorted_Vector<C3_String, C3_Double>} Result
FROM {all_counties} AS {County} county
   DEFINE area_mbr AS { county.extent.Get_MBR() }
FROM {Grid.Lat_Ind_Range(area_mbr ) AS {C3_Int32} lat_ind
FROM {Grid.Lon_Ind_Range(area_mbr ) AS {C3_Int32} lon_ind
   DEFINE delta AS { (Grid.Resolution)/2.0 }
   DEFINE center_lat AS { Grid.Low_Lat+(Grid.Resolution)*lat_ind + delta
   DEFINE center_lon AS { Grid.Low_Lon+(Grid.Resolution)*lon_ind + delta
   DEFINE square AS { (center_lat - delta <= lat <= center_lat + delta)
   && (center_lon - delta <= lon <= center_lon + delta))
WHERE { area.TV( lat/lat_val, lon/lon_val) }
   DEFINE cell_ts AS { Grid[lat_ind][lon_ind] }
SELECT { cell_ts[month].NDVI - cell_ts[month+Max_Month/2].NDVI }
INTO {C3_Avg<C3_Double>} avg_drop
FROM { C3F_Range_Mon(0, Max_Month-1) } AS month
;
END COMPREH

Sample Input

40
3.16 NDVI Composition Algorithm

In this section we show an EOSCUBE implementation of the Vegetation Composition Algorithm. The main task of the algorithm is to analyze the input stream of pixel measurements and to apply the corresponding VI computation method based on the number of well measured pixels in each composition period. The detailed description of the algorithm can be found in the Vegetation Index ATBD document, and we will not repeat it here.

This EOSCUBE implementation of the algorithm demonstrates the following EOSCUBE benefits:

- the EOSCUBE program reads as a very well-structured high-level procedure; the correspondence between EOSCUBE code and the data flow diagram in the Vegetation Index ATBD document is very clear

- Using the EOSCUBE language FROM clauses and the RANGE monoid for iteration over the stream indices hides all details of loop creation of a traditional programming language, such as checking if the index is in the range or index increment, thus making the program more compact and readable

- Use of built-in COUNT, MAX, and MAXPAIR monoids allows to perform the operations of counting elements, finding minimums and maximums in just one line of code and within the model of the query language. Also each of those monoids incorporates pieces of the code that would otherwise require a separate implementation

We assume that the input stream is represented by a two-dimensional vector all_pixels. The first dimension is spatial and corresponds to all pixels, the second is the temporal dimension with daily resolution. The type of the elements is Pixel_Data which contains various input parameters required for the VI Composition Algorithm, such as reflectivities, angles and quality control data. The Pixel_Data type is shown below:

```cpp
class Pixel_Data {
    float ro_nir;
    float ro_red;
    float theta_v;
    float theta_s;
    float phi_v;
    float phi_s;
    QC qc;
};
```
The following EOSCUBE product implements the main part of the VI Composition Algorithm, following the algorithm description in the Vegetation Index ATBD document:

```plaintext
EXEC COMPREH
FROM { C3F_Range_Monoid<C3_Int32>(0,max_pixels,1) } AS {C3_Int32} pix_ind
FROM { C3F_Range_Monoid<C3_Int32>(0,max_days/8,1) } AS {C3_Int32} 8_ind
DEFINE Good_Pix_Count AS
SELECT { day_ind } INTO {Count<C3_Int32>}
FROM { C3F_Range_Monoid<C3_Int32>(8_ind*8, 8_ind*8+7,1) } AS {C3_Int32} d
WHERE { Is_Good( all_pixels[pix_ind][day_ind] ) }
DEFINE NDVI_value AS {
    switch ( Good_Pix_Count ) {
        case >= 5 : BRDF(pix_ind, 8_ind)
        case <= 5 && > 0 : Max_NDVI_Angle(pix_ind, 8_ind)
        case 0 : Max_NDVI(pix_ind, 8_ind)
    }
PERFORM Result[pix_ind][day_ind] = NDVI_value
END COMPREH
```

The programs `Max_NDVI_Angle(pix_ind, 8_ind), Max_NDVI(pix_ind, 8_ind)` and `BRDF(pix_ind, 8_ind)` are implemented as described in the Vegetation Index ATBD document and specify different NDVI composition methods. Below we give an implementation for the `Max_NDVI_Angle(pix_ind, 8_ind)` and `Max_NDVI(pix_ind, 8_ind)` programs:

```plaintext
DEFINE Max_NDVI_Angle(pix_ind, 8_ind) AS
    DEFINE ndvi_pair AS
    SELECT C3_Pair<all_pixels[pix_ind][day_ind].theta_v, NDVI(all_pixels[pix_ind][day_ind])>
    INTO {MINPAIR<C3_Double,C3_Double>} ndvi_pair
    FROM RANGE(8_ind*8, 8_ind*8+7) AS {C3_Int32} day_ind
    MAX( ndvi_pair.1st, ndvi_pair.2nd)
END

DEFINE Max_NDVI(C3_int32 pix_ind, C3_int32 8_ind) AS
    SELECT NDVI(all_pixels[pix_ind][day_ind])
    INTO {MAX<C3_Double>} Max_NDVI
    FROM RANGE(8_ind*8, 8_ind*8+7) AS {C3_Int32} day_ind
END
```

42
Here MINPAIR is monoid which computes two smallest numbers in a collection and is capable of storing additional information corresponding to those numbers, which is a pair of NDVI values in this case. The NDVI function encodes the formula computing NDVI out from pixel data using one of the formulas described in the Vegetation Index ATBD document.
Part III

EOSCUBE Features and Language

4 Introduction

Constraints provide a flexible and uniform way to conceptually represent diverse data capturing spatio-temporal behavior, complex modeling requirements, partial and incomplete information etc, and have been used in a wide variety of application domains. Constraint databases (CDBs) have recently emerged to deeply integrate data captured by constraints in databases. Although a relatively new realm of research, constraint databases have drawn much attention and increasing interest, mostly in aspects of expressibility and complexity, but also in algorithms and optimization.

Constraint databases are very promising for applications requiring to support large heterogeneous data sets that can be uniformly captured by constraints. This includes (1) engineering design; (2) manufacturing and warehouse support; (3) electronic trade with complex objectives; (4) command and control (such as spatio-temporal data fusion and sensor management [ABK95] and maneuver planning [BVCS93]); (5) distribution logistics; and (6) market analysis.

While many fundamental research questions are yet to be answered, we believe that the area of constraint databases became mature for a reliable research prototype that could serve as a stable platform for experimentation with algorithms and optimization techniques as well as for real-life case studies of a number of promising application domains. We believe that building such a system is a crucial step toward proving the validity of constraint databases as a technology with a significant practical impact.

Motivation and Design Goals

Until now, most of the work on CDBs has been theoretical (see Related Work section). CDB researchers are being challenged by the question whether the CDB technology can really work on real-life, real-size, real-performance applications, or it is just an intellectual toy that will eventually fade away. This parallels, in a sense, to the state of relational databases before the first two prototypes, Ingres and System-R, were developed. Our view is that the viability test for the CDB technology will be the ability to achieve competitive performance and scalability. Therefore, algorithms, data structures and optimization techniques are the most critical issues.

At the beginning of the EOSCUBE work we had two main choices regarding the design objectives: (1) to naively implement a high-level and purely declarative
constraint DB language, such as Lyric, focusing on its interface, but ignoring the performance and scalability, or (2) to develop an extensible infrastructure (i.e. an intermediate, optimization-level language, in which evaluation plans can be explicitly expressed) suitable for developing and testing optimization techniques, algorithms and data structures.

The first choice would lead to a much faster and simpler implementation, and, in fact, this is the way most research prototypes are implemented. The second choice, clearly, is considerably more work- and time-intensive, but is essential for our overall objectives. Of course, ideally, we would like to have both a high-level language and a full-scale optimizer in place, but this would not be possible without developing first the optimization infrastructure of (2). This direction was indeed our choice.

The following were our main design principles:

1. In terms of constraint domains and operators, a careful balance between expressiveness and complexity must be achieved. It is easy to fall into a trap of highly expressive constraint domains for almost any imaginable types of data, but with absolutely impractical complexity.

2. The language and the model should be object-oriented, since many object-oriented features are important for the target applications (e.g. [ABK95]).

3. The language should be suitable for explicitly expressing highly optimized evaluation plans (preserving their I/O time and space complexity). It must be flexible enough to support object-oriented optimization (e.g. ENCORE [Zdo89], O2 [ea90], POSTGRES [SRH90]), constraint database optimization (e.g. [BJM93]) and constraint indexing and filtering (e.g. [BW95]).

4. The system should be extensible with respect to (constraint and other) data types, operators and predicates, and special data structures and algorithms.

5. The system should allow easy interaction with an underlying programming language in order to be usable directly by system or application programmers.

6. Provided the previous principles are met, the language should ideally be as high-level and easy-to-use as possible.

EOSCUBE Features and Architecture

The EOSCUBE data manipulation language, Constraint Comprehension Calculus is an integration of a constraint calculus for extensible constraint domains within monoid comprehensions, which were suggested as an optimization-level language for object-oriented queries [FM95]. In the following, when no misinterpretation arises, we will
be using the same name EOSCUBE for both the system and the language of the constraint comprehension calculus. ¹

The data model for constraint calculus is adapted from constraint spatio-temporal (CST) objects [BK95], that may hold spatio-temporal constraint data, conceptually represented by constraints (i.e. symbolic expressions). In the current version, linear arithmetic constraints (i.e. inequalities and equations) over reals ² are implemented. New CST objects are constructed using logical connectives, existential quantifiers and variable renaming, within a mutli-typed constraint algebra.³ The constraint module also provides predicates such as for testing satisfiability, entailment etc, that are used as selecting conditions in hosting monoid comprehension queries.

CST objects possess great modeling power and as such can serve as a uniform data type for conceptual representation of heterogeneous data, including spatial and temporal behavior, complex design requirements and partial and incomplete information. Moreover, the constraint calculus operating on CST is a highly expressive and compact language. For example, just linear arithmetic CSTs and its calculus currently implemented in the system, allow description and powerful manipulation of a wide variety of data, including (1) 2- or 3-D geographic maps; (2) geometric modeling objects for CAD/CAM; (3) fields of vision of sensors; (4) 4-D (3 + 1 for time) trajectories of objects moving in a 3-D space, based on the movement equations; (5) translations of different system of coordinates; and (6) operations research type models such as manufacturing patterns describing interconnections between quantities of manufactured products and resource materials. It is important to note that the conceptual and physical representations of CST objects are orthogonal: while conceptually constraints are viewed as symbolic expressions, the physical representation is typically chosen to facilitate efficient storage and manipulation.

The general framework of the EOSCUBE language is the monoid comprehensions language, in which CST objects serve as a special data type, and are implemented as a library of interrelated C++ classes. The data model for the monoid comprehensions is based on the notion of monoid, which is a conceptual data type capturing uniformly aggregations, collections, and other types over which one can “iterate”. This includes (long) disjunctions and conjunctions of constraints.

The ability to treat disjunctive and conjunctive constraints uniformly as collections is a very important feature of EOSCUBE: it allows to express and implement many constraint operations through nested monoid comprehensions, i.e. in the same language as hosting queries. For example, the satisfiability test of a disjunction of

---

¹In fact, the name EOSCUBE was originated from the shorthand C³ for Constraint Comprehension Calculus.

²using finite precision arithmetic

³As explained in later sections, users can view the constraint layer as either calculus, or algebra, interchangeably.
conjunctions of linear inequalities is expressed as a monoid comprehension query that iterates over the disjuncts (each being a conjunction), and tests the satisfiability of every conjunction (using the simplex algorithm).

In turn, the ability to express a constraint operation as a sub-query in the hosting query is crucial for what we call *deeply interleaved optimization*: it gives the flexibility to re-shuffle and interleave parts of the constraint algorithm (sub-query) with the hosting query. This re-shuffling can be done by additional global query transformations (discussed in the paper) involving approximations, indexing, re-grouping, pushing cheaper selections earlier, replacing sub-queries with special-purpose algorithms, and so forth. Figuratively speaking, constraint operations are not treated in EOSCUBE as *black boxes* plugged into a query, which would severely restrict optimization opportunities, but rather as *white boxes with black holes*.

The EOSCUBE system architecture supports:

1. Besides CST objects, any data structures expressible in C++.

2. An extensible family of parameterized and possibly nested collection monoids currently including sets, bags, lists, as well as (long) disjunctions and conjunctions of CST objects.

3. An extensible family of aggregation monoids such as *sum*, *count*, *some* and *all*.

4. An extensible family of search structures implemented as parameterized monoids and currently including B-trees, hashing and kD-trees for multidimensional rectangles. Because the search structures are implemented as monoids, they can be used uniformly anywhere in queries where monoids are allowed.

5. An extensible family of special-purpose algorithms, such as the sort-join and the constraint join [BJM93], which are implemented as parameterized monoids. These special algorithms are important for performance because they cannot be matched, in terms of I/O complexity, by standard monoid comprehension algorithms with dynamic buffer management (although many other algorithms, such as the loop join and standard selections, can). Since the special algorithms are expressed as monoids, they can be easily plugged in monoid comprehension queries to replace equivalent sub-queries.

6. Approximation-based filtering, indexing and regrouping based on internal components of nested collection monoids. These features are especially important for achieving deeply interleaved optimization in presence of constraint operations.
7. Orthogonal features inherited from the commercial OODB ObjectStore, including persistence, dynamic buffer management, transaction management, data integrity, crash recovery, version management, and multi-client/multi-server architecture.

The functionality of the EOSCUBE system is a combination of the new EOSCUBE layer and ObjectStore. EOSCUBE as a virtual system (1) inherits "lower" features of ObjectStore, (2) replaces "middle" ObjectStore's features with those of the EOSCUBE layer, and (3) adds "upper" features of the EOSCUBE layer. The implementation of the EOSCUBE layer, in turn, uses ObjectStore and the linear programming package CPLEX. We feel important to note that while EOSCUBE is a research prototype, we believe that it is a scalable system designed to carry out implementations of serious, massive data applications. That is partly due to the use of commercially available components (i.e. ObjectStore and CPLEX).

EOSCUBE, similar to ObjectStore, can be better viewed as a powerful extension of C++ with constraint database features, rather than a full-scale DBMS, and is currently to be used from within a hosting C++ program. As a C++ extension, EOSCUBE uses the native C++ data structures and its type system. In fact, in the current implementation only the monoid comprehensions are precompiled, and all the other EOSCUBE features, including the the constraint calculus, are implemented as C++ libraries; hence the native C++ syntax is preserved.

The use of the "dirty" C++ data model, as opposed to "clean" and formally defined models such as of ODMG OQL [ABD+96] or XSQL [KKS92] was our pragmatic choice due to the intended purpose of EOSCUBE: an intermediate optimization-level language, i.e. one in which an optimizer or a programmer can (explicitly) write highly optimized queries, using appropriate order, nesting, special operators (e.g. for the sort join) and built-in optimization primitives. Because of the intended use as an intermediate language, we prefer to regain the flexibility of and the uniformity with the underlying programming language, C++. We designed EOSCUBE to be used both for the implementation and optimization of high-level constraint object-oriented query languages such as Lyric or constraint extensions of OQL, and for directly building software systems (by application or system programmers) requiring extensible use of constraint database features.

The focal point of our work is achieving the right balance between the expressiveness, complexity and representation usefulness [Bro] without which the practical use of the system would not be possible. To that end, the EOSCUBE constraint calculus guarantees polynomial data complexity, and, furthermore, is tightly integrated with the monoid comprehensions to allow deeply interleaved global optimization.
5 CST Objects and EOSCUBE Queries by Example

In this section we informally discuss EOSCUBE queries, including CST objects, the constraint calculus and monoid comprehensions using an Earth image example. We assume, briefly, that a database stores a collection of country (or county) maps, which have extents (or shapes). It also stores an Earth image which is an 1x1 degree resolution grid of pixels. We assume that there are parameter values (like vegetation index, cloud amount, or see surface temperature) associated with each pixel. A scientist then may ask queries that output spatial and temporal descriptions of areas where the parameters satisfy a certain condition. Those queries can be compactly answered in EOSCUBE without using user implemented predicates or functions.

5.1 Constraints, CST objects and Schema by Example

Consider application schema for our earth image example (see Appendix B). The schema uses regular object-oriented features, and also what we call spatio-temporal constraint (CST) classes, such as CST\( (\text{lon}, \text{lat}) \), which means, intuitively, a constraint in free variables \( \text{lon} \) and \( \text{lat} \). Consider also a county which is approximated by a 2D conjunctive constraint in the latitude-longitude linear system of coordinates \( \text{lat}, \text{lon} \). For example, a constraint (formula)

\[
(-4 \leq \text{lat} \leq 4) \land (-2 \leq \text{lon} \leq 2)
\]

with the variables ranging over reals can be viewed as a set of points

\[
\{(\text{lat}, \text{lon})|(-4 \leq \text{lat} \leq 4) \land (-2 \leq \text{lon} \leq 2)\}
\]

in two-dimensional space and describes, say, the extent of a small rectangular area. More accurately, the constraint formula \((-4 \leq \text{lat} \leq 4) \land (-2 \leq \text{lon} \leq 2)\) will be interpreted as an infinite relation over the schema \( \text{lat}, \text{lon} \), that contains all tuples \((\text{lat}, \text{lon})\) satisfying the constraint.

In the EOSCUBE syntax the above formula will look as follows:

\[
(-4 \leq \text{lat} \leq 4) \&\& (-2 \leq \text{lon} \leq 2)
\]

We use \&\& and == instead of \& and =, correspondingly, to preserve the C++ style. Interestingly, the above constraint syntax is native in C++, which is achieved by
exploiting the C++ operators' overloading mechanism. How this is done is explained in more detail in Section 7.3. Users can intuitively think of a constraint with $d$ free variables as a (possibly) infinite relation of $d$-tuples, as an object in $d$-dimensional space (i.e. a set of points), or as a symbolic expression, interchangeably, depending on the application and the context of its use. Thus, we will be referring to a constraint by a generic name CST (i.e. *constraint spatio-temporal*) object.

5.2 EOSCUBE Queries by Example

Consider the following EOSCUBE query, yet without CST objects, which finds a bag of all countries colored in red:

```c++
SELECT c // for file cabinet
INTO {Bag<COUNTRY*>>} result // result is a bag-collection
FROM all_countries
AS {COUNTRY*} c // iterator: fc iterates
WHERE c->color == 'red' // predicate, i.e. condition
```

The SELECT clause is followed by a possibly interleaved list of the FROM-clause iterators and WHERE-clause conditions. Any order of iterators and predicates, in which variables are only used after they are bound is allowed. However, in general, different orders may lead, as we shall see, to different resulting collections. In the SELECT clause we may have any C++ expression, possibly using variables bounded in the iterators, or invoking another monoid comprehension.

The semantics of the query is best understood, intuitively, through the following loop program, which is a conceptual skeleton of the actual algorithm evaluating monoid comprehensions.4

```c++
result = empty_bag;
FOREACH c IN {BAG} all_countries DO
    IF c->color == 'red'
    THEN
        INSERT c INTO result
```

Also important to note is that a query can be always written with just one FROM clause containing all the iterators, followed by just one WHERE clause with all the conditions. However, we allow any order of interleaved iterators and conditions, in

---

4The real algorithm also deals with many other issues such as persistence, dynamic buffer management, type management and interface with C++ etc.
order to control the evaluation of the query, as is necessary for optimization-level
languages such as EOSCUBE.

Consider an example of an earth area where its extent in latitude and longitude
coordinates is the set of points \{((lon, lat)|(100 \leq lat \leq 110) \land (50 \leq lon \leq 60))\} which
is captured as the CST object \((100 \leq lat \leq 110) \land (50 \leq lon \leq 60)\) in free variables
\(lat, lon\), i.e., of class \(\text{CST}(lat, lon)\).

The following EOSCUBE query, for a given area, computes the intersection of this
area with each country, then finds VI for each intersection using the mean average,
and produces a collection of pairs containing the computed value and the correspond-
ing intersection scaled down by 2 on all coordinates. This query also shows that
constraints are used in EOSCUBE queries to manipulate, as well as express boolean
conditions on CST objects:

\[
\begin{align*}
\text{CST area} &= (100 \leq \text{lat} \leq 110 \land 50 \leq \text{lon} \leq 60); \\
\text{SELECT pair}(\text{avg}, \text{isec}.\text{Subst}(\text{lon/lon}*0.5, \text{lat/lat}*0.5)) & \text{ INTO \{Bag<pair\} result} \\
\text{FROM all_countries AS c} \\
\text{DEFINE isec AS \{CST\} c.extent \&\& area} \\
\text{WHERE SAT(isec)} \\
\text{SELECT pix.VI INTO \{AVG<real\} avg} \\
\text{FROM Grid AS \{PIXEL\} pix} \\
\text{DEFINE rect AS (pix.lat-0.5 \leq \text{lat} \leq pix.lat+0.5) \&\&} \\
(pix.lon-0.5 \leq \text{lon} \leq pix.lon+0.5) \\
\text{WHERE SAT(rect \&\& isec)}
\end{align*}
\]

Variable names are considered to be a part of the constraint database schema and
are analogous to attribute names in the relational model. In constraint formulas
the system joins variables that has the same symbolic name. Of course, there are
situations when explicit renaming is required to achieve our goals (for example when
making a self-join of two conjunctions). Expressions \text{DEFINE expr1 AS expr2} cause
the replacement \text{expr1} by \text{expr2} in the remainder of the comprehension; they are used
simply as shortcuts. SAT stands for the satisfiability test of the constraint expression
inside the parentheses which checks whether \text{rect intersects isec}.

Below we describe in more detail different types of queries and their syntax.

Create-a-result query

\text{EXEC \text{COMPREH}}

51
SELECT { cpp_expr_1 } INTO { cpp_type_1 } cpp_var_1
FROM { cpp_expr_2 } AS { cpp_type_2 } cpp_var_2
DEFINE cpp_var_3 AS { cpp_expr_3 }
PERFORM { cpp_expr_4 }
<select_subquery>
WHERE { cpp_expr_5 }
;
END COMPREH

cpp_type_1 is a C++ type of the resulting monoid cpp_var_1, which is allocated by EOSCUBE; cpp_expr_1 must yield the type of the elements of the result; cpp_expr_2 must yield a collection monoid with element type cpp_type_2; cpp_var_3 refers to cpp_expr_3 by textual substitution and can be used in the rest of the query; cpp_expr_4 is any expression to be executed; cpp_expr_5 must yield bool or int types that are used as conditions; <select_subquery> denotes any CCUBE query that can be inserted into the main query; the semicolon wraps up the SELECT (sub)query

Append-to-result query
Same as the create-a-result query, but the INTO clause omits the monoid type:

EXEC COMPREH
SELECT { cpp_expr_1 }
INTO cpp_var_1
FROM { cpp_expr_2 } AS { cpp_type_2 } cpp_var_2
DEFINE cpp_var_3 AS { cpp_expr_3 }
PERFORM { cpp_expr_4 }
<select_subquery>
WHERE { cpp_expr_5 }
;
END COMPREH

The resulting variable cpp_var_1 is not created here; it is assumed to have been defined previously;

Subqueries
A subquery may be put inside another query just like another clause
EXEC COMPREH

... 
SELECT ... // main query
...
SELECT ... // subquery
...
; //this semicolon wraps the inner query
...
; //this semicolon wraps the outer query
END COMPREH

Action queries

An action query does not create or append the result. It just executes C++ statements, indicated in the PERFORM clause. Note, there is no SELECT clause in an action query. Action queries are typically used for modifying collections or printing results.

EXEC COMPREH
PERFORM
FROM { cpp_expr_1 } AS { cpp_type } iter_name
PERFORM { cpp_expr_2 }
END COMPREH

The first PERFORM statement without parameters indicates to EOSCUBE that it is an action query. The second perform statement executes cpp_expr_2 for each iteration.
Part IV
EOSCUBE Background, Architecture and Implementation

6 EOSCUBE Monoids and Monoid Comprehensions

In this section we describe the syntax, semantics and implementation of the EOSCUBE monoids and monoid comprehensions. The formal counterpart of the EOSCUBE monoid comprehensions is monoid comprehensions of [FM95], which is a restricted version of monoid homomorphisms [BTBN91, BTS91, BTBW92] written using the syntax of monad comprehensions [Wad90], as is done by [BLS+94]. We first review the formal definition of monoids and monoid comprehensions borrowing heavily from [FM95] and [BW95].

6.1 Review of Monoid Comprehensions

\[
\text{BAG}\{\ c \mid fc \leftarrow \text{all\_countries}, \\
\quad c->\text{color} == \text{"red"} \}
\]

This is the original monoid comprehension syntax for the first EOSCUBE query in Subsection 5.2. Here, BAG indicates the type of the resulting collection (monoid); fc to the left of | is what we SELECT; \(\leftarrow\) is used to denote an iterator, i.e. the statement in the FROM clause; and the rest is predicates, i.e. logical conditions appearing anywhere in the WHERE clauses. The intuitive meaning is given by the nested loop program in Subsection 5.2.

In addition to collections, we can also compute aggregation functions. For example,

\[
\text{SUM}\{\ 1 \mid fc \leftarrow \text{all\_file\_cabinets}, \\
\quad fc->\text{color} == \text{"red"}, \\
\quad dr \leftarrow fc->\text{drawers}, \\
\quad dr->\text{color} == \text{"blue"} \}
\]

will count the number of file\_cabinets in the result.

More formally, a set of basic data types given, e.g., int, real and char, and a set of type constructors, e.g., set, list, bag. A data type is defined recursively as a basic data type or a constructed type \(T(\alpha)\) determined by the type parameter \(\alpha\).
A monoid is a triple \((T, \text{zero}, \text{merge})\), where \(T\) is a data type and \(\text{merge}\) is an associative function, of type \(T \times T \rightarrow T\), with left and right identity \(\text{zero}\). For example, \(\text{sum} = (\text{int}, 0, +)\) is a monoid. A collection monoid is a quadruple \((T(\alpha), \text{zero}, \text{unit}, \text{merge})\), where (1) \(T(\alpha)\) is a constructed type determined by the type parameter \(\alpha\), (2) \((T(\alpha), \text{zero}, \text{merge})\) is a monoid, and (3) \(\text{unit}\) is a function of type \(\alpha \rightarrow T(\alpha)\). As an example, \((\text{list(int)}, [], f, ++)\), where \(\[]\) is the empty list, \(f(i) = [i]\) for each \(i\) and \(++\) is the concatenation operation on lists.\(^5\) Finally, a primitive monoid is a quadruple \((T, \text{zero}, \text{unit}, \text{merge})\), where (1) \((T, \text{zero}, \text{merge})\) is a monoid and (2) \(\text{unit}\) is the identity function of type \(T\). Examples of primitive monoids include \(\text{prod} = (\text{int}, 1, \text{id}, \ast)\), where \(\text{id}(i) = i\) for each integer.

Intuitively, a monoid \(\mathcal{M} = (T, \text{zero}, \text{merge})\) is an abstract definition of a data type. Collection monoids capture the bulk types, and primitive monoids capture the basic types. Each instance of the collection type \(\mathcal{M} = (T(\alpha), \text{zero}, \text{unit}, \text{merge})\) is expressed as compositions of functions \(\text{zero}\), \(\text{unit}\) and \(\text{merge}\) on instances of type \(\alpha\). As an example, the monoid \((\text{list(int)}, [], f, ++)\) given earlier defines a data type of the integer lists. An instance of the type is intuitively a list of integers and the list is expressed as a composition of functions \([\[]\), \(u\) and \(++\) applying on integers. For example, the list \{1, 2, 3, 1\} can be expressed as \(++(u(1), ++(u(2), ++(u(3), ++(u(1), []))))\).

A monoid \((T, \text{zero}, \text{merge})\) is called commutative (idempotent, resp.) if function \(\text{merge}\) is commutative (idempotent, resp.). For example, the monoid \(\text{set}^\alpha = (\text{set}(\alpha), \{\}, f', \cup)\), where \(f'(i) = \{i\}\) for each instance \(i\) of type \(\alpha\), is a commutative and idempotent monoid, and \(\text{bag}^\beta = (\text{bag}(\beta), \{\}, f'', \cup)\), where \(f''(i) = \{i\}\) for each instance \(i\) of type \(\beta\) and \(\cup\) is the additive bag union, is a commutative monoid. Intuitively, less properties correspond to more structure. For example, the monoid \(\text{bag}\) has more structure than the monoid \(\text{set}\) because repetitions in a \(\text{bag}\) do matter (since it is not idempotent), whereas in a \(\text{set}\) do not (since it is idempotent). Similarly, the monoid \(\text{list}\), being not commutative, has more structure than the monoid \(\text{bag}\), which is commutative. For monoids \(\mathcal{M}\) and \(\mathcal{N}\), we say \(\mathcal{N} \preceq \mathcal{M}\) if that \(\mathcal{N}\) is commutative (idempotent, resp.) implies that \(\mathcal{M}\) is commutative (idempotent, resp.), i.e. \(\mathcal{N}\) has the same or less properties than \(\mathcal{M}\). This exactly corresponds to the intuitive notion that \(\mathcal{N}\) has more structure than \(\mathcal{M}\). It is easily seen that \(\text{bag}^\beta \preceq \text{set}^\alpha\). If \(\mathcal{N} \preceq \mathcal{M}\), then an instance of type \(\mathcal{N}\) can be “translated” deterministically, by using the merge function of the monoid \(\mathcal{M}\), into an instance of the type \(\mathcal{M}\), but not necessarily vice versa.

Queries on monoids are expressed as monoid comprehensions. A monoid comprehension over the monoid \(\mathcal{M}\) takes the form

\[
\mathcal{M}\{e \mid r_1, \ldots, r_n\}
\]

\(^5\)We use \([a_1, \ldots, a_n]\) to denote a list and \(\{a_1, \ldots, a_k\}\) to denote a bag.
where \( e \) is an expression called the head of the comprehension, and \( r_1, \ldots, r_n \) is a list of qualifiers, each of which is either

- a iterator of the form \( v \leftarrow e' \), where \( v \) is a variable, and \( e' \) is an expression that evaluates to an instance of a collection monoid of type which is \( \leq M \) or

- a selection-predicate, which is an expression that evaluates to true or false.

The expressions in turn can include monoid comprehensions. An important condition for the monoid comprehension is that for each \( 1 < i \leq n \), each free variables (i.e., free variables in the expressions and predicates) appearing in \( r_i, \ldots, r_n \) must appear as the variable of an iterator among \( r_1, \ldots, r_{i-1} \), and each free variables in the \( e \) must appear as the variable of a iterator among \( r_1, \ldots, r_n \).

It is assumed that each instance of a monoid appearing as argument in a monoid comprehension is represented as an expression involving merge, unit and zero functions. For example \( \text{BAG}\{2,1,3\} \) can be represented as

\[
\text{merge}( \text{merge}(\text{merge}(\text{merge}(\text{zero},\text{unit}(1)),\text{unit}(1)),\text{unit}(2)),\text{unit}(1)) \text{, unit}(3))
\]

or, since merge is associative and zero is a left (and right) identity, as

\[
\text{merge}( \text{merge}( \text{merge}( \text{merge}( \text{merge}( \text{merge}(\text{zero},\text{unit}(1)),\text{unit}(1)),\text{unit}(2)),\text{unit}(1)),\text{unit}(1)) \text{, unit}(3))
\]

We will assume that every monoid instance is (conceptually) represented this way, and, thus, the notation \( \mathcal{N}\{a_1,a_2,\ldots,a_n\} \) will denote the expression

\[
\text{merge}(\ldots \text{merge}(\text{merge}(\text{merge}(\text{zero},\text{unit}(a_1)),\text{unit}(a_2)),\ldots,\text{unit}(a_n))
\]

Furthermore, \( \mathcal{N}\{} \), and \( \mathcal{N}\{a_1,a_2,\ldots,a_n\} \) where \( n = 0 \) will both denote zero\(\mathcal{N}\), i.e. the empty monoid instance.

A monoid comprehension \( \mathcal{M} \) over a monoid \( M = (T, \text{zero}, \text{unit}, \text{merge}) \) (collection or primitive), defines an instance of type \( T \) \(^6\) by first initializing result with zero\(M\), and then invoking the procedure insert\(\mathcal{MC}\)(result, \( \mathcal{M} \)) defined recursively by the following reduction rules:

\(^6\)Our definition here is different from, but equivalent to the original one; ours is closer to the implementation.
(r1) \( \text{insert} \_\text{MC}(\text{result}, \mathcal{M}\{e | \}) \)
\( \rightarrow \text{result} := \text{merge}^{\mathcal{M}}(\text{result}, \text{unit}^{\mathcal{M}}(e)) \)

(r2) \( \text{insert} \_\text{MC}(\text{result}, \mathcal{M}\{e | \text{false}, \bar{r}\}) \)
\( \rightarrow \text{nil} \) (i.e. do nothing)

(r3) \( \text{insert} \_\text{MC}(\text{result}, \mathcal{M}\{e | \text{true}, \bar{r}\}) \)
\( \rightarrow \text{insert} \_\text{MC}(\text{result}, \mathcal{M}\{e | \bar{r}\}) \)

(r4) \( \text{insert} \_\text{MC}(\text{result}, \mathcal{M}\{e | x \leftarrow \mathcal{N}\{a_1, \ldots, a_n\}, \bar{r}\}) \)
\( \rightarrow \text{for } i = 1 \text{ to } n \text{ do insert} \_\text{MC}(\text{result}, \mathcal{M}\{e | \bar{r}\}[x/a_i]) \)

where \( \mathcal{N} \) is a collection monoid \((S, \text{zero}^{\mathcal{N}}, \text{unit}^{\mathcal{N}}, \text{merge}^{\mathcal{N}})\) with the condition \( \mathcal{N} \subseteq \mathcal{M} \). Note that \( \mathcal{M}\{e | \bar{r}\}[x/a_i] \) denotes the replacement of \( x \) with \( a_i \) in \( \mathcal{M}\{e | \bar{r}\} \). Note also that the last rule is deterministic as far as the resulting monoid instance is concerned.

6.2 Monoids in EOSCUBE

To understand the minimum requirements for primitive and collection monoids, consider the recursive rules defining the result of a monoid comprehension. For the monoid \( \mathcal{M} \) to appear in the result of the monoid comprehension, we only need to (1) use \( \text{zero}^{\mathcal{M}} \) and (2) know how to perform

\[ \text{result} = \text{merge}^{\mathcal{M}}(\text{result}, \text{unit}^{\mathcal{M}}(e)) \]

which is, in fact the \( \text{insert}^{\mathcal{M}}(\text{result}, e) \) operation (i.e. we define \( \text{insert}^{\mathcal{M}} \) this way). In order for the collection monoid \( \mathcal{N} \) to appear inside the comprehension, we only need to be able to iterate over \( \mathcal{N}\{a_1, \ldots, a_n\} \), i.e. to perform the for loop.

The representation (and implementation) of collection monoids in EOSCUBE is based on two C++ template classes, parameterized with the type A of collection elements: CollectionMonoid and Iterator:

```cpp
template < class A > class CollectionMonoid
{
friend class Iterator<A>;
public:
    CollectionMonoid(); // C++ constructor used as zero
    virtual void Insert(A& ) = 0;
    virtual Iterator<A>* CreateIterator() = 0;
private:
    // specific subclasses contain
```

\(^7\)For a primitive monoid the name insert is probably strange; we really mean by insert exactly \( \text{result} := \text{merge}(\text{result}, \text{unit}^{\mathcal{M}}(e)) \). For example, for primitive monoid sum (int, +, 0, identity), \( \text{insert}^{\mathcal{M}}(\text{result}, 5) \) is \( \text{result} := \text{result} + 5 \).
The class `CollectionMonoid` reflects the minimum requirements: it has zero, implemented as a class constructor, and `Insert` and `CreateIterator` member functions. An `Iterator` object, created by `CreateIterator`, has `First`, `More` and `Next` member functions which can be directly used in the C++ for-loop. Specific collection monoids implemented in EOSCUBE, depicted in Figure 4, are implemented each with two classes derived from the classes `CollectionMonoid` and `Iterator`, correspondingly. The collection monoids `list`, `set`, and `bag` are currently implemented using the ObjectStore collections.

As opposed to collection monoids, primitive ones only require zero and the `insert` member function, since they are not used in the query iterators.

```cpp
template< class T > class PrimitiveMonoid
{
    public:
        PrimitiveMonoid( );               // Note: creates zero of monoid
        virtual void Insert( T& ) = 0;
        operator T ( );
        static T zero;
    protected:
        T value;
};
```

Figure 2: Collection Monoids in EOSCUBE
The extensible family of primitive monoids and monoid templates in EOSCUBE, depicted in Figure 5, includes $\text{Prod}<T>$, $\text{Max}<T>$, $\text{Sum}<T>$, $\text{Some}$ and $\text{All}$. Note, that the type used in $\text{Some}$ and $\text{All}$ is bool. These monoids work as disjunction and conjunction of conditions, respectively.

### 6.3 Syntax and Semantics of EOSCUBE queries

The syntax of the EOSCUBE comprehensions has been explained by examples. More accurately, it is of the form:

```
SELECT C++expr
INTO [(monoid_type)] [result]
% [from-where-define-list]
[from-where-define-perform-list]
```

$C++\text{expr}$ in the SELECT clause is an arbitrary C++ expression that evaluates to the type of result's elements. Note that $C++\text{expr}$ may involve variables instantiated in the FROM clauses and may also contain nested monoid comprehensions. The first (optional) parameter in the INTO clause specifies the type of result. If this argument is omitted, the system assumes that result is defined elsewhere in the C++ program. When the monoid comprehension is nested, result argument may
be omitted. The `from-where-define-perform-list` is a sequence of the FROM, DEFINE, PERFORM and WHERE clauses (explained earlier by examples) in any order. Note that any number of iterators, separated by commas, may appear in each FROM clause; further, any number of predicates (conditions) may appear in each WHERE clause. Also important is that nesting is recursively allowed anywhere in the monoid comprehension, provided that nested monoid comprehensions return appropriate types. For instance, collection monoid comprehensions may stand anywhere a collection monoid can; or, monoid comprehensions returning TRUE or FALSE may stand in place of any predicate. This flexibility also enables the use of special algorithms (such as for the constraint and sort join, indexing or regrouping), provided that they produce the appropriate types (e.g. collection monoid) as their outputs.

The semantics of EOSCUBE monoid comprehension queries is defined by the corresponding formal monoid comprehension. Furthermore, the basic evaluation is by the nested loop algorithm with dynamic buffer management. Important, however, is that the nested monoid comprehension in the FROM clause does not create physical intermediate results, but rather supports the pipe-lining.

7 CST Objects and Constraint Calculus

7.1 Framework for Constraint Algebra and Calculus

EOSCUBE uses the multi-typed algebra framework of [Bro], which we review here. As seen in the examples, the notion of CST data relies on a simple and fundamental duality: a constraint (formula) \( \phi \) in free variables \( x_1, \ldots, x_n \) is interpreted as a set of tuples \( (a_1, \ldots, a_n) \) over the schema \( x_1, \ldots, x_n \) that satisfy \( \phi \); and, conversely, a finitely representable object in \( (x_1, \ldots, x_n) \) space can be viewed as a constraint. That is, the syntax is constraints, i.e. symbolic expressions; the semantics are the corresponding, possibly infinite, relations.

CST objects are represented by a sub-family of the first order logic, (i.e. with the logical connectors \( \land, \lor, \neg \) and \( \exists \)) and by a family of atomic constraints, such as linear arithmetic over reals, polynomial or dense order. CST objects are manipulated by means of a constraint algebra, whose operators are expressed using a sub-family of the first-order logic, renaming of variables, and atomic (e.g. arithmetic) constraints. For example, if \( P \) and \( Q \) are CST objects in \( x_1, \ldots, x_n \), their intersection can be represented by \( P \land Q \); union by \( P \lor Q \); the test of containment of \( P \) in \( Q \) by \( \forall x_1, \ldots, \forall x_n (P \rightarrow Q) \) (this is, in fact, the entailment test, ENTAIL); emptiness of \( P \) by \( \neg \exists x_1, \ldots, \exists x_n P \) (this is, in fact the satisfiability test, SAT); disjointness of \( P \) and \( Q \) by \( \neg \exists x_1, \ldots, \exists x_n (P \land Q) \); the projection of \( P \) on axes \( x_1, \ldots, x_i, 1 \leq i < n \), by \( \exists x_{i+1} \ldots \exists x_n P \) etc. If we only use linear constraints over reals, as implemented in
EOSCUBE, within the first-order logic we can express any linear transformation such as rotation, translation and stretch; check convexity, discreteness and boundness [?]; compute the convex hull, augment objects, change coordinate systems; etc.

Thus, constraint objects can be manipulated by a very expressible language. Moreover, since this language uses only a small number of operators (i.e. logical connectors and quantifiers), it is also very compact, as compared to using a separate operator for each specific type of transformation, which is typically done in extensible or spatial database systems. It is also claimed, that for linear constraints, query languages manipulating constraint objects are deeply optimizable, in terms of indexing and filtering (e.g. [BLLM95, KRV93, Sri92]), and constraint algebra algorithms and global optimization (e.g. [BJ93, GK95]).

More specifically, constraint algebras operate on a family $\mathcal{F}$ of canonical representations of constraint expressions (objects). For constraint objects $C_1, \ldots , C_n$ a first-order logic formula $\phi(C_1, \ldots , C_n)$ such as $\exists y(C_1[u_1/y, v_1/z] \land \ldots \land C_n[u_n/y, v_n/z])$, where $[u_i/y, v_i/z]$ denotes the variable replacement, defines the following constraint algebra operator $\mathcal{op}$: (1) replace each $C_i$ by the corresponding constraint expression, (2) do all variable replacements and (3) transform the resulting constraint expression into the required (equivalent) canonical representation in $\mathcal{F}$. Thus $\mathcal{op}$ can be seen as a function from $\mathcal{F} \times \ldots \times \mathcal{F}$ to $\mathcal{F}$. On the other hand, the operator $\mathcal{op}$ has the interpretation $\mathcal{I}(\mathcal{op})$, which is a query that maps $n$ relations to one. Given $\mathcal{I}(C_1), \ldots , \mathcal{I}(C_n)$, where $\mathcal{I}(C_i)$, $1 \leq i \leq n$, is the relational interpretation of $C_i$ and $x_1, \ldots , x_m$ are all free variables, $\mathcal{I}(\mathcal{op})$ computes the following relation:

$$\{(x_1, \ldots , x_m) \mid \phi(C_1, \ldots , C_n)\}$$

Clearly, the duality between constraints and point sets carries over to the constraint algebra/calculus, that is, the following commutative property holds:

$$\mathcal{I}(\mathcal{op}(C_1, \ldots , C_n)) = \mathcal{I}(\mathcal{op})(\mathcal{I}(C_1), \ldots , \mathcal{I}(C_n))$$

A constraint family $\mathcal{F}$ is defined by choosing (1) an atomic constraint domain, (e.g. polynomial over reals or linear over integers), (2) the structure of the logical formula allowed (e.g. disjunction of conjunctions or existentially quantified disjunction) and (3) the required canonical form (e.g. whether to eliminate existential quantifiers, eliminate each redundant disjunct, extract all implicit equalities in conjunction or eliminate redundancy in conjunctions). The definition of a constraint algebra amounts to choosing the structure of first-order formulae and the atomic constraints allowed in the query.

The challenge here (and a major area of research) is the development of constraint families and algebras, that strike, for each application realm, a careful balance between (1) expressiveness, (2) computational complexity and, very importantly, (3) representation usefulness.
As one extreme, if the entire first-order logic (as studied in [?, ?]), and the same atomic constraints are allowed in both the constraint family $\mathcal{F}$ and the algebra, we get a very expressive algebra with the low data complexity, since no actual manipulation of constraints would be required. However, the representation of the result might consist of a very large unsimplified constraint expression that might not be useful for the user. For instance, the answer to a query "is constraint object $C$ empty" would be $\exists x_1 \ldots \exists x_n C$, where $x_1, \ldots, x_n$ are all free variables, whereas the user expects a true or false answer.

An example of a very expressive, but having high (exponential) time data complexity is the DISCO (Datalog with Integer and Set order Constrains) query language [BR95]. Constraint representation is DISCO is useful in many, but not all applications. For example, to express satisfiability of a simple propositional formula, the user needs to encode the formula by a datalog (with constraints) program, in a fairly unnatural way.

Close to the other end, the framework [?] requires a fairly restricted sub-family of first-order logic in constraint objects: disjunction of (unquantified) conjunctions of atomic constraints (the algebra, however, allows more, including quantifier elimination). This representation is useful for many, but not all applications: for example a constraint representation of a triangle given by vertices $(a_1, b_1), (a_2, b_2), (a_3, b_3)$,

$$\exists t_1 \exists t_2 \exists t_3 (x = a_1 t_1 + a_2 t_2 + a_3 t_3 \land y = b_1 t_1 + b_2 t_2 + b_3 t_3 \land t_1, t_2, t_3 \geq 0 \land t_1 + t_2 + t_3 \leq 1)$$

is not directly representable in that framework. Still, for some atomic constraint families, such as linear inequalities over reals, this framework may be computationally unmanageable: the quantifier elimination may result in a constraint exponential in the size of the original conjunction, although for many sub-families more efficient algorithms were developed (e.g. [GK95, JMSY92, HLL90, LL91]). A more flexible first-order logic structure that allows the entire linear constraints over reals while controlling computational complexity was described in [BJM93, BK95].

### 7.2 EOSCUBE Constraint Families and Canonical Forms

In EOSCUBE we concentrate on linear constraint over reals, which are expressive and useful in a variety of application domains. However, in order to control the computational complexity, we design a more flexible first-order logic structure by constructing a number of interrelated constraint families. This continues the line of work in [BJM93, BK95].

The six interrelated constraint families in EOSCUBE are depicted in Figure 6. The four main families are for unrestricted linear constraints over reals: C \_LIN, for
Conjunctive Linear, stands for constraints represented in the form $\land_{i=1}^{n} C_i$, where $C_i$ is a linear inequality; **EC.LIN**, for Existential Conjunctive, corresponds to the form $\exists x \land_{i=1}^{n} C_i$; **DC.LIN**, for Disjunctions of Conjunctions, corresponds to the form $\lor_{i=1}^{m} \land_{j=1}^{n} C_{ij}$; and **DECLIN**, for Disjunctions of Existential Conjunctive, corresponds to the form $\exists x \lor_{i=1}^{m} \land_{j=1}^{n} C_{ij}$. The other two families are for range constraints, i.e. of the form $a \op x \op b$, where $\op$ is either $<$ or $\leq$ and $a$ and $b$ are either real numbers or $-\infty$ or $\infty$. Namely, **C.RANGE**, for Conjunctive Range, stands for constraints represented in the form $\land_{i=1}^{n} C_i$, where $C_i$ is a range constraint; and **DC.RANGE** corresponds to the form $\lor_{i=1}^{m} \land_{j=1}^{n} C_{ij}$.

We use the EOSCUBE notation for operations: **not**, **&**, **I**, and (**...**) for projection. We distinguish between projections on one variable, denoted (**one**); on zero attributes, denoted (**0**), i.e. all free variables are existentially quantified; on all variables, denoted (**all+**), i.e. no variables are quantified; and, on any number of attributes, denoted (**any**), for arbitrary projection. The user is recommended to use the most specific projection operator in order to achieve the strongest (i.e. lowest) resulting types.

---

Figure 4: Families of CST Objects

---

63
Not only the projections in EOSCUBE can eliminate existing free variables, but they can also add new ones. For example, a CST \((1 \leq x \leq 5)\) can be transformed by the "projection" on \((x, y)\) into \((x, y) \mid (1 \leq x \leq 5)\), thus adding the new free variable \(y\), and getting a new interpretation as a relation over \(x, y\) of all tuples with \(x\) as required and an arbitrary real number \(y\). However, in the classification of the projection cases discussed earlier, we only consider free variables physically appearing in the constraint expressions.

Thick arrows indicate type hierarchy. For example, C.LIN is a sub-type of DCLIN, EC.LIN and, transitively, DEC.LIN, meaning that a CST object of type C.LIN may be used as an argument wherever its supertypes are allowed.

Thin arrows indicate, for each constraint family, the allowed operations and the type of the result, which may belong to a different constraint family. For example, && is allowed on C.LIN, returning the result in the same family, while () and (one) return the result in C_RANGE (which is also in C.LIN as a supertype of C_RANGE). Note, that the result of || on arguments from C.LIN will be in DCLIN, not in C.LIN. Some of the operations are implicit: for instance, while || does not explicitly appear in C.LIN, it can be applied since it is allowed for its supertype DCLIN.

Operators may be overloaded: for example && in C.LIN is different from && in DC.LIN; they are implemented differently and return the results of different types (with different representations). The actual operator applied depends on the types of its arguments. As an example, &&(C.LIN,C.LIN) will use the C.LIN operator; whereas, &&(DC.LIN,DC.LIN), as well as &&(C.LIN,DC.LIN) will use the operator from DC.LIN. In general, for the application of op(arg1,arg2) we use the lattice structure of the type hierarchy, where sub-type \(\preceq\) super-type (i.e. the higher the bigger). The actual op chosen is the one of the CST type that is the least upper bound of type(arg1) and type(arg2) on which op is defined. Note, that the CST families are constructed in such a way that, for every op used, such least upper bound, if exists, is unique; hence, there is no ambiguity. If no such bound exists, op is not allowed on arg1 and arg2 and would result in a compile-time error.

For users of the EOSCUBE CST library it is easy to remember what's allowed and what's not. &&, ||, and () can be freely applied on arguments of any CST family. Only not is restricted: it can only be applied to arguments of the type C.LIN (and thus its sub-type C_RANGE). The system will always produce the strongest (i.e. least) type possible for the resulting constraint.

In addition to the logical algebraic operators, all families have the following operators:

1. RENAME(CST_obj,"'x1/e1,...,xN/eN'") where \(x1,...,xN\) are the names of the variables to be replaced with the variables \(e1,...,eN\).

2. SAT(CST_obj) to check satisfiability of the argument, i.e. whether there exists
an assignment of real constants into its free variables that makes it true. There is also MUT.SAT(CST.obj_1, CST.obj_2) which is equivalent to SAT(CST.obj_1 && CST.obj_2).

3. TRUTH.VALUE(Var.Assign, CST.obj) returns the truth value of the CST under the assignment Var.Assign of constants into the CST.obj free variables.

4. MIN.POINT(lin.func, CST.obj) and MAX.POINT(lin.func, CST.obj) where lin.func is a linear function with real coefficients. Returned is the assignment of constants into the variables of lin.func that maximizes it subject to the constraints in the CST.

5. MIN(var.name, CST.obj) and MAX(var.name, CST.obj) that return MIN and MAX of the first argument subject to the constraints in the second.

Note that the MIN and MAX operators correspond to the problem of linear programming. In addition, ENTAIL(DCLIN, CLIN) operator is allowed.

Finally, since all the disjunctive CST objects can be considered as collections of disjuncts and all the conjunctive CST objects as collection of conjuncts, we make these CST families EOSCUBE collection monoids by implementing the required iterators and member functions.

The six CST families are carefully constructed with the complexity consideration in mind as follows. First, all operations allowed on the families have polynomial data complexity. This is the reason, for example, that CLIN is not closed under the general projection: transforming the result into CLIN will require quantifier elimination and thus the size of the result (and, of course, time complexity) may be exponential in the number of variables eliminated. Whereas, ECLIN is closed under the general projection since the general projection in ECLIN is lazy: ECLIN allows quantifiers in the internal representation and hence no physical quantifier elimination is performed. Similar, not is allowed on conjunctive CST families, C.RANGE and CLIN, but not on, say, DCLIN. The reason is that transforming an expression of the form ¬V ∧C or, of the form ∧V ¬C, into DCLIN may result in an expression of an exponential size, which we would like to avoid. We discuss what operators involve computationally in more detail in the next subsection.

The CST families use canonical forms, i.e. useful standard forms of constraints, that we adopt in EOSCUBE from [LHM89, BJM93] and review here from [BJM93]. For CST objects in the disjunctive families, some disjuncts might be redundant in the sense that omitting them results in an equivalent constraint. Clearly, a canonical form that eliminates such disjuncts would be desirable. However, the problem of detecting
such tuples is co-NP-complete [Sri92], and so we will perform only one simplification of disjunctions: the deletion of inconsistent disjuncts.

For CST objects in the conjunctive families, there are a number of simplifications that can be requested by the user. One choice is to write all the equations in the form \( \{x_i = t_i \mid i = 1, \ldots, n\} \) where \( x_i \)'s are distinct and appear nowhere else in the constraint. A second choice is whether all equations which are implicit in the inequality constraints should be represented explicitly. (As a simple example of this, consider the constraints \( x + y \leq 2, x + y \geq 2 \).) A third is the extent to which redundancy within the inequalities should be removed. [LHM89] presents a classification of redundancy that suggests simple forms of redundancy removal. A fourth choice is whether to keep the inequalities in a different form, such as the simplex tableau form.

In the current EOSCUBE implementation, the only simplification is the removal of inconsistent disjuncts in the disjunctive families; however, a range of simplifications on the conjunctions is presently being implemented.

### 7.3 Implementation of CST families

On the conjunctive families, the \&\& operator simply combines its arguments and is constant time; (one) \( \mid \ C \), which is a projection on a single variable, involves applying a linear program (using the simplex algorithm of CPLEX) twice for finding the minimum and the maximum of the variable subject to \( C \); (\( \emptyset \mid C \)), which is eliminating all variables in \( C \) works as a satisfiability test, using the first phase of the simplex, as does the SAT predicate.

On the disjunctive families, the || operator is constant-time, while \( D_1 \&\& D_2 \), where \( D_1 = \bigvee_{i=1}^n C_1_i \) and \( D_2 = \bigvee_{j=1}^m C_2_j \) is more involved:

\[
D_1 \&\& D_2 = \bigvee_{i=1}^n C_1_i \&\& \bigvee_{j=1}^m C_2_j = \bigvee_{i=1}^n \bigvee_{j=1}^m (C_1_i \& C_2_j)
\]

that is, the result consists of all combinations of \( C_1_i \) and \( C_2_j \) that are mutually consistent (i.e. their conjunction is satisfiable). Since the CST families are monoids, \( D_1 \&\& D_2 \) is implemented as the following EOSCUBE query:

```
SELECT *c1 &*c2
INTO {DEC_LIN} conj_D1_and_D2
FROM D1 AS {EC_LIN*} c1,
     D2 AS {EC_LIN*} c2
WHERE SAT(*c1,*c2)
```

We show how such queries are optimized using the approximation-based filtering, indexing and re-groupings in the next section. Similar, \( SAT(D) \), where \( D \) is of the type \( DEC_LIN \) (as well as other disjunctive families) is represented as
SELECT SAT(*c)
INTO {Some} satisf_flag
FROM D AS {EC_LIN*} c

Note, that c is of the type EC.LIN and so SAT in the WHERE clause works on conjunctions. Further recall that Some is a primitive monoid whose merge operator is a logical or; thus, the satisf_flag will be true if and only if at least one component is true. Finally, ENTAIL(D, C), where D is DC.LIN and C is C.LIN, is represented as

SELECT ENTAIL(*c,C)
INTO {Some} imply_flag
FROM D AS {C.LIN*} c

Beyond the algorithms for constraint operations discussed, there are two subtle design problems that we address in EOSCUBE: compile-time maintenance of the type lattice and a lazy evaluation. Support for the lazy evaluation of constraint (i.e. involving CST) expressions is necessary for efficiency. For example, if we are interested in the SAT test of an expression involving logical connectors, it is typically wasteful to perform simplifications of subexpressions.

To exemplify the problem arising from the type lattice maintenance, consider the && operator. In fact, while && has one conceptual meaning, it works differently in every CST family. Moreover, since && is defined on DEC.LIN, the arguments may be any subtypes of DEC.LIN. Thus, every ordered pair of (sub) types for arguments of && works uniquely: we need to find the least upper bound type, to perform corresponding type conversions, and then to apply the physical algorithm of the resulting CST family. One possibility is implementing a separate function for each pair of subtypes, but this would result in a quadratic number of functions for each logical operator: 30 for the six families, and impractically many for future extensions of EOSCUBE with new CST families. On the other hand, the direct implementation of a subtype relationship using the C++ inheritance mechanism does not work, since each family has its own implementation, data-structures etc, which should not be inherited by its subclasses. Of course, there is also a possibility of maintaining just one global CST type, and to distinguish individual subfamilies only at run time. This, however, would eliminate the capability of the compile-time type checking, an important feature of EOSCUBE.

To solve the type lattice problem we designed a two-layer architecture for the CST families: the lower layer, called basic.CST, supports the physical representation and manipulation of the CST families; the upper layer, called lazy_typed.CST, is responsible for the type lattice management and the lazy evaluation, while the actual evaluation is passed to the lower, basic.CST layer.

The basic.CST layer is composed of the six classes basic.C.LIN, basic.DC.LIN etc., each maintaining its own data structures to represent the underlying constraints;
and one super (base) class, basic_CST. No automatic type casts are supported on this layer. However, each basic family has member functions for explicit type conversions into basic types that are higher in the type hierarchy. For example, transforming basic_CLIN into basic_DC_LIN creates a basic_DC_LIN object (disjunction) that has a single disjunct in it.

The lazy_typed_CST layer, on the other hand, does support automatic sub-typing and the ability to determine the least upper bounds of operators' arguments at compile-time. The six families lazy_typed_CLIN, lazy_typed_DC_LIN etc., are implemented as six classes with a class hierarchy that exactly matches the type hierarchy of the CST families. However, all the lazy_typed classes have similar internal representation, which is inherited from the abstract class lazy_typed_CST. The representation is basically an expression tree (hence “lazy”), with internal nodes storing the constraint operators (such as && or ||) and encoding the strongest type to which the subtree can be converted; the leaves are objects of the lower layer, basic_CST. It is important to emphasize that the CST type checking we do in EOSCUBE heavily uses capabilities of C++ and would be impossible (at compile-time without any precompiling) in languages such as C.

The EOSCUBE system also supports two generic parameterized CST families: Gen_Conj<T> for generic conjunctions and Gen_Disj<T> for generic disjunctions, where T is an arbitrary, possibly complex, CST type. Both are collection monoids and support the TRUTH_VALUE function; further, SAT is supported by Gen_Disj<T> provided it is supported by T. Also ENTAIL(Gen_Disj<T>,T) is defined provided it is defined on T. These operations are represented again with monoid comprehension queries. For example, SAT(D), where D is of the type Gen_Disj<T>, is represented as

SELECT SAT(c)
INTO {Some} satisf_flag
FROM D AS {T} c

Finally, we explain how the native C++ syntax is preserved in constraint formulas, such as in \( 2 \leq z \leq 5 && x + z \leq 7 \). The logical connective && is supported by the C_LIN class. In turn, each of the C++ expressions \( 2 \leq z \leq 5 \) and \( x + z \leq 7 \) must yield an object of type C_LIN. This is done by overloading operators \( \leq \) and \(+\). Clearly, in such an expression, \( x \) and \( z \) must be C++ variables that have already been declared within an appropriate C++ scope (the usual C++ scoping rules apply). The type for the C++ variables \( x \) and \( z \) is a special class, called CST_Var, which keeps inside the a constraint variable name, i.e. a string. It is convenient, although not required, to use the same name for a constraint variable name and the corresponding C++ variable into which the constraint variable is assigned. Each CST object keeps inside its free constraint variables that can be also shared among different CST objects.
8 Optimization by Approximation-based Filtering and Indexing

General optimization of object-oriented queries (e.g. ENCORE [Zdo89], O2 [ea90], POSTGRESS [SRH90]) and monoid comprehensions in particular, (e.g. [FM95]), as well as optimization in presence of expensive predicates [CS93, HS93] is outside the scope of this paper; We concentrate here on approximation-based filtering, regrouping and indexing [BW95], that EOSCUBE is designed to support. More specifically, we describe, mostly by examples, the EOSCUBE primitives for approximation and inverse groupings [BW95] and indices and special purpose algorithms.

To understand the idea, we use a modification of an example from [BW95] of the query: “find all trajectories passing over the Fairfax county”. It will be assumed here that a set of 4D aircraft trajectories as well as a map is stored in the database. A trajectory is assumed to have a piece-wise linear representation, i.e. it is represented as a DC_LIN CST object

\[
\bigvee_{i=1}^{n} \left( t_{i-1} \leq t < t_{i} \land x = a_{i,1}t + b_{i,1} \land y = a_{i,2}t + b_{i,2} \land z = a_{i,3}t + b_{i,3} \right)
\]

Where \(x, y, z\) are variables for a location, \(t\) is a time variable, and \(t_{i-1}, a_{i,1}, b_{i,1}, a_{i,2}, b_{i,2}, a_{i,3}, b_{i,3}, 1 \leq i \leq n,\) are constants. Note that for each \(i\), \((t_{i-1} \leq t < t_{i} \land x = a_{i,1}t + b_{i,1} \land y = a_{i,2}t + b_{i,2} \land z = a_{i,3}t + b_{i,3})\) describe the movement equations for the time interval \([t_{i-1}, t_{i})\), for the constant 3-D velocity vector \((a_{i,1}, a_{i,2}, a_{i,3})\), starting from the point \((b_{i,1}, b_{i,2}, b_{i,3})\). All_trajectories is a variable of type SET<DC_LIN*>, i.e. a set of pointers to trajectories. The Fairfax county is assumed to be represented as a polygon,\(^9\) i.e. as a C_LIN CST object Fairfax_area in variables \(x\) and \(y\). The query can be directly expressed in EOSCUBE as

```sql
SELECT traj
INTO {Set<DC_LIN*>>} result
FROM All_Trajectories AS {DC_LIN*} traj
WHERE MUT_SAT(*traj, Fairfax_area ) // Note: MUT_SAT
    // on DC_LIN
```

or, if MUT_SAT is expressed, in turn, as a monoid comprehension:

```sql
SELECT traj
INTO {Set<DEC_LIN*>>} result
FROM All_Trajectories AS {DC_LIN*} traj
```

\(^9\) in fact, it is not convex, but we’ll assume that to simplify the example
WHERE SELECT MUT_SAT(*segment,Fairfax_area)  // Note: MUT_SAT
    INTO {Some}
    // on C_LIN
    FROM *traj AS {C_LIN*} segment  // SELECT returns
    // True of False

which is an expensive query if evaluated directly.

**Inverse Grouping (IG)**

To optimize the last query, we can first use the inverse grouping, described graphically in Figure 7. Intuitively, each trajectory can be viewed as composed of 4D-segments, and each segment has a projection, say an $xy$-segment on the horizontal plane $x,y$.

Now, consider a collection of all $xy$-segments that originate from all the trajectories. The inverse grouping primitive tracks back, for each $xy$-segment in the collection, the set of all the trajectories from which this $xy$-segment came.

The correspondence between an $xy$-segment and the set of trajectories is captured through a parameterized C++ class $G$-Pair<$T_1,T_2>$ which represents a pair containing a reference to an object of type $T_1$ (C_LIN* in our case) as its first element and a reference to a set of objects of type $T_2$ (DC_LIN* in our case) as its second element. The inverse grouping is captured through a special C++ class which essentially represents a set of G_Pair-s. To create an inverse grouping the user must create an instance of that class, passing the original collection as a constructor pa-
rameter. Assuming that All_Traj_IG has been created as the inverse grouping for the All_Traj collection, we can now re-write the previous query as follows:

```sql
SELECT traj
INTO {Set<DC_LIN*>>} result
FROM All_Traj_IG AS {G_Pair<C_LIN*,DC_LIN***>} G_pair
//
DEFINE xy_segment AS {C_LIN*} G_pair->first
WHERE MUT_SAT( *xy_segment, Fairfax_area ) // Note: MUT_SAT
// on C_LIN

DEFINE trajectories_of_xy_segment
AS {Set<DC_LIN*>>} G_pair->second
FROM *trajectories_of_xy_segment AS {DC_LIN*} traj
```

First note that the new query is equivalent to the previous one. The reason is that a trajectory passes over the Fairfax_area if and only if one of its xy-segment intersects the Fairfax_area. The new query, however, is likely to perform better. Note that different trajectories may “share” the same xy-segment, for example if one passes exactly above another. Therefore, the previous query will apply the MUT_SAT (i.e. intersection) test multiple times for every shared xy-segment. Whereas, in the last query we eliminate such duplication by checking MUT_SAT for each xy-segment only once and then quickly retrieving the corresponding trajectories using the inverse grouping.

**Approximation Grouping (AG)**

Even though we have minimized the number of the MUT_SAT checks, we know that MUT_SAT on C_LIN is relatively expensive. To optimize further, we can approximate each xy-segment with a minimum bounded box (MBOX), which is of type C_RANGE. Then, before testing MUT_SAT on C_LIN it can be tested first on MBOXes, which is cheaper. Also, it opens opportunities for indexing. This is the purpose of the approximation grouping primitive, which we introduce next. Intuitively, approximation grouping takes a collection of objects and creates another collection. The elements of the latter are approximations of the objects from the original one (see dashed lines in Figure 7). The approximation grouping, similar to the inverse one, must track back, for each approximation, the set of original objects that yield this approximation.

The implementation of the approximation grouping is similar to that of inverse grouping. It uses the same class G_Pair<T1,T2> which now represents the relationship between an approximation and the corresponding objects. The approximation grouping is constructed by creating an object of a special class, passing the original
collection and the approximation method as constructor parameters. As in the case of inverse grouping, the object being created represents a set of G_Pair-s. To continue the optimization of the last query the original collection in our case would be the inverse grouping collection All_Traj_IG. Recall that All_Traj_IG is a set of inverse grouping pairs G_Pair<C_LIN*,DC_LIN*>. If we define an approximation of an inverse grouping pair as an approximation of its first element (i.e. xy-segment) then we can create an approximation grouping object All_Traj_IG_AG which would be a set of G_Pair<C_RANGE*,G_Pair<C_LIN*,DC_LIN>>-s.

We can now re-write the last query as follows:

```sql
SELECT traj
INTO {Set<DC_LIN>>} result
//
FROM All_Traj_IG_AG AS {G_Pair<C_RANGE*,G_Pair<C_LIN*,
DC_LIN>>}> AG_pair
DEFINE min_box_of_xy_segment AS {C_RANGE*} AG_pair->first
WHERE MUT_SAT( *min_box_of_xy_segment, // On C_RANGE;
min_box_of_Fairfax) // precomputed outside // of the query
DEFINE Candidate_Traj_IG AS {Set<G_Pair<C_LIN*,DC_LIN>>>}
AG_pair->second
//
FROM *Candidate_Traj_IG AS {G_Pair<C_LIN*,DC_LIN>>} IG_pair
DEFINE xy_segment AS {C_LIN*} IG_pair->first
WHERE Mut_Sat( *xy_segment, Fairfax_area ) // Note: MUT_SAT
// on C_LIN
DEFINE trajectories_of_xy_segment
AS {Set<DC_LIN>>} IG_pair->second
FROM *trajectories_of_xy_segment AS traj
```

Indexed Approximation Grouping (IAG)

The last query involves the retrieval of rectangles that are mutually consistent with a given one. Another optimization possibility is to maintain an index on the collection of rectangles. This is done using the indexed approximation grouping, instead of the approximation grouping. The IAG has the same functionality as the approximation grouping with the following differences: 1) the first element of an IAG pair is always a rectangle (i.e. of type C_RANGE) and 2) there is a kD-tree index imposed on the rectangles. The IAG is constructed by creating an object of a special class which, in addition to the approximation grouping functionality, contains member functions for
search. In our example we create an indexed approximation grouping object named All_Traj.IG.IAG. As in the case of approximation grouping, All_Traj.IG.IAG is a set of G_Pair<C_RANGE*,G_Pair<C_LIN*,DC_LIN>>-s. In the query we invoke the MUT_SAT(C_RANGE) method on that object. The method returns, for each MBOX (of type C_RANGE), the set of all G_Pair<C_LIN*,DC_LIN>*-s for which the corresponding rectangles intersect that MBOX.

Note, that the returned set is not a physical set collection, but rather a structure allowing to iterate over its elements (and thus no intermediate evaluation is necessary when used within monoid comprehension). The last query can be re-written as follows:

```
SELECT traj
INTO {Set<DC_LIN*>} result
//
FROM All_Traj.IG.IAG.MUT_SAT(min_box_of_Fairfax)
    AS {Set<G_Pair<C_LIN*,DC_LIN>>} Candidate_Traj.IG
//
FROM *Candidate_Traj.IG AS {G_Pair<C_LIN*,DC_LIN>>} IG_pair
DEFINE xy_segment AS {C_LIN*} IG_pair->first
WHERE Mut_Sat( *xy_segment, Fairfax )       // Note: MUT_SAT
    // on C_LIN
//
DEFINE trajectories_of_xy_segment AS {Set<DC_LIN>>} IG_pair->second
FROM *trajectories_of_xy_segment AS traj
```

It is important to note, that, while we intuitively explained the use of the IG, AG and IAG by examples, these primitives can be applied to any CollectionMonoid<A>. For the IG the user needs to provide a transformation producing, for each element of type A an instance of (another) commutative and idempotent monoid (see [BW95] for details). For the AG and IAG, an approximation of elements of type A must be provided by the user. The IG, AG and IAG are used to facilitate the query transformation rules supporting approximation-based filtering (by using less expensive predicates first) and indexing (see [BW95]).

9 Related Work

No technology for declarative and efficient querying of databases involving constraint objects exists today. Applications of the kind discussed are typically implemented by special purpose programs; while these programs may use database and constraint programming tools, they typically require a considerable programming effort and
are not flexible to changes. In addition, they do not perform overall optimization that interleaves database, mathematical programming and computational geometry manipulation techniques. Existing DBMS do not manage constraints as persistently stored data. Constraint Logic Programming, on the other hand, was not designed to deal with large amounts of persistent data. Extensions of DBMS with spatio-temporal operators typically (1) are limited to low (two- or, at most three-) dimensional space, (2) have query languages restricted to predefined spatio-temporal operators, and (3) lack global economical filtering and deep optimization.

There has been work on the use of constraints in databases, earlier of which include. The pioneering work proposed a framework for integrating abstract constraints into database query languages by providing a number of design principles, and studied, mostly in terms of expressiveness and complexity, a number of specific instances. The work considered polynomial equality constraints, adopting local propagation steps for reasoning on constraints. A restricted form of linear constraints, called linear repeating points, was used to model infinite sequences of time points. More recent works on deductive databases considered the manipulation and repositioning of constraints for optimizing recursion. Algorithms for constraint algebra operators such as constraint joins, and generic global optimization were studied in, and constraint approximation-based optimization in. The work proposed an efficient data structure for secondary storage suitable for indexing constraints, that achieves not only the optimal space and time complexity as priority search trees, but also full clustering. The work proposed an approach to achieve the optimal quality of constraint and spatial filtering. A number of works consider special constraint domains: integer order constraints; set constraints; dense-order constraints. Linear constraints over reals drew special attention. The use of constraints in spatial database queries was addressed in. The work used constraints to describe incomplete information. Constraint aggregation was studied in.

DISCO (Datalog with Integer and Set order Constraints) is a constraint database system being developed at the university of Nebraska. DISCO incorporates a highly expressive family of constraints. However, its query language has time complexity exponential in the size of a database; hence DISCO’s applicability to real-size database problems is not clear. Further, DISCO does not support the standard database features such as persistent storage, transaction management and data integrity.

Note, integrity constraints used in conventional databases are not data, but rather something the data must satisfy.
10 Lessons Learned

We first summarize some lessons we learned while building the EOSCUBE system.

The two-layer implementation of the constraint algebra was a simple and ex-
tensible solution to the problem of finding the appropriate C++ structures. The
experiences learned while implementing the algebra can be applied to the C++ im-
plementation of other similar multi-typed algebras.

Using ObjectStore and CPLEX as underlying components allowed us to concen-
trate on issues related to constraint implementation and to ignore many common
aspects such as the simplex algorithm, persistence, data integrity etc.

However, in the process of development many problems came up while coding
and maintaining some ObjectStore-specific parts. The goal was to put the library
on a higher level, making it more flexible and uniform to use. Also, we did not
want to tie the library design to the specific components used underneath. Rather,
we aimed to make it portable enough to be easily transferred on the top of another
object manager and/or LP package. This has been achieved by implementing an
intermediate interface between EOSCUBE and the underlying components, which
isolated the developers and users of the system from ObjectStore technicalities. In
order to use a different object manager/LP package, only a relatively small part of
the code will have to be re-written. A lot of effort has gone in making this interface
both portable and easy to use.

Still, a fair amount of time has been spent on the implementation of the supporting
structures such as search trees and sparse matrices. Even though there are some C++
packages that handle these issues, none of them was flexible enough to be directly
plugged in EOSCUBE.

Currently EOSCUBE does not parse the full C++ grammar. Instead, there is a
pre-compiler that passes its output over to the C++ compiler. EOSCUBE queries
are embedded in hosting C++ programs and are allowed to use variables declared in
the appropriate scope (with some restrictions). During the implementation of this
scheme we encountered some non-trivial technical problems that required much effort
to resolve.

It is important to note that C++ was the best pragmatic choice for our purposes.
Its compactness and expressiveness power enabled us to make things that would be
hardly possible in any other programming language. The library extensively uses tem-
plates, operator overloading and multiple inheritance. Those features also provided
higher reusability of components which allowed us to reduce the total amount of the
source code (around 11,000 lines now, not including the commercial components).

Even though the query language we presented is an optimization-level language
in which evaluation plans can be explicitly expressed, we found that the optimiza-
tion primitives are not currently automated to the extent they can possibly be. For
example, similar to a regular index, the grouping primitives require dynamic main-
tenance. If there is an update in the original collection, the corresponding change
must be made in the grouping structures. EOSCUBE does not support the dynamic
updates in the current implementation. The work in this direction is currently being
performed.

We have described the work on the development of the first constraint object-
oriented database system. Our work aims at the developing a practical and useful
technology for a wide variety of important application realms, for which no existing
technology is applicable. For example, EOSCUBE can be directly used to implement
the real-life data fusion and sensor management system for air-space command and
control [ABK95]. EOSCUBE is a deeply optimizable and extensible system, striking
the balance between expressiveness and computational complexity.

Many research questions remain open (see [Bro] for an overview): in constraint
modeling and canonical forms, data models and query languages, indexing and approximation-
based filtering, and, most importantly, special constraint algebra algorithms for spe-
cific domains and global optimization.
Part V
Global Optimization using Workflows: Work in Progress

11 Workflow Systems

A workflow is a collection of cooperating, coordinated activities designed to carry out a well-defined complex process. In the context of EOSCUBE, the activities considered are computation of interrelated EOSDIS and other scientific products.

A number of workflow representation frameworks has been proposed, the most common being control flow graphs, triggers (also known as event-condition-action rules) and temporal constraints.

![Control Flow Graphs](image)

Figure 6: Control Flow Graphs

To exemplify a control-flow graph, consider an abstract example depicted in the graph in Figure 1. The example depicts achieving a global task (e.g., generating a
set EOSDIS products for the satellite data of 24 hours) using activities E1 through E12 (e.g., involving generating various levels of EOSDIS products, and possibly some auxiliary intermediate results) going through the starting state A to the final state K.

The graph indicates that activity E1 must precede all other activities, leading to reaching states B and C, both of which must be pursued to reach the final state K (the completion of the house construction). There are two alternatives to reach state J from state B: through D and H or through E. The first possibility must involve activities E2, E6, and E7, in this order; the second possibility must involve activity E8 which must be preceded by the activity E3. Similarly, there are two ways to reach node K from node C.

Today, EOSDIS products do not have this choice of more than one way of computing. However, as many variants for spatial correlation exemplify, in order to compute a series of products, a number of generation programs can be implemented (especially if it only requires little effort). For example, instead of directly computing a product from its inputs (lower-level products), those inputs can be preprocessed into a database that will have auxiliary data structures and features (e.g., indexing and clustering) and then more efficient evaluation strategies can be chosen. Such strategy may well pay off if a many products are used the preprocessed data, thus amortizing the preprocessing effort. Or, one can consider a strategy when a number of products are evaluated “on the fly” in parallel, when input streams are used in more then one computation. That is, when a block is retrieved from secondary or tertiary storage, or produced by a product program, it is used in as many places as possible, while it still reside in main memory. Advantages of each choice depend on the overall data and system state. Workflow optimization is a natural way to make optimal choices dynamically depending on the system state.

The control flow graph is most appropriate for depicting the local execution dependencies of the activities in a workflow; it is a good way to visualize the overflow of control. Control flow graphs are the primary specification means in most commercial implementations of workflow management systems. As seen in the above example, a typical graph specifies the initial and the final activity (or a state) in a workflow, the successor activities for each activity in the graph, and whether these successors must all be executed concurrently, or it suffices to execute just one branch nondeterministically. Intuitively, concurrent execution corresponds to AND edges in the graph, whereas non-deterministic choice corresponds to OR edges. Edges in a control flow graph can be labeled with transition conditions. The condition applies to the current state of the workflow (which, in a broad sense, may include the current state of the underlying database, the output of the completed tasks, the current time, etc.). When the task at the tail of an edge completes, the task at the head can begin only if the corresponding transition condition evaluates to true. The Workflow Management
Coalition [10] identifies additional controls, such as loops and sub-workflows.

In the following, we present some work in progress on workflow optimization, to be used for global optimization in multi-product generation environment. This is a joint work with Larry Kerschberg from GMU and Samuel Varas from University Chile at Santiago, and currently visiting GMU.

12 Workflow Representation and Scheduling

In this section we introduce a workflow graph representation called Workflow Graphs (G), and we show how the workflow characteristics are represented in such graphs. Intuitively, Figure 1 represents a Workflow Graph. The notion of Workflow Graph is formalized as follows.

**Definition 1.** A Workflow Graph G is a triple \((V, E, W)\)^{11}, where:

1. \(V = \{v_1, \ldots, v_n\}\) is the set of nodes (called events),
2. \(E = \{e_1, \ldots, e_m\}\) is a set of edges (called activities). Each \(e_i\) in \(E\) is a pair \((T(e_i), H(e_i))\) where \(T(e_i) \in V\), called the tail of \(e_i\), and \(H(e_i) \subseteq V - \{T(e_i)\}\) is called the head of \(e_i\), and
3. \(W : E \to \mathbb{R}^k\) is a weight function that maps each activity \(e_i\) in \(E\) to a \(k\)-tuple of weights \((w_1, \ldots, w_k)\).

Note that the tail of \(e_i\) \(T(e_i)\) represents the incoming node of \(e_i\), and head of \(e_i\) \(H(e_i)\) represents the set of all outgoing nodes of \(e_i\).

The Workflow Graph that corresponds to Figure 1 is as follows. \(N = \{a, b, c, d, e, f, g, h, i, j, k\}\), \(E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}\), and \(e_1\) is \((a, \{b, c\})\), \(e_2\) is \((b, \{d\})\), \(e_3\) is \((b, \{e\})\), \(e_4\) is \((c, \{f\})\), \(e_5\) is \((c, \{g\})\), \(e_6\) is \((d, \{h\})\), \(e_7\) is \((h, \{j\})\), \(e_8\) is \((e, \{f\})\), \(e_9\) is \((f, \{i\})\), \(e_{10}\) is \((g, \{k\})\), \(e_{11}\) is \((j, \{k\})\), and \(e_{12}\) is \((i, \{k\})\).

We denote by \(O(v_i)\) the set of all outgoing edges of node \(v_i\), i.e., \(O(v_i) = \{e_j | T(e_j) = v_i\}\), and by \(I(v_i)\) the set of all incoming edges at node \(v_i\), i.e., \(I(v_i) = \{e_j | v_i \in H(e_j)\}\). In addition, we denote by \(|e_i|\) the cardinality of \(e_i\), i.e., the total number of incoming and outgoing nodes of \(e_i\). The size of a Workflow Graph \(G\), denoted by \(size(G)\), is defined as \(\sum_{e \in E} |e_i|\).

Now, we define workflow graph paths and workflow execution plans on Workflow Graphs.

**Definition 2.** Let \((V, E, W)\) be a Workflow Graph. Then, a path from a node \(s\) to a node \(t\) of length \(q\), denoted by \(P_{st}(q)\), is a sequence of nodes and edges, \((s = \ldots = t)\).
Figure 7: Path on Workflow Graph

\[ v_1, e_1, v_2, e_2, \ldots, v_j, e_j, v_{j+1}, \ldots, v_q, e_q, v_{q+1} = t, \text{ such that for all } j, 1 \leq j \leq q, v_j = T(e_j) \text{ and } v_{j+1} \in H(e_j). \]

As example, in Figure 2 \((a, e_1, c, e_4, f, e_9, i, e_{12}, k)\) is a path from node \(a\) to node \(k\).

If \(t = s\), \(P_{st}(q)\) is said to be a cycle. In a simple path all edges are distinct, and a simple path is elementary if all nodes \(v_1, v_2, \ldots, v_{q+1}\) are distinct. Simple and elementary cycles we define similarly. A path is said to be cycle-free if it does not contain any subpath which is a cycle.

To execute a workflow we need to perform a set of activities with a predefine order. We introduce the concept of execution plan to formalize how workflow can be executed.

To understand intuitively, an example of execution plan is depicted in Figure 3, where workflow starts at event \(a\), then activity \(e_1\) is executed. After that, activities \(e_2, e_6, e_7,\) and \(e_{11}\) are executed sequentially. In parallel to that, activities \(e_5, e_9,\) and \(e_{12}\) are executed sequentially. More formally.

**Definition 3.** Let \((V, E, W)\) be a Workflow Graph. An hyperpath, also called execution plan interchangeably, from an event \(s\) to an event \(t\), denoted by \(\Pi_{st}\), is a minimal Workflow Graph \((V_{\Pi}, E_{\Pi}, W_{\Pi})\), such that:

1. \(E_{\Pi} \subseteq E\)
2. \(s, t \in V_{\Pi} \subseteq V\)
3. For every \(v \in V_{\Pi}\), \(v\) is connected to \(t\) in \((V_{\Pi}, E_{\Pi}, W_{\Pi})\) by a cyclic-free simple path, and
4. $W_\Pi$ and $W$ agree on $E_\Pi$, i.e., for every $e \in E_\Pi$, $W_\Pi(e) = W(e)$.

Figure 3 depicts an example of an execution plan from event $a$ to event $k$. In this case $V_\Pi = \{a, b, c, d, f, h, i, j, k\}$, $E_\Pi = \{e_1, e_2, e_4, e_6, e_7, e_9, e_{11}, e_{12}\}$, and $W_\Pi = \{W_1, W_2, W_4, W_6, W_7, W_9, W_{11}, W_{12}\}$.

Note that there are 4 possible workflow execution plans in the Workflow Graph depicted in Figure 1. In general, for larger or more complex Workflow Graphs the number of possible execution plans grows exponentially. Therefore, analyzing execution plans’ properties on a Workflow Graph could be a hard problem. We consider the following problem characterization to analyze workflow execution plans’ properties.

Let $G = (V, E, W)$ be a Workflow Graph, and $\Pi$ the set of all execution plans from a node $v_0$ to a node $v_d$ on $G$. We suggest associating an optimal criterion $f$ to execution plans on $\Pi$, i.e., $f : \Pi \rightarrow \mathbb{R}$. Also, execution plans in $\Pi$ may be constrained, i.e., constraints $C(\rho_k)$ are given, $\rho_k \in \Pi$. Then, we consider the following problems.

1. Consistency: Determine if there exists an execution plan that satisfies $C$.

2. Verification:
3. Scheduling: Find an execution plan where $C$ holds.

4. Scheduling Optimization: Find the best workflow execution plan between a node $v_o$ and a node $v_d$, i.e., the optimization problem is as follows:

$$
\min_{\rho \in \Pi} \{f(\rho)\} \\
\text{s.t. } C(\rho_k)
$$

(1)

In general, complexity of (1) corresponds to the number of elements in set $\Pi$, which are in the worse case exponential in the size of $G$. A possible solution strategy to (1) corresponds to to examines all their elements. However, there are alternative mechanisms to solve (1) rather than an exhaustive enumeration. Recursive optimization formulations have been proposed, but in general, the problem is a NP-hard [? , ?].

In the following sections we analyze different cases of problem (1), and provide algorithms to solve effectively.

13 Unconstrained Localizable Scheduling

In this section we consider the case of finding a solution of (1) when there are no constraints other than those defined by the graph. Still the problem is NP-hard in general, and we first consider a special important class of objective functions, that we call locally computable.

13.1 Problem Statement

**Definition 4.** Let $G = (V, E, W)$ be a Workflow Graph, and $\Pi$ be the set of all hyperpaths in $G$. We say that a function $f : \Pi \to \mathbb{R}$ is a locally computable function if for all pair of nodes $(v_o, v_d)$, such that $\rho_{od} \in \Pi$, $f$ can be expressed as follows:

$$
f(\rho_{od}) = \begin{cases} f_0, & \text{if } v_o = v_d \\ w(e) + \mathcal{F}(\{f(\rho_{kd}) \mid v_k \in H(e)\}), & \text{if } v_o \neq v_d. \end{cases}
$$

where $\rho_{od}$ starts with edge $e$, $\mathcal{F}$ is a nondecreasing function in terms of $\{f(\rho_{kd}) \mid v_k \in H(e)\}$, and $f_0$ is a real value.

An important locally computable function corresponds to the workflow execution time, i.e., the necessary time to carry out the workflow. In this case, $w$ represent the activity execution time for each $e_i \in E$, and function $\mathcal{F}$ is as follows.

$$
\mathcal{F}(\{f(\rho_{kd}) \mid v_k \in H(e)\}) = \max_{v \in H(e)} \{f(\rho_{kd})\}
$$

(2)
Next proposition provides a mechanism to re-write (1) as a recursive optimization problem.
Proposition 1. Let $G = (V, E, W)$ be a Workflow Graph, $\Pi_{ad}$ be the set of all hyperpaths from $v_o$ to $v_d$, and $f : \Pi \rightarrow \mathbb{R}$ be a locally computable function. Then,

$$\begin{align*}
1. \min_{\rho \in \Pi} f(\rho) &= f_0 \\
2. \min_{\rho \in \Pi} f(\rho) &= \min_{e \in O(v)} \{w(e) + F(\{\min_{\rho' \in \Pi} f(\rho') \mid v_k \in H(e)\})\}
\end{align*}$$

(3)

Proof.

Problem (3) correspond to the generalized Bellman's equations [?, ?], and its solution is a generalization of the well-known shortest-path problem. Now, we present an algorithm to solve (3) in linear time.

13.2 Shortest Execution Plan Algorithm

The shortest execution plan algorithm [?, ?] assumes that $G$ is ordered in inverse topological order, i.e., nodes are enumerated, such that the following condition is satisfied:

$$T(e_k) = v_i \land (v_j \in H(e_k)) \Rightarrow v_j < v_i.$$  

(4)

The following algorithm Acyclic – ITO [?, ?] has as input a Workflow Graph ($G$), and its outputs are whether $G$ is acyclic, and if it is, the inverse topological order. The algorithm is as follows.

Procedure Acyclic-ITO($G$)

for each $i \in V$ do $r_i = 0$

for each $e = (\{i\}, H(e)) \in E$ do $r_i = r_i + |T(e)|$

$k = 0; Q = \emptyset$

for each $i \in V$ if $r_i = 0$ then $Q = Q \cup \{i\}$

while $Q \neq \emptyset$ do

select and remove $u \in Q$

$k = k + 1; \pi_u = k$

for each $e \in (\{i\}, H(e)) \in I(u)$ do

$r_i = r_i - 1$

if $r_i = 0$ then $Q = Q \cup \{i\}$

if $(k = n)$ then

return "$WFG$ is acyclic"

else return "$WFG$ is not acyclic"

\footnote{Q is implemented as queue}
When \( G \) is acyclic, then \( \pi_u \), for all \( u \in E \), provides the inverse topological order. Since each edge is examined only one, complexity of \( Acyclic - ITO \) correspond to \( O(size(G)) \) [?].

Now, the algorithm to find the shortest execution plan in a Workflow Graph \( G \). The input are the destination node \( v_d \) and the Workflow Graph \( G \). The algorithm is as follows.

**Procedure SEP-Acyclic\((v_d, G)\)**

for each \( i \in V \) do

\[ P_i = 0 \]

if \( (i = v_d) \) then \( f(i) = 0 \) else \( f(i) = \infty \)

for each \( e_j \in E \) do \( k_j = 0 \)

for \( i = 1 \) to \( |V| - 1 \) do

for each \( e_j = (y, H(e_j)) \in I(i) \) do

\[ k_j = k_j + 1 \]

if \( k_j = |H(e_j)| \) then

\[ \eta = w(e_j) + \mathcal{F}(\{f(i) \mid i \in H(e_j)\}) \]

if \( f(y) > \eta \) then

\[ f(y) = \eta \]

\[ P_v = e_j \]

The output is a set \( \{P_i \mid i \in V\} \) indicating, for each node \( i \), which edge has been selected. Therefore, the execution path is constructed by a forward recursively enumeration, starting from the origin node of \( G \).

To analyze the complexity of \( SEP - Acyclic() \) algorithm, we first note that each node and each edge is selected at most once. Therefore, the overall complexity is \( O(size(G)) \) [?].

### 14 Constrained Localizable Scheduling

In this section we analyze the constrained problem (1) under *locally computable* objective functions assumption. In general, this is a harder problem than the unconstrained one, because feasibility (constraint satisfaction) needs to be checked at each hyperpath. We will consider a special case of constraints called *locally computable constraints*, which are defined as follows.
14.1 Problem Statement

Definition 5. Let $G = (V, E, W)$ be a Workflow Graph, and $\Pi$ be the set of all hyperpaths to a node $v_d$ in $G$. Let $g : \Pi \rightarrow \mathbb{R}^+$ be a function and $R$ be a positive real value. Then, we say that a constraint $s(\Pi_k) = R - g(\Pi_k) \geq 0$, $\Pi_k \in \Pi$, is a locally computable constraint if for all $\Pi_{od} = (V_{\Pi}, E_{\Pi}, W_{\Pi}) \in \Pi$, $s$ can be expressed as follows:

$$
\begin{align*}
  s(\Pi_{dd}) &= R \\
  s(\Pi_{od}) &= -w_2(e_j) + S(\{s(\Pi_{kd}) | v_k \in H(e_j)\}) \geq 0, \{e_j\} = E_\Pi \cap O(v_o)
\end{align*}
$$

where $S$ is a nondecreasing function in terms of $\{s(\Pi_{kd}) | v_k \in H(e_j)\}$. Function $s$ is called slack constraint, and $s(\Pi_{od})$ is the slack available at node $v_o$.

This type of constraints can be associated with execution time (where $R$ represents the maximum execution time required). Other characterization correspond to traversal cost, rank, or distance [?, ?].

Finding a feasible hyperpath recursively requires to define some criteria to discriminate at each step whether an edge may belong to a feasible hyperpath. We suggest as a criterion the minimum feasible slack defined as follows.

Definition 6. Let $G = (V, E, W)$ be a Workflow Graph, $s$ be a slack constraint, and $\Pi$ be the set of all possible hyperpaths from a node $v_o$ to a node $v_d$. We say that $(r_1, r_2, \ldots, r_n)$ is a $n$-tuple of minimum feasible slack at nodes in $V$, defined by

$$
r_i = \begin{cases} 
\min_{\Pi \in \Pi} \{s(\Pi_{id}) | s(\Pi_k) \geq 0\}, & \text{if } v_i \in \bigcup_{\Pi \in \Pi} V_{\Pi} \\
0, & \text{otherwise.}
\end{cases}
$$

Now we can formulate (1) as a constrained recursive optimization problem as follows.

Proposition 2. Let $G = (V, E, W)$ be a Workflow Graph, $s$ be a locally computable objective function, $f$ be a slack constraint, and $(r_1, \ldots, r_n)$ be a vector of minimum feasible slacks. Then, the optimal feasible hyperpath from a node $v_o$ to a node $v_d$ problem (1) is equivalent to the following optimization problem.

$$
\begin{align*}
  f(\Pi_{id}) &= \min_{e \in O(v)} \{w_1(e_j) + F(\{f(\Pi_{kd}) | v_k \in H(e_j)\}) | \\
  s(\Pi_{id}) &= -w_2(e_j) + S(\{s(\Pi_{kd}) | v_k \in H(e_j)\}) \geq r_i, \\
  \forall v_i \in V - \{v_D\} \\
  s(\Pi_{dd}) &= R, \\
  f(\Pi_{dd}) &= f_0.
\end{align*}
$$

\footnote{We assume that $w_1$ is related to the objective function and $w_2$ to the constraint.}
where $F$ and $S$ are nondecreasing functions, and $f_0$ is a constant and $R$ is a positive constant.

Proof.

14.2 Constrained Execution Plan Algorithm

The algorithm also requires a topological order of nodes, which is provided by algorithm Acyclic – ITO. Then, the set of all minimum feasible slack is calculated with the following algorithm.

\begin{verbatim}
Procedure MinFeSlack(G)
  for each $i \in V$ do
    if ($i = v_0$) then $r_i = 0$ else $r_i = \infty$
    for $i = |V|$ to 1 do
      for each $k \in H(e_j) \mid i = T(e_j)$ do
        if $r_i + w(e_j) < r_k$ then
          $r_k = r_i + w(e_j)$
\end{verbatim}

This algorithm examines all nodes and edges once, then, its complexity is $O(size(G))$. Finally, a modification of SEP – Acyclic algorithm provides the solution of our constraint problem (6).
Procedure SCP-Acyclic(vd, G)
for each i ∈ V do
    \( P_i = 0 \)
    if (i = vd) then
        \( f(i) = 0, s(i) = R \)
    else
        \( f(i) = \infty, s(i) = 0 \)
for each \( e_j \in E \) do \( k_j = 0 \)
for i = 1 to \(|V| - 1\) do
    for each \( e_j = (y, H(e_j)) \in I(i) \) do
        \( k_j = k_j + 1 \)
        if \( k_j = |H(e_j)| \) then
            \( \eta = w_1(e_j) + \mathcal{F}(\{f(i) \mid i \in H(e_j)\}) \)
            \( \nu = -w_2(e_j) + S(\{s(i) \mid i \in H(e_j)\}) \)
            if \( f(y) > \eta \) and \( \nu \geq r_i \) then
                \( f(y) = \eta \)
                \( s(y) = \nu \)
                \( P_y = e_j \)

Algorithm SCP-Acyclic for each node analyzes if there going to be a enough slack to reach the root node. When there is no enough slack, this is an infeasible possibility, and then, it is not consider. Since this is just a variation of SEP-Acyclic, with additional evaluation (slack at each node), its complexity is \( O(size(G)) \).

15 Additive Unconstrained Optimization

In this section we extend the concept of locally computable function an optimization. We reduce problem (1) under certain condition to a network flow (integer linear) problem, where the solution is always integer.

15.1 Problem Statement

Since locally computable functions require that their values being calculated just in terms of local elements, many types of functions cannot satisfy this condition. For example, total cost, total execution time, or any function where its value depends of an hyperpath. To overcome this limitation, we introduce additive functions defined as follows.
Definition 7. Let \( G = (V, E, W) \) be a Workflow Graph, and \( \Pi \) be the set of all hyperpaths in \( G \) from a node \( v_o \) to the node \( v_d \). Then, we say that a function \( f : \Pi \to \mathbb{R} \) is a additive function if for all \( \Pi_{ad} = (V_{\Pi}, E_{\Pi}, W_{\Pi}) \in \Pi \), \( f \) can be expressed as follows:

\[
f(\rho_{ad}) = \sum_{e \in \rho} w(e)
\]

Now, we introduce the concept of hyperflow on a Workflow Graph \([?]\). Intuitively, an hyperflow is a flow in hypergraphs, where for each node a conservation flow equation is specified. In particular, we consider that an unit flow is introduced in node \( v_o \) of \( G \) and a unit flow is gotten in node \( v_d \). We define the following rule when an edge \( e \) has \( |H(e)| \geq 2 \): for each unit incoming flow at edge \( e \) the outcoming flow is \( |H(e)| \), i.e., additional flows are created for each \( v_i \in H(e) \).

Although previous rule preserves the unitary of the flow, it creates an additional problem by the artificial flow creation. To fix that, we introduce the concept of \( h \)-artificial flow edge at each node as follows.

Definition 8. Let \( G = (V, E, W) \) be a Workflow Graph. We say that edge \( e_i \) is an \( h_i \)-artificial flow edge at node \( v_k \) if: (a) \( |H(e_i)| \geq h_i \) and \( h_i \geq 2 \), and (b) there exists \( h_i \) different paths from \( e_i \) to \( v_k \) without common elements except \( e_i \) and \( v_k \).

From Figure 1, edge \( e_1 \) is a 2-artificial edge to node \( k \), i.e., a unit flow incoming to edge \( e_1 \) becomes in two units of flows incoming to node \( k \).

Now, we can define formally the concept of unit hyperflow in a Workflow Graph as follows.

Definition 9. Let \( G = (V, E, W) \) be a Workflow Graph, and \( \theta(v_k) \) be a set of \( h_i \)-artificial flow edges at node \( v_k \), \( v_k \in V \). We say that an unit hyperflow from \( v_o \) to \( v_d \) in \( G \) is a function \( x : E \to \{0\} \cup \mathbb{R}^+ \) that satisfies the following conservation constraints:

\[
\sum_{e \in O(v)} x_e - \sum_{e \in I(v)} x_e - \sum_{e \in \theta(v)} (h_u - 1) x_e = \begin{cases} 
1, & \text{if } v_i = v_o \\
0, & \text{if } v_i \neq v_o \text{ and } v_i \neq v_d \\
-1, & \text{if } v_i = v_d
\end{cases}
(8)
\]

Note that (8) can be represented in a matricial way \( A\bar{x} = \bar{d} \), where \( \bar{x} = (x_1, \ldots, x_m)^t \), \( \bar{d} = (1, 0, \ldots, 0, -1)^t \), and a \( n \times m \) matrix \( A \), called unitary incidence matrix, where

\[\text{We consider that } v_o \text{ and } v_d \text{ are in the first and last position of } \bar{d} \text{ respectively.}\]
its $(i,j)$ element correspond to:

\[
a_{ij} = \begin{cases} 
+1, & \text{if } v_i = T(e_j) \\
-1, & \text{if } v_i \in H(e_j) \\
1 - h_u, & \text{if } e_u \in \theta(v_i) \\
0, & \text{otherwise.}
\end{cases}
\]  

(9)

Matrix $A$ has two important properties, namely: (a) it is an integer matrix, i.e., all elements are integer, and (b) $\sum_{i=1}^{n} a_{ij} = 0$. Last property comes from the fact that for each edge $e$ all incoming flow is balanced with the outcoming flow. Next proposition provides the basis for the solution of system $Ax = \bar{d}$.

**Theorem 1.** Let $G = (V, E, W)$ be a Workflow Graph, and $A\bar{x} = \bar{0}$ be the hyperflow constraint conservation on $G$. If $A = (B R)$ and $\bar{x} = (\bar{x}_B, \bar{x}_R)$, where $B$ is a base of $A\bar{x} = \bar{0}$. Then, for every base $B$ of $A\bar{x} = \bar{0}$ there exists an hyperpath $\Pi$ in $G$ and vice versa.

**Proof.**

Now, we re-write the optimization problem (1) for additive functions as an hyperflow conservation problem, where a unit flow is sent from node $v_o$ to a node $v_d$.

**Proposition 3.** Let $G = (V, E, W)$ be a Workflow Graph, $\Pi$ be the set of all hyperpaths, $f : \Pi \rightarrow \mathbb{R}$ be an additive function, and $A\bar{x} = \bar{d}$ be the unit hyperflow constraint conservation of $G$. Then,

\[
\min \sum_{e \in E} w(e_j) x_j \\
\text{s.t. } A\bar{x} = \bar{d}
\]  

(10)

**Proof.**

Therefore, we can use the Simplex algorithm to solve (10), and the solution is an hyperpath that minimizes the objective function. Next section presents an algorithm to determine the unitary incidence matrix $A$.

**15.2 Algorithm**

The only remaining part from previous section is how matrix $A$ is determined. The algorithm consists in two steps, where the first one determines an intermediate matrix...
A' as follows.

\[
    a'_{ij} = \begin{cases} 
        +1, & \text{if } v_i = T(e_j) \\
        -1, & \text{if } v_i \in H(e_j) \\
        0, & \text{otherwise.}
    \end{cases}
\]

(11)

Then, this intermediate matrix A' is fixed by adding the $h_i - \text{artificial}$ edges in its values. This step assume that the Workflow Graph is in inverse topological order, and each edge $e$ has $\psi$ a set of pairs $(e, K)$ to indicate the $k^{th}$ node in the $H(e)$. This step is as follows.

**Procedure IncidenceMatrix(A', G)**

\begin{algorithmic}
\For {each $v_i \in V$} 
\State $k_i = 0$
\EndFor
\For {each $e_j \in E$}
\State $\psi_j = \emptyset$
\For {q = 1 to $|H(e_j)|$}
\State $\psi_j = \psi_j \cup (j, q)$
\EndFor
\For {i = $|V|$ to 1}
\For {each $e_j \in \Omega(i)$}
\If {$|\alpha_j| \geq 2$}
\State $a'_{ij} = a'_{ij} - |\alpha_j|$
\State $\psi_j = \psi_j - \alpha_j$
\EndIf
\If {$|\psi_j| \geq 2$}
\For {each $e_k \in O(i)$}
\State $\psi_k = \psi_k \cup \alpha_j$
\EndFor
\EndIf
\EndFor
\EndFor
\EndFor
\end{algorithmic}

Where set $\Omega(i) = \{ e \mid \exists(e, r) \in \cup_{u \in I(v)} \psi_u \}$ and set $\alpha_j = \{(e, q) \mid \exists(e^*, r) \in \cup_{u \in I(v)} \psi_u, e = e^* \land r \neq q \}$.

Algorithm **IncidenceMatrix(A', G)** analyzes each node once, and for each node calculate set $\Omega(i)$ and for element in such set calculates $\alpha_j$. Then, the overall complexity corresponds to the complexity of set $\Omega(i)$ times complexity of set $\alpha_j$ times the number of nodes in $G$, i.e., $O(|V| \times O(\Omega) \times O(\alpha))$.

16 Extended Workflow Scheduling Problem

In this section we extend the workflow scheduling problem considering that there are some constraints over a subset of activities indicating if those activities may or may not mandatory be executed. As example of this type of constraints consider the case
when two activities $e_1$ and $e_2$ are simultaneously allowed or disallowed to appear in an feasible execution plan, i.e., the logic constraint is $(\neg e_1 \land \neg e_2) \lor (e_1 \land e_2)$.

16.1 Problem Statement

In general, finding a feasible execution plan under such constraints makes the problem NP-hard, because there is necessary to check all combinations of allowed activities. Before we formulate our optimization model we need to define the notion of feasible instantiation as follows.

Definition 10. Let $G = (V, E, W)$ be a Workflow Graph, $\hat{C}$ be a constraint over activities $e_1, \ldots, e_k$, and $\vec{w} = (\omega_1, \ldots, \omega_k)$ be a Boolean vector, such that $\omega_i = TRUE$, $1 \leq i \leq k$, if $e_i$ must be in the execution plan, FALSE if $e_i$ must not be in the execution plan. Then, we say that an instantiation of $\vec{w}$ is feasible if it satisfies $\hat{C}$.

Note that there are $2^k$ possible instantiations of $\vec{w}$ to check whether is feasible or not, which is the source of the NP-hardness.

The optimization problem can be formulated similar to (1) with the additional constraint $\hat{C}(e_1, \ldots, e_k)$. Then finding the best execution plan between node a $v_o$ and a node $v_d$ can be formulated as

$$\min_{\Pi \in \Pi} \{f(\Pi_k)\}$$
$$s.t. \ C(\Pi_k)$$
$$\hat{C}(e_1, \ldots, e_k)$$ (12)

In general, solving problem (12) requires exponential time. However, when a feasible instantiation of $\vec{w}$ is given, problem (12) is similar to (1). This observation is used to propose a local search algorithm [?, ?, ?] in the next section. In the following, we consider that function $f$ and constraint $C$ satisfy the condition from sections 13, 14, and 15.

16.2 Local Search Algorithm

A local search algorithm is structured as follows: a number of local searches are performed, where for each one, the algorithm checks if the local optimum is better than the current objective function value. This procedure is repeated until there is no acceptable neighborhood possible or some criteria are satisfied (number of iterations, minimum value, etc.).

In this case acceptable neighborhoods correspond to feasible instantiations of $\vec{w}$. Therefore, we need a procedure to generate those neighborhoods. There are many
possible ways to implement this procedure, (sequential, random , etc.). However, we
do not discuss them here, and we just consider that there is such a procedure called
GetNextNeighborhood().

The local search framework get a $\omega$ instantiation, let say $\omega_k$, and modifies $G$ with
that information (assigns an huge cost to the activities with FALSE). The resulting
problem is one of the problem formulated in sections 13, 14, and 15. Therefore, we
can use the their algorithms to solve the local search (Step 1). The local search
framework is as follows.

Step 0. Assign 0 to $k$, $\omega_k = \text{GetNextNeighborhood}()$, and $f^* = \infty$.

Step 1. Assign $w_i = \infty$ if $\omega_i = \text{FALSE}$, and perform a local search, i.e., solve the
shortest execution plan, getting solution $\Pi_k$.

Step 2. If $f(\Pi_k) < f^*$, then $f^* = f(\Pi_k)$, and $\Pi^* = \Pi_k$.

Step 3. Increase $k$ by 1, select a new $\omega$ instantiation, and go to step 2. If there is
no $\omega$ instantiation, go to step 4.

Step 4. Report objective function $f^*$ and solution $\Pi^*$.

Local search framework complexity, in the worst case, is exponential in terms of
$k$, i.e., the number of activities involved in constraint $\tilde{C}$. In particular, complexity
is $O(2^k \times | E | \times \text{size}(G))$, since we have to execute local search algorithm (shortest
execution path) at most $2^k$ times. However, using some additional stop criteria, it is
possible get good solutions without examine all possible combinations.
Part VI
Conclusions and Suggested Future Directions

Main Conclusions in Phase 1:

- EOSCUBE has the potential for significant productivity gain in specification and generation of EOSDIS and other scientific products
- Generation of scientific products from real data sets is feasible using the EOSCUBE prototype
- An industrial-strength EOSCUBE implementation will be necessary for deployment and massive use of the system.
- The EOSCUBE language should allow incremental extensions, which are unavoidable in diverse scientific domains
- The overall evaluation model should also support data-flow processing (i.e. pipeline evaluation), in addition to query processing.
- The main aspects of global optimization should deal with interleaved pipelined evaluation of series of inter-related products, and concentrate on optimizing throughput via data flow control, buffer management, and materialization supporting clustering and indexing.

Future Action Paths for EOSCUBE:

We elaborate on recommended activities in Section VI. Below is a summary of main paths of action that will have to be carefully discussed and planned with EOSDIS.

Research Path

- Optimization algorithms for workflow (for data flow evaluation)
  - Optimization for quasi-views, which dynamically control with each product query, when the evaluation is postponed and when is restarted
  - Specialized techniques for spatio-temporal indexing and clustering
  - Optimization of materialized views, which support (especially created) intermediate results to support clustering and indexing
- GIS constraint algebras, which will support interoperability with GIS under unified constraint model.

**Industrial-strength implementation path**
- CCUBE/EOSCUBE core, focusing on performance for individual queries.
- Pipeline evaluation model.
- ODBC support, and through it, a range of object managers and DBMS.
- Platforms support, including mass storage system.
- Workflow optimization module.
- GIS integration.

Collaborative work with Earth scientists on a specific set of new products, and continued customization of EOSCUBE for them. This will also used as a leverage for later massive deployment of EOSCUBE.

**Deployment of EOSCUBE to Centers and Support**

**References**


M. Stonebraker, M. Rowe, and L. Hiroshama. The implementation of Postgress. IEEE Transactions on Knowledge and Data Engineering, 1990.


Appendix A
Query 5 \( (\text{SST}, \text{CLD correl.} \geq 0.6) \)
Query 3 (SST difference > 2.0)
Query 4 (input area)
Query 2 \((\text{cl}d \geq 75.0)\)
Query 1 output:

0.0846923

Query 4 output:

0.35008 0.235478 0.0327797 0.0294338 0.056053 0.0490407 0.0870465 0.0434318 0.000460318 0.2558 0.321847 0.156925 0.0482001 0.159197 0.130963 0.149418 0.10525 0.021373 0.0475394 -0.0249535 -0.0012261 0.0866486 0.317112 -0.0618164
Appendix B
County
String name;
String Color;
CST(lat,lon) extent;

Grid
Vector< PIXEL>

PIXEL
Double NDVI;
Double SST;
Double CLD;

Simplified Schema
**Title and Subtitle**

A Constraint Database System for High-Level Specification and Efficient Generation of EOSDIS Products

**Abstract**

The EOSCUBE constraint database system is designed to be a software productivity tool for high-level specification and efficient generation of EOSDIS and other scientific products. These products are typically derived from large volumes of multidimensional data which are collected via a range of scientific instruments.