How to Build Your Own Moving Objects Database System

Ralf Hartmut Güting

Fernuniversität Hagen, Germany

Purpose of this talk

- Explain and demonstrate a prototype of a moving objects database system

In more detail:

- Explain and demonstrate Secondo, an environment for building database system prototypes for research and teaching
- Show how a moving objects DBMS prototype can be built in Secondo, and show it running

Thanks

Secondo

- Victor Almeida
- Thomas Behr
- Stefan Dieker
- Zhiming Ding
- Christian Düntgen
- Frank Hoffmann
- Markus Spiekermann
- Ulrich Telle

Moving Objects

- Victor Almeida
- Mike Böhlen
- Zhiming Ding
- Martin Erwig
- Luca Forlizzi
- Christian Jensen
- Nikos Lorentzos
- Enrico Nardelli
- Markus Schneider
- Michalis Vazirgiannis

Why should SECONDO be exciting for you? What can you do with it?

- Implement your new index structure as an algebra module in Secondo and test it in a complete system environment.
- Implement a new concept that affects all levels of a system (kernel, optimizer, user interface). Example: progress estimation.
- Experiment with query processing techniques without being disturbed by an optimizer.
- Write optimization rules and cost functions for your new index structure or join algorithm.
- Play with non-traditional data models.
- Write application-specific extensions including the user interface. Example: algebra for chess games.
- Use Secondo in teaching architecture and implementation of database systems, and let students build extensions to it.

... and the moving objects component?

- Use it for applications, e.g. mobility analysis in a city.
- Write your own extensions. Examples: support for networks, periodic movement, uncertainty in movement, ...

Outline

1. Secondo
    - Kernel
    - Optimizer
    - GUI
2. Moving objects DBMS model and prototype
    - Spatio-temporal data types
        - Concept
        - Design
    - Demo

Secondo - Overview

An environment for implementing DBMS with new kinds of data models, suitable for research prototyping and teaching. Developed in the last ten years or so at University of Hagen, Germany.

- no fixed data model
- system frame can be filled with implementations of different data models, e.g.
  - relational
  - object-relational
  - graph/network-oriented
  - sequence-oriented
- goes beyond extensibility just by attribute data types
- system frame contains data model independent parts of a DBMS
- data model dependent parts implemented in algebra modules
- current “contents”: basically a relational system with several advanced data type packages

Open source software, available at

`http://www.informatik.fernuni-hagen.de/import/pi4/Secondo.html/`

Secondo - Overview

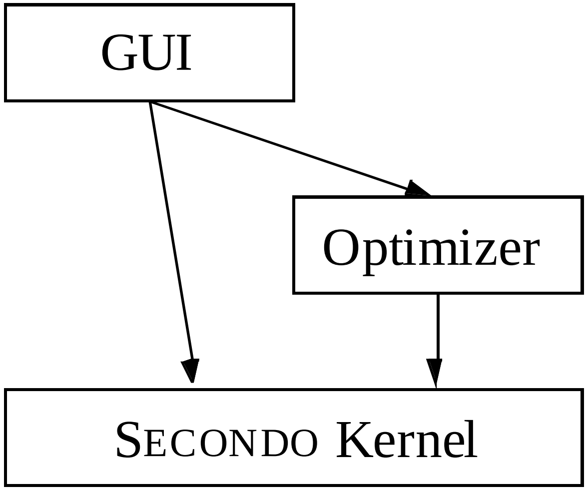
Three major components:

- Secondo Kernel:
  - implements specific data models
  - extensible by algebra modules
  - provides query processing over the implemented algebras
  - implemented on top of BerkeleyDB storage manager
  - written in C++
- Optimizer:
  - core capability: conjunctive query optimization
  - currently supports a relational model with an SQL-like language
  - written in PROLOG
- GUI:
  - extensible interface for an extensible DBMS like Secondo
  - extensible by viewers
  - sophisticated spatial / spatio-temporal viewer, extensible by data types
  - written in Java

Secondo - Overview

Components work together:

- GUI sends executable query (query plan) to the kernel, displays result
- GUI sends query to optimizer, receives plan, sends plan to kernel, displays result
- optimizer sends commands and executable queries to kernel to get information about DB objects, e.g. selectivities

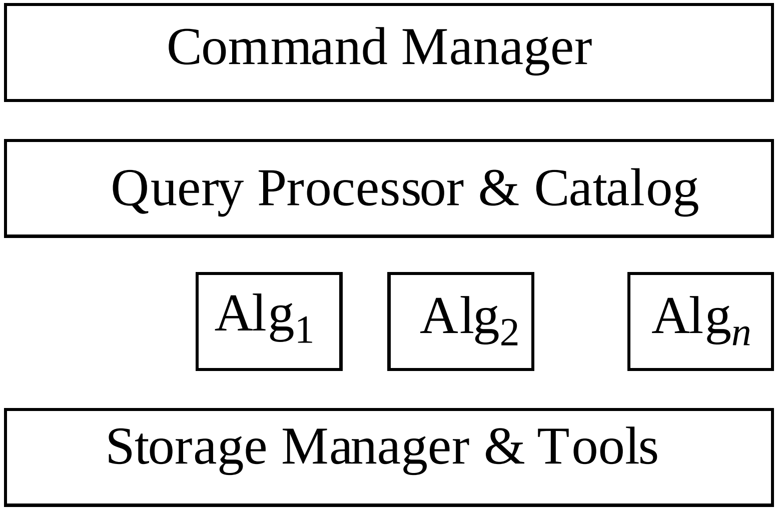


Secondo - Kernel

- Secondo Kernel:
  - implements specific data models
  - extensible by algebra modules
  - provides query processing over the implemented algebras
  - implemented on top of BerkeleyDB storage manager
  - written in C++

Secondo - Kernel

Rough architecture:



Command Manager

Query Processor & Catalog

Alg$_1$ Alg$_2$ Alg$_n$

Storage Manager & Tools

Secondo - Kernel

Underlying Concept: Second-Order Signature

A formalism to describe

- a *descriptive algebra*, defining a data model and query language,
- an *executable algebra*, specifying a collection of data structures and operations capable of representing the data model and implementing the query language,
- rules to enable a query optimizer to map descriptive algebra terms to executable algebra terms (*query plans*).

Secondo - Kernel - Second-Order Signature

Basic idea: Use two coupled signatures. The first signature describes a type system, the second an algebra over the types generated by the first signature.

Example:

$$\begin{array}{lll} & \to \text{DATA} & \underline{\mathit{int}}, \underline{\mathit{real}}, \underline{\mathit{bool}} \\ \text{DATA} & \to \text{SET} & \underline{\mathit{set}} \end{array}$$

Terms of the first signature (= types of the type system)

$\underline{\mathit{int}}, \underline{\mathit{real}}, \underline{\mathit{bool}}, \underline{\mathit{set}}(\underline{\mathit{int}}), \underline{\mathit{set}}(\underline{\mathit{real}}), \underline{\mathit{set}}(\underline{\mathit{bool}})$

are sorts of the second.

$\forall$ *data* in DATA.

$$\begin{array}{lll} \mathit{data} \times \mathit{data} & \to \underline{\mathit{bool}} & =, \neq, <, \leq, \geq, > \end{array}$$

Secondo - Kernel - Second-Order Signature

Specifying a Descriptive Algebra

kinds IDENT, DATA, TUPLE, REL

type constructors

	→	DATA	*int*, *real*, *bool*, *string*
(IDENT × DATA)$^+$	→	TUPLE	*tuple*
TUPLE	→	REL	*rel*

Example term (= type, schema):

rel(*tuple*([(name, *string*), (age, *int*)])

Secondo - Kernel - Second-Order Signature

Specifying a Descriptive Algebra

operators

$\forall$ *data* in DATA.

$data \times data \rightarrow \underline{bool}$ $=, \neq, <, \leq, \geq, >$ _ # _

$\forall$ rel: $\underline{rel}(tuple)$ in REL.

$rel \times (tuple \rightarrow \underline{bool}) \rightarrow rel$ **select** _ # [_]

$\forall$ $tuple$: $\underline{tuple}(list)$ in TUPLE, $attrname$ in IDENT,

$member(attrname, attrtype, list)$.

$tuple \times attrname \rightarrow attrtype$ **attr** # (_, _)

A query:

people **select**[**fun** (p: person) **attr**(p, age) > 20]

Secondo - Kernel - Second-Order Signature

Specifying an Executable Algebra

kinds IDENT, DATA, TUPLE, RELREP

type constructors

		DATA	*int*, *real*, *bool*, *string*
$(\text{IDENT} \times \text{DATA})^+$	→	TUPLE	*tuple*
TUPLE	→	RELREP	*srel*, *relrep*

Secondo - Kernel - Second-Order Signature

Specifying an Executable Algebra

operators

$\forall$ …

tuple in TUPLE.

$\underline{\mathit{relrep}}(\mathit{tuple})$	$\rightarrow$	$\underline{\mathit{stream}}(\mathit{tuple})$	**feed**	_ #
$\underline{\mathit{stream}}(\mathit{tuple}) \times (\mathit{tuple} \rightarrow \underline{\mathit{bool}})$	$\rightarrow$	$\underline{\mathit{stream}}(\mathit{tuple})$	**filter**	_ # [_]
$\underline{\mathit{stream}}(\mathit{tuple})$	$\rightarrow$	$\underline{\mathit{srel}}(\mathit{tuple})$	**consume**	_ #

A query plan:

people **feed** **filter**[**fun** (p: person) **attr**(p, age) > 20] **consume**

Secondo - Kernel - Second-Order Signature

Commands

A database is a pair (T, O) where T is a finite set of named types, and O a finite set of named objects. Seven basic commands to manipulate a database:

```
type <identifier> = <type expression>
delete type <identifier>

create <identifier>: <type expression>
update <identifier> := <value expression>
let <identifier> = <value expression>
delete <identifier>

query <value expression>
```

Secondo - Kernel - Commands

Basic Commands	Inquiries

`type <identifier> = <type expression>`
`delete type <identifier>`
`create <identifier>: <type expression>`
`update <identifier> := <value expression>`
`let <identifier> = <value expression>`
`delete <identifier>`
`query <value expression>`
	`list type constructors`
	`list operators`
	`list algebras`
	`list algebra <identifier>`
	`list databases`
	`list types`
	`list objects`

Databases	Transactions

`create database <identifier>`
`delete database <identifier>`
`open database <identifier>`
`close database`
	`begin transaction`
	`commit transaction`
	`abort transaction`

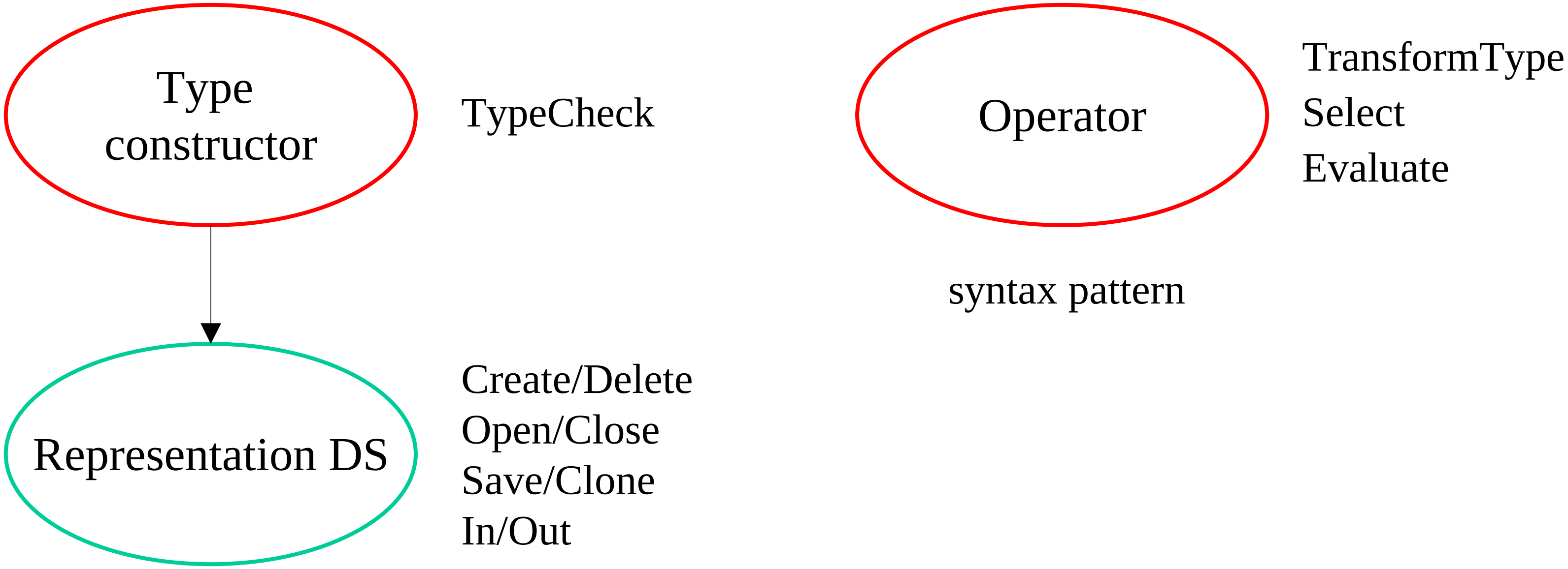
Import and Export	

`save database to <file>`
`restore database <identifier> from <file>`
`save <identifier> to <file>`
`restore <identifier> from <file>`	

Secondo - Kernel

Structure of Algebra Modules

Each algebra module offers some type constructors and some operators. The module contains:



Type constructor

TypeCheck

Representation DS

Create/Delete
Open/Close
Save/Clone
In/Out

Operator

TransformType
Select
Evaluate

syntax pattern

Secondo - Kernel – Algebra Modules

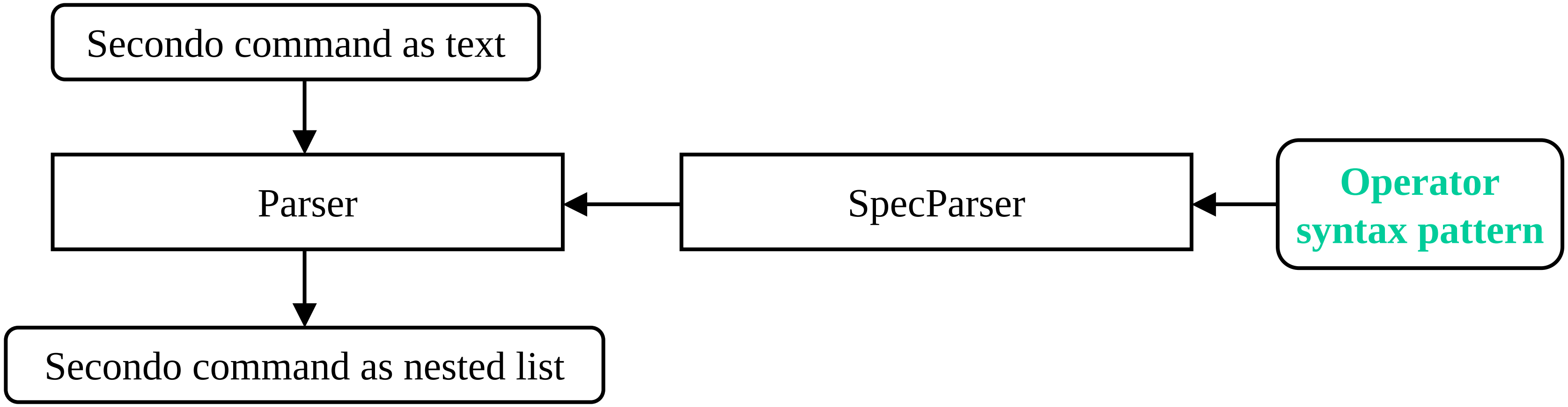
Some currently available algebra modules:

StandardAlgebra	*int*, *real*, *string*, *bool*
RelationAlgebra	*rel*, *tuple*
BTreeAlgebra	*btree*
RTreeAlgebra	*rtree*
SpatialAlgebra	*point*, *points*, *line*, *region*
DateTimeAlgebra	*instant*, *duration*
TemporalAlgebra	*periods*, *rangeint*, .., *mint*, *mreal*, *mbool*, *mpoint*
MovingRegionAlgebra	*mregion*

Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

```
query plz feed filter[.Ort = "Hagen"] consume
```



```
(query (consume (filter (feed plz) (fun (tuple1 TUPLE) (= (attr tuple1
Ort) "Hagen")))))
```

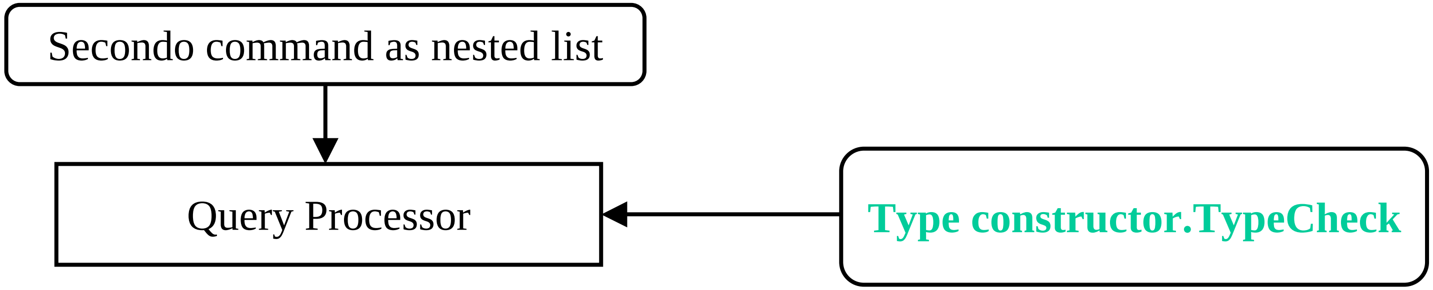
```
  operator feed alias FEED pattern _ op
  operator attr alias ATTR pattern op (_, _)
```

Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

Processing Type Expressions

```
type <identifier> = <type expression>
create <identifier>: <type expression>
```



Secondo command as nested list

Query Processor

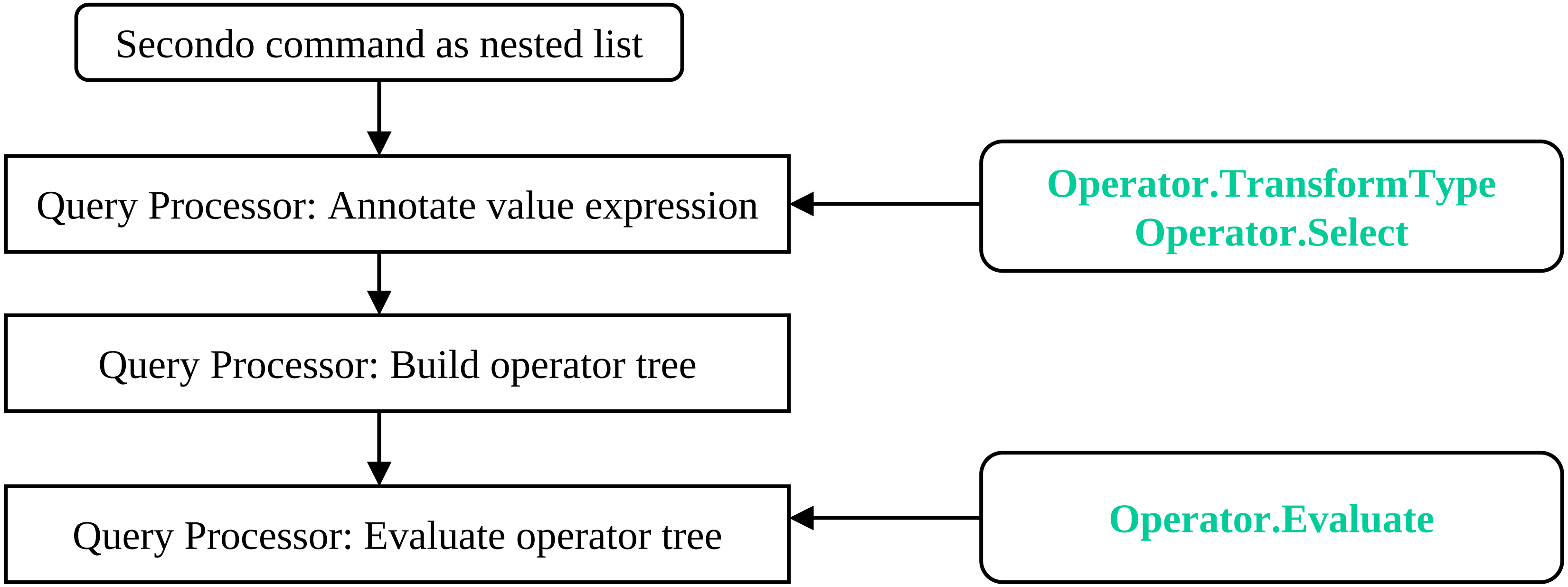
Type constructor.TypeCheck

Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

Processing Value Expressions

```
update <identifier> := <value expression>
let <identifier> = <value expression>
query <value expression>
```



Secondo command as nested list

Query Processor: Annotate value expression

Operator.TransformType
Operator.Select

Query Processor: Build operator tree

Query Processor: Evaluate operator tree

Operator.Evaluate

Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

Operator evaluation functions all have the same generic interface. They call query processor primitives to evaluate arguments (subtrees) that are functions or streams:

– getArguments – request	to evaluate a parameter function subtree
– open – request – close – received	to communicate with argument streams

Evaluation functions for stream operators return special values YIELD or CANCEL.

Secondo - Kernel

Demo: The Kernel

```
SecondoTTYBDB
list databases
list algebras
list algebra RTreeAlgebra
open database opt
query 3 * 5
create x: int
update x := 7
delete inc
let inc = fun(n:int) n + 1
query inc(inc(7))
query Orte
query plz count
query plz feed filter[.Ort = “Hagen”] consume
query plz_Ort plz exactmatch[“Hagen”] consume
query Orte feed {o} plz feed {p} hashjoin[Ort_o, Ort_p, 99997]
  count
```

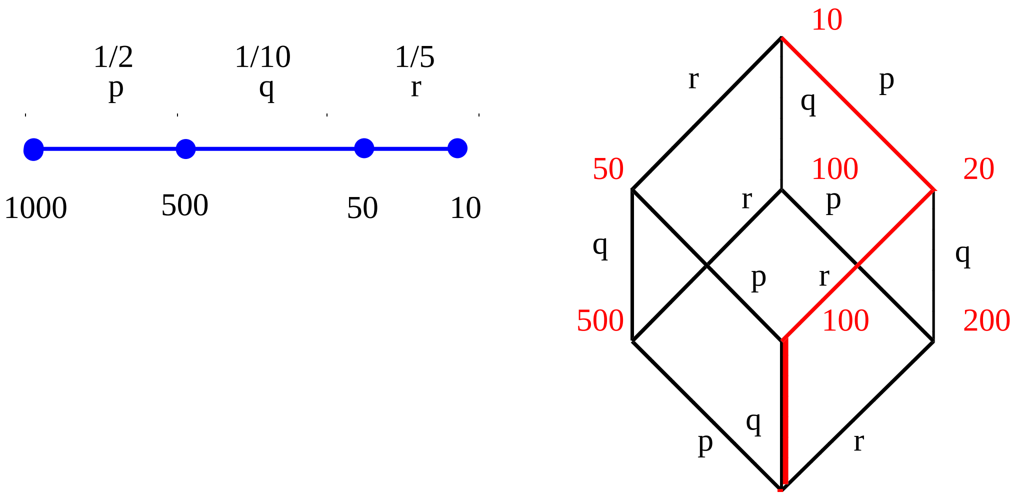
Secondo - Optimizer

- Optimizer:
    - core capability: conjunctive query optimization
    - currently supports a relational model with an SQL-like language
    - written in PROLOG

Secondo - Optimizer

Performs conjunctive query optimization: given a set of relations and a set of selection or join predicates, find a good plan. Uses a new algorithm for this.

Based on **shortest path search** in a **predicate order graph**.



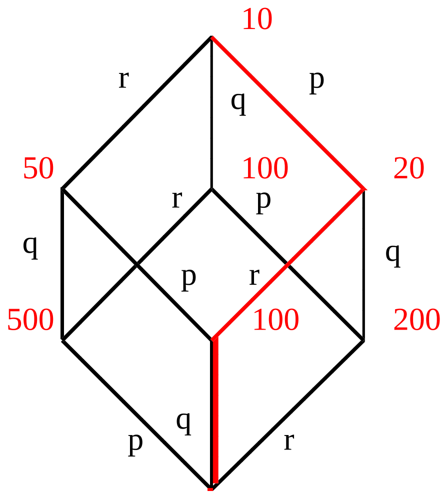
Selectivities of predicates are determined in advance by evaluating selections and joins on small sample relations. Selectivities once determined are stored for future use.

Optimizer implements an SQL-like language in a notation adapted to PROLOG.

Secondo - Optimizer

Optimization algorithm:

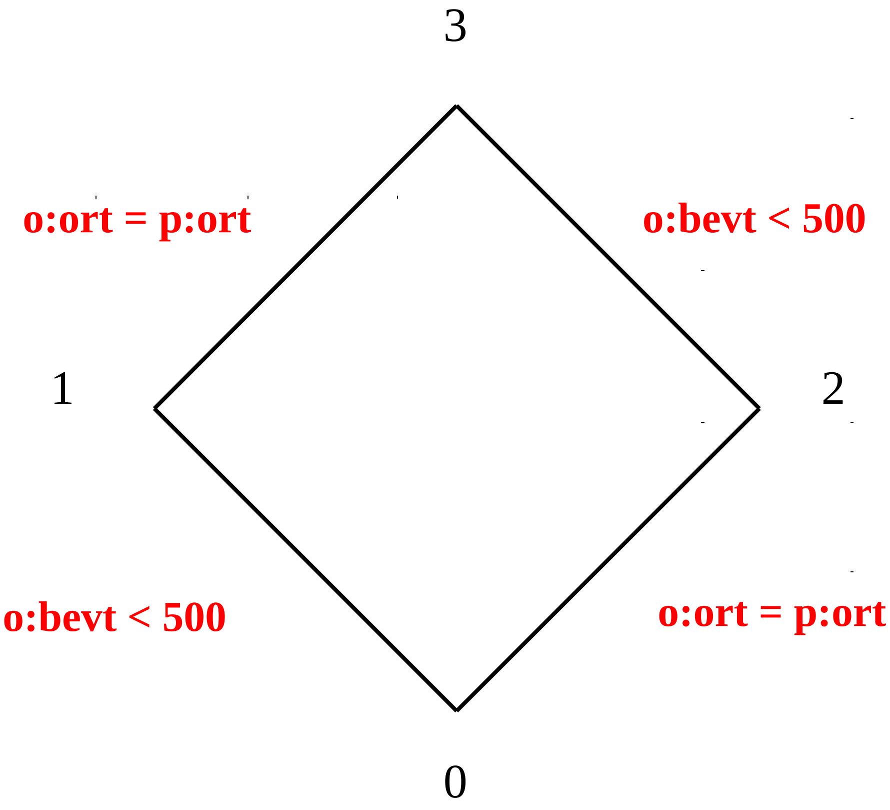
1. For given relations and predicates, construct the POG.
2. For each edge, construct plan edges. Controlled by optimization rules for selections and joins.
3. For given sizes of arguments and selectivities of predicates, assign sizes to all nodes (intermediate results). Also annotate edges of POG with selectivities.
4. For each plan edge, compute its cost. Based on sizes of arguments, selectivity along the edge, and cost function for each operator occurring in a plan edge.
5. Use algorithm of Dijkstra to find a shortest path from bottom to top node through the graph of plan edges. This is the plan.



Secondo - Optimizer

Example:

```
sql select count(*) from [orte as o, plz as p]
    where [o:bevt < 500, o:ort = p:ort]
```



Secondo - Optimizer

Demo: The Optimizer

```
open 'database opt'.    [updateRel(orte), updateRel(plz), assert(noProgress).]

sql select count(*) from [orte as o, plz as p] where [o:bevt < 500, o:ort =
    p:ort]
```

Step 1: construct the POG

```
writeNodes.
writeEdges.
```

Step 2: construct plan edges

```
writePlanEdges.
```

Step 3: assign sizes

```
writeSizes.
```

Step 4: compute cost edges

```
writeCostEdges
```

Step 5: compute shortest path

```
dijkstra(0, 3, Path, Cost), plan(Path, Plan), plan_to_atom(Plan, Query).
```

Secondo - Optimizer

Demo: The Optimizer

A more complex query:

```
sql select count(*)
from [orte as o, plz as p1, plz as p2, plz as p3]
where [
   o:ort = p1:ort,
   p2:plz = p1:plz + 7,
   (p2:plz mod 5) = 0,
   p2:plz > 30000,
   o:ort contains "o",
   o:bevt > 200,
   o:bevt < 700,
   p3:plz = p2:plz + 40]

writeSizes.
```

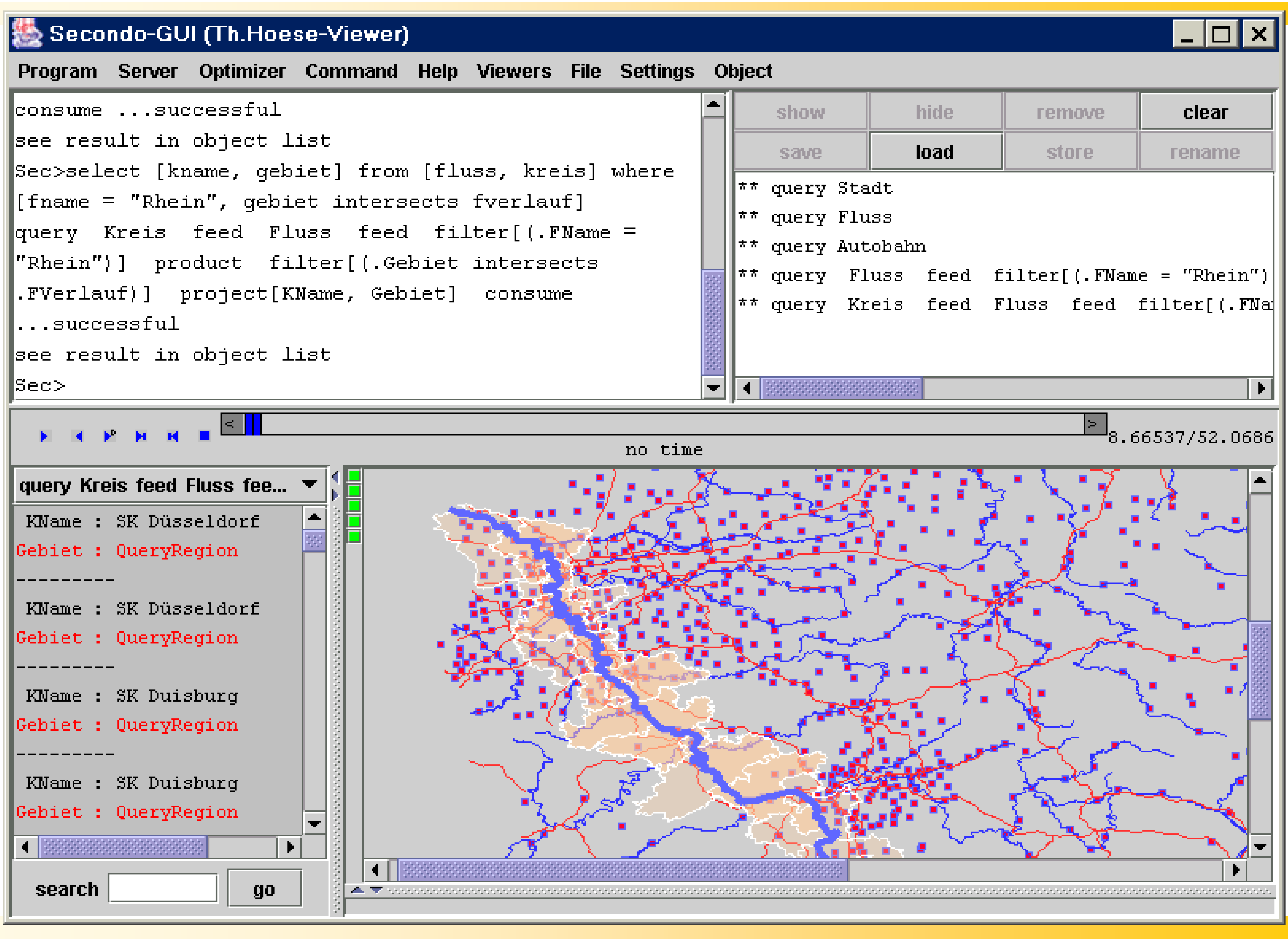
Secondo - Optimizer

Some Interesting Properties

- Simple concept, easy to understand, relatively easy to implement
- Guaranteed to find the optimal plan
  [among available plans, assuming correct cost functions, selectivity estimates, attribute independence]
- Exponential complexity, POG has 2^n nodes, n * 2^{n-1} edges. Works fine for up to about 10 predicates (less than a second).
- If an efficient plan exists, Dijkstra explores only a small part of the POG
- A variant builds only the part of the POG that is explored by Dijkstra.
- Deals with expensive predicates (important for non-standard applications such as moving objects)
- Selectivity estimation by sampling works automatically as soon as a new operation is implemented (histograms not feasible)
- Sample queries for selection predicates cheap
- Sample queries for join predicates more expensive, but exhausted after a while
- Easy to write optimization rules in PROLOG

Secondo - Graphical User Interface

– GUI:
  - extensible interface for an extensible DBMS like Secondo
  - extensible by viewers
  - sophisticated spatial / spatio-temporal viewer, extensible by data types
  - written in Java

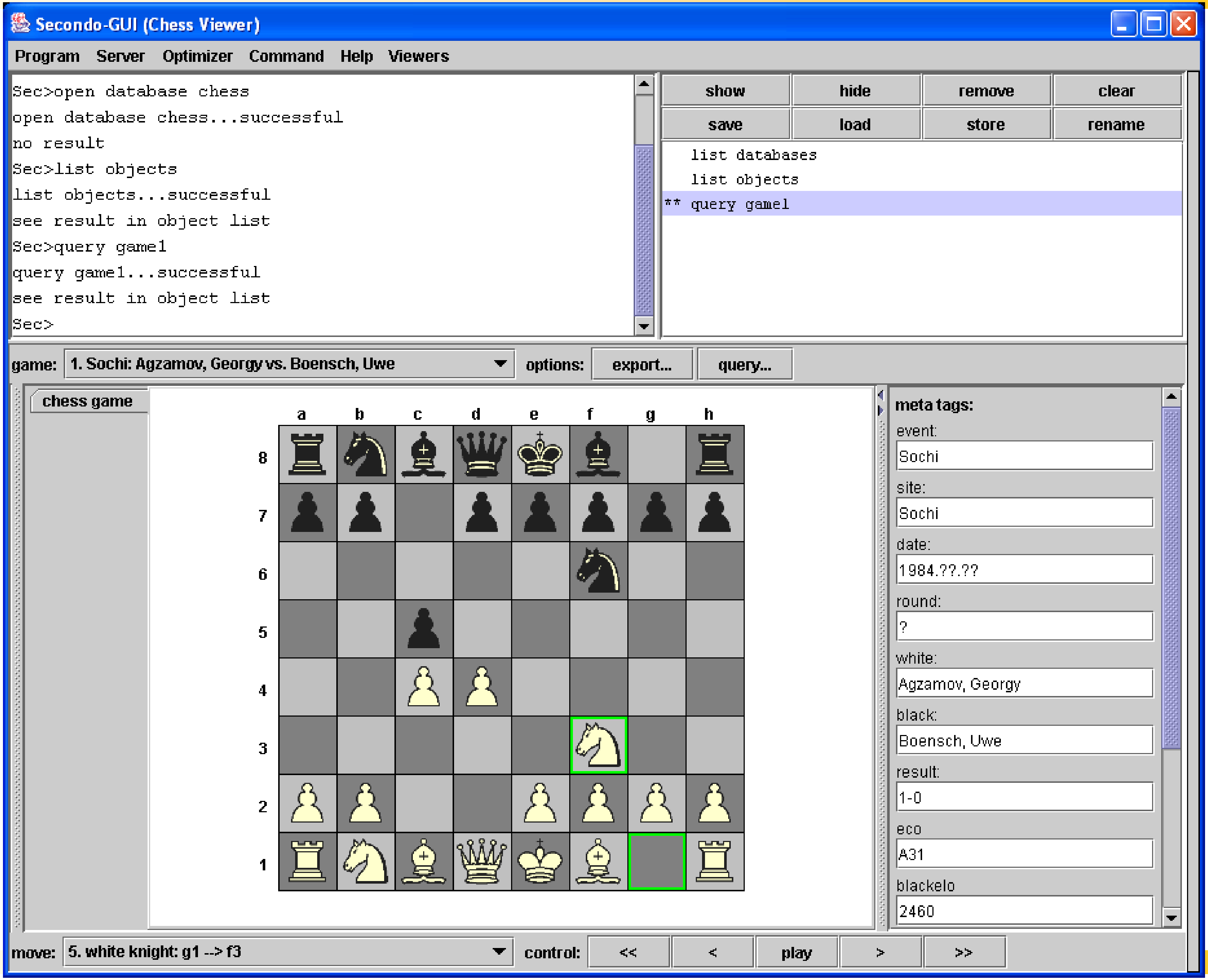


```
consume ...successful
see result in object list
Sec>select [kname, gebiet] from [fluss, kreis] where
[fname = "Rhein", gebiet intersects fverlauf]
query  Kreis  feed  Fluss  feed  filter[(.FName =
"Rhein")]  product  filter[(.Gebiet intersects
.FVerlauf)]  project[KName, Gebiet]  consume
...successful
see result in object list
Sec>
```

```
** query Stadt
** query Fluss
** query Autobahn
** query  Fluss  feed  filter[(.FName = "Rhein")
** query  Kreis  feed  Fluss  feed  filter[(.FNa
```

```
8.66537/52.0686
no time
```

```
query Kreis feed Fluss fee...
 KName : SK Düsseldorf
Gebiet : QueryRegion
---------
 KName : SK Düsseldorf
Gebiet : QueryRegion
---------
 KName : SK Duisburg
Gebiet : QueryRegion
---------
 KName : SK Duisburg
Gebiet : QueryRegion
```



```
Sec>open database chess
open database chess...successful
no result
Sec>list objects
list objects...successful
see result in object list
Sec>query game1
query game1...successful
see result in object list
Sec>
```

```
   list databases
   list objects
** query game1
```

game: 1. Sochi: Agzamov, Georgy vs. Boensch, Uwe

meta tags:

event: Sochi

site: Sochi

date: 1984.??.??

round: ?

white: Agzamov, Georgy

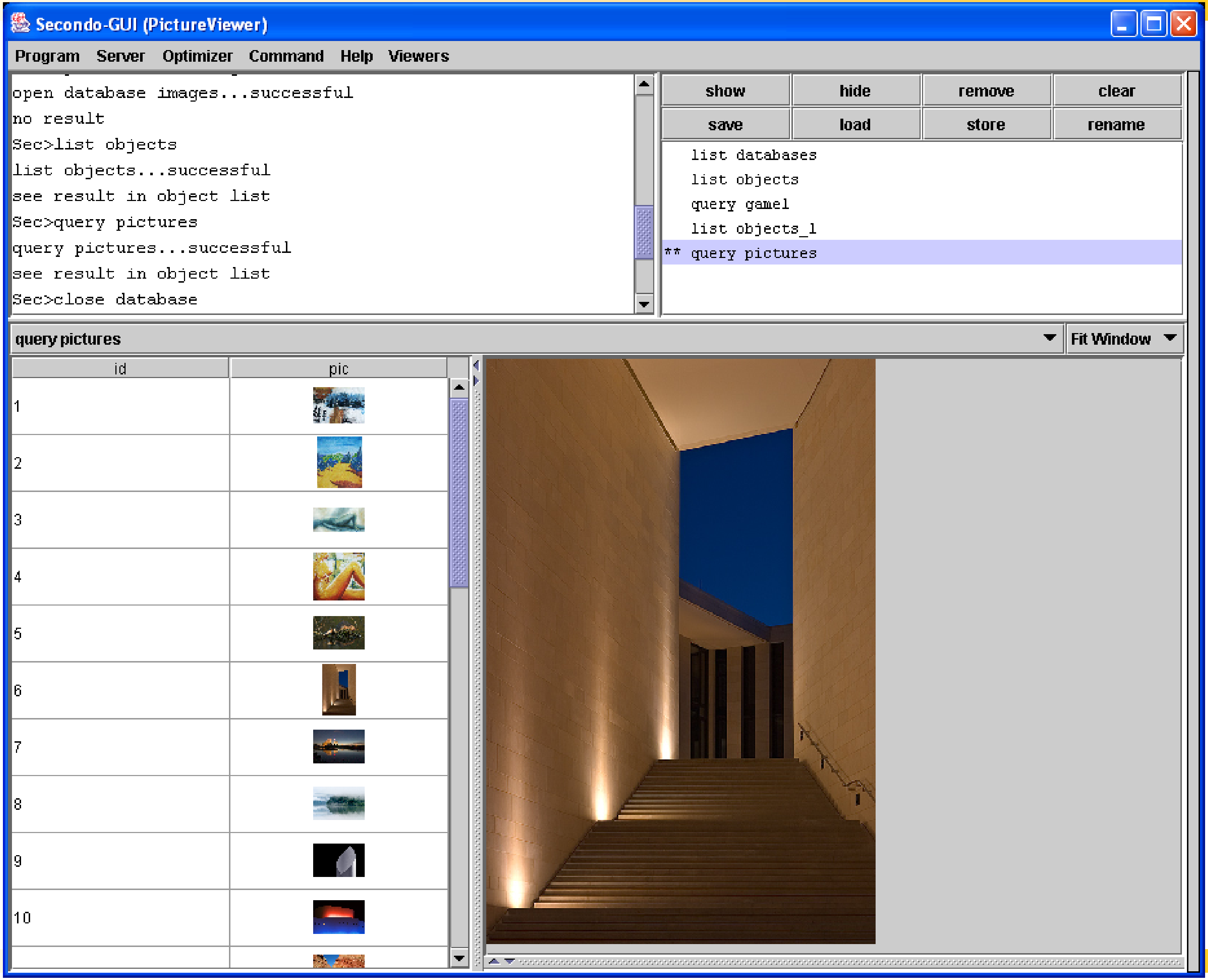
black: Boensch, Uwe

result: 1-0

eco: A31

blackelo: 2460

move: 5. white knight: g1 --> f3



```
open database images...successful
no result
Sec>list objects
list objects...successful
see result in object list
Sec>query pictures
query pictures...successful
see result in object list
Sec>close database
```

```
   list databases
   list objects
   query gamel
   list objects_l
** query pictures
```

query pictures

id	pic
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Secondo - Graphical User Interface

Demo: The GUI

- after part on moving objects

A Moving Objects DBMS Prototype

Moving Objects Databases

Database support for the modeling and querying of time-dependent geometries & moving objects.

Physical objects have position and extent which may change over time, e.g.

- countries, rivers, roads, pollution areas, land parcels, ...
- taxis, air planes, oil tankers, criminals, polar bears, hurricanes, flood areas, oil spills
- users with location-aware portable wireless networked devices such as mobile phones, PDAs, ...

Spatio-temporal databases - roots in spatial and in temporal databases.

New: support for continuously changing geometries = movement.

Consider moving point objects and regions that may move and change their shape. Also time-dependent linear features (but less frequent and relevant).

Current and near future movement / history of movement.

Moving Objects Databases

Applications

- Logistics, fleet management
- traffic analysis and management
- analysis of movements of people (customers) in mobile computing
- earth sciences
- environmental studies
- biology (e.g. animal behaviour, tracking)
- meteorology
- geographic information systems
    - any kind of temporal development of spatial data
    - land ownership
- history

Moving Objects Databases

Goal:

- any kind of moving entity can be represented in a database
- powerful query languages available to formulate any kind of questions about such movements

Two perspectives:

- location management
- spatio-temporal data

Three approaches:

Location management:

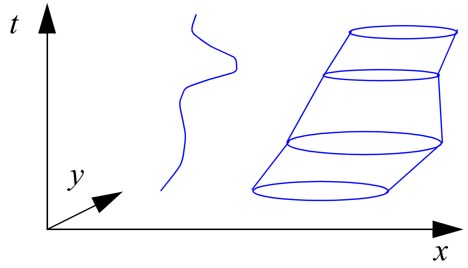
- Wolfson et al.

Spatio-temporal data:

- spatio-temporal data types
- constraint approach

Spatio-Temporal Data Types

- Extend the strategy used in spatial databases to offer abstract data types with suitable operations.
- Offer *spatio-temporal data types* such as *moving point* (*mpoint*) and moving region (*mregion*)
- Values of such types are functions from time into the domains, e.g.
    - *f*: *instant* → *point* (value of *mpoint*)
    - *f*: *instant* → *region* (value of *mregion*)



Spatio-Temporal Data Types

```
flight (id: string, from: string, to: string, route: mpoint)
weather (id: string, kind: string, area: mregion)
```

The data types include suitable operations such as:

intersection: *mpoint* × *mregion* → *mpoint*

distance: *mpoint* × *mpoint* → *mreal*

trajectory: *mpoint* → *line*

deftime: *mpoint* → *periods*

length: *line* → *real*

min: *mreal* → *real*

Spatio-Temporal Data Types

Some Example Queries

Query 1: “Find all flights from Düsseldorf that are longer than 5000 kms.”

```
SELECT id
FROM flights
WHERE from = 'DUS' AND length(trajectory(route)) > 5000

mpoint     -> line       trajectory
line       -> real       length
```

Query 2: “Retrieve any pairs of air planes that during their flight came closer to each other than 500 meters!”

```
SELECT f.id, g.id
FROM flights AS f, flights AS g
WHERE f.id <> g.id AND min(distance(f.route, g.route)) < 0.5

mpoint x mpoint  -> mreal    distance
mreal            -> real     min
```

Spatio-Temporal Data Types

Some Example Queries

Query 3: “At what times was flight BA488 within the snow storm with id S16?”

```
SELECT deftime(intersection(f.route, w.area))
FROM flights AS f, weather AS w
WHERE f.id = 'BA488' AND w.id = 'S16'

mpoint x mregion       -> mpoint   intersection
mpoint                 -> periods  deftime
```

An Algebra for Moving Objects

Design Goals**:** Design a system of data types & operations which is

- closed
- simple
- powerful

Closed**:** under application of type constructors, in particular:

- For all base types of interest, we have corresponding time-dependent (temporal, “moving”) types.
- For all temporal types, we have types to represent their *domain* and *range* projections.

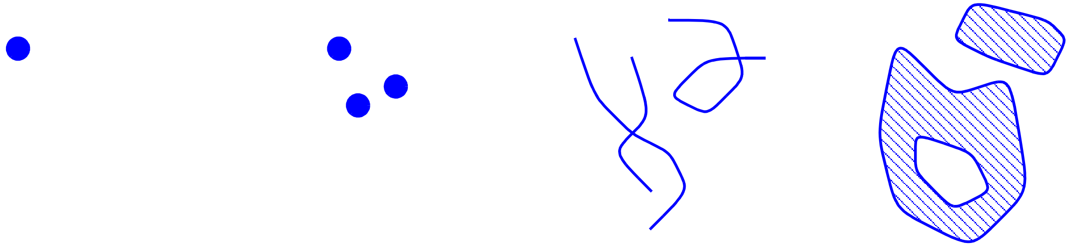
Simple:

- There are lots of types around, avoid proliferation of operations → use generic operations as much as possible.
- Explore the space of possible operations systematically.
- Achieve consistency of operations on base and temporal types.

Powerful**:** more or less a result of the previous two.

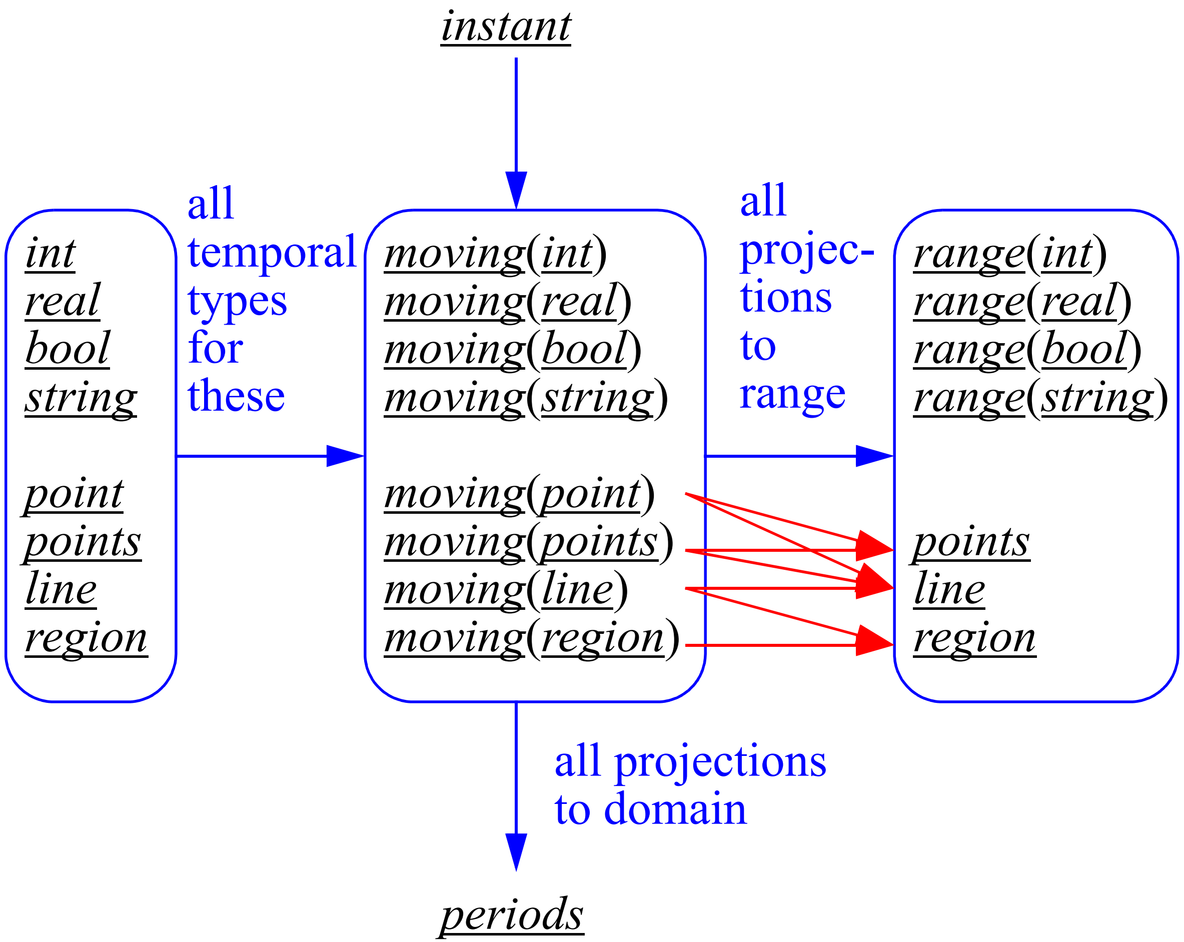
Data Types

Spatial Types



point *points* *line* *region*

Data Types



instant

int
real
bool
string

point
points
line
region

all temporal types for these

moving(*int*)
moving(*real*)
moving(*bool*)
moving(*string*)

moving(*point*)
moving(*points*)
moving(*line*)
moving(*region*)

all projections to range

range(*int*)
range(*real*)
range(*bool*)
range(*string*)

points
line
region

all projections to domain

periods

Data Types

Type System as a Signature

Type constructor	*Signature*	
int, *real*, *string*, *bool*		→ BASE
point, *points*, *line*, *region*		→ SPATIAL
instant		→ TIME
range	BASE ∪ TIME	→ RANGE
moving, *intime*	BASE ∪ SPATIAL	→ TEMPORAL

Terms of the signature are types, for example

int, *region*, *range*(*instant*), *moving*(*point*), *intime*(*bool*)

Operations: Overview

Design Goals:

- as generic as possible
- achieve consistency between operations on non-temporal and temporal types

e.g.

point × *point*	→ *real*	**distance**
mpoint × *mpoint*	→ *mreal*	**mdistance**

$$\text{distance}(\text{atinstant}(mp_1, t), \text{atinstant}(mp_2, t))$$

$$= \quad \text{atinstant}(\text{mdistance}(mp_1, mp_2), t)$$

Three steps:

1. Define operations on non-temporal types
2. “Lift” them all to temporal types
3. Add specific operations for temporal types

Operations on Non-Temporal Types

Generic view: *point* and *point set* in some space

1D Spaces

2D Space

Space	point type π	point set type σ
Integer	*int*	*range*(*int*)
Real	*real*	*range*(*real*)
Bool	*bool*	*range*(*bool*)
String	*string*	*range*(*string*)
Time	*instant*	*periods*
2D	*point*	*points*, *line*, *region*

π × σ → *bool* **inside** instantiates to

int × *range*(*int*) → *bool* **inside**

bool × *range*(*bool*) → *bool*

instant × *periods* → *bool*

point × *line* → *bool* etc.

Operations on Non-Temporal Types …

Class	Operations
Predicates	**isempty** =, /=, **intersects**, **inside** <, <=, >=, >, **before** **touches**, **attached**, **overlaps**, **on_border**, **in_interior**
Set Operations	**intersection**, **union**, **minus** **crossings**, **touch_points**, **common_border**
Aggregation	**min**, **max**, **avg**, **center**, **single**
Numeric	**no_components**, **size**, **perimeter**, **duration**, **length**, **area**
Distance and Direction	**distance**, **direction**
Base Type Specific	**and**, **or**, **not**

Lifting Operations to Time-Dependent Operations

$$\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \quad \rightarrow \beta \qquad \textbf{op}$$

Lifting yields operations

$$\underline{moving}(\alpha_1) \times \alpha_2 \times \ldots \times \alpha_n \quad \rightarrow \underline{moving}(\beta) \ \ \textbf{op}$$
$$\alpha_1 \times \underline{moving}(\alpha_2) \times \ldots \times \alpha_n \quad \rightarrow \underline{moving}(\beta) \ \ \textbf{op}$$
$$\underline{moving}(\alpha_1) \times \underline{moving}(\alpha_2) \times \ldots \times \alpha_n \quad \rightarrow \underline{moving}(\beta) \ \ \textbf{op}$$

…

$$\underline{moving}(\alpha_1) \times \ldots \times \underline{moving}(\alpha_n) \quad \rightarrow \underline{moving}(\beta) \ \ \textbf{op}$$

For example:

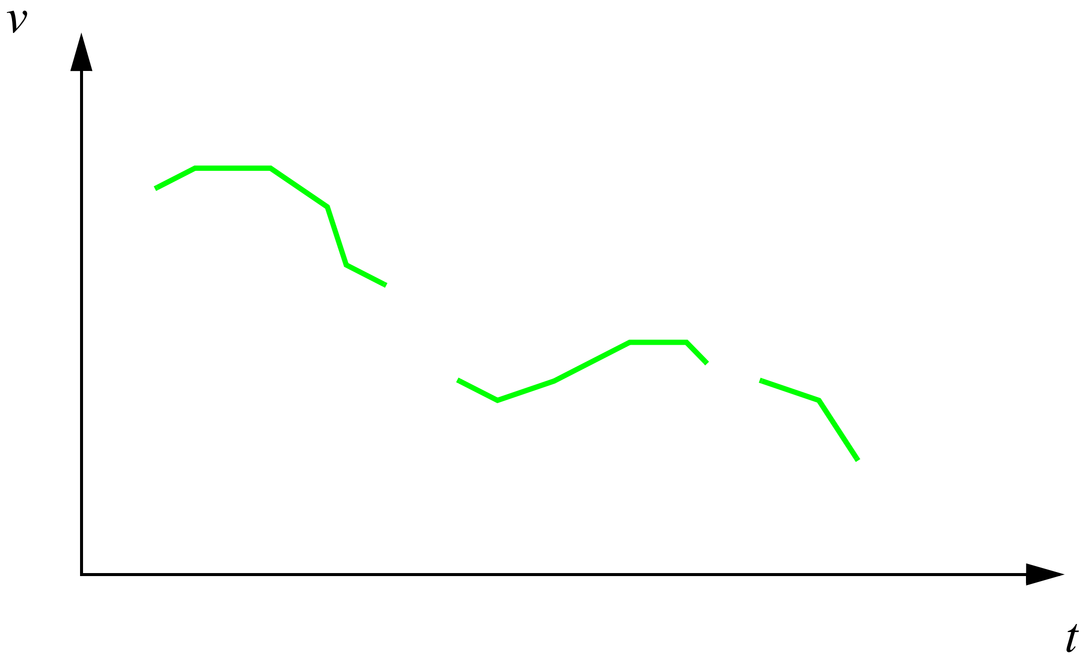
$\underline{real} \times \underline{real}$	$\rightarrow \underline{bool}$	=	$\underline{region} \times \underline{point}$	$\rightarrow \underline{point}$	**intersection**
$\underline{mreal} \times \underline{real}$	$\rightarrow \underline{mbool}$	=	$\underline{mregion} \times \underline{point}$	$\rightarrow \underline{mpoint}$	**intersection**
$\underline{real} \times \underline{mreal}$	$\rightarrow \underline{mbool}$	=	$\underline{region} \times \underline{mpoint}$	$\rightarrow \underline{mpoint}$	**intersection**
$\underline{mreal} \times \underline{mreal}$	$\rightarrow \underline{mbool}$	=	$\underline{mregion} \times \underline{mpoint}$	$\rightarrow \underline{mpoint}$	**intersection**

Operations on Temporal Types

Values of these types are partial functions

$f\colon A_{\underline{instant}} \to A_{\alpha}$

Projection to Domain / Range



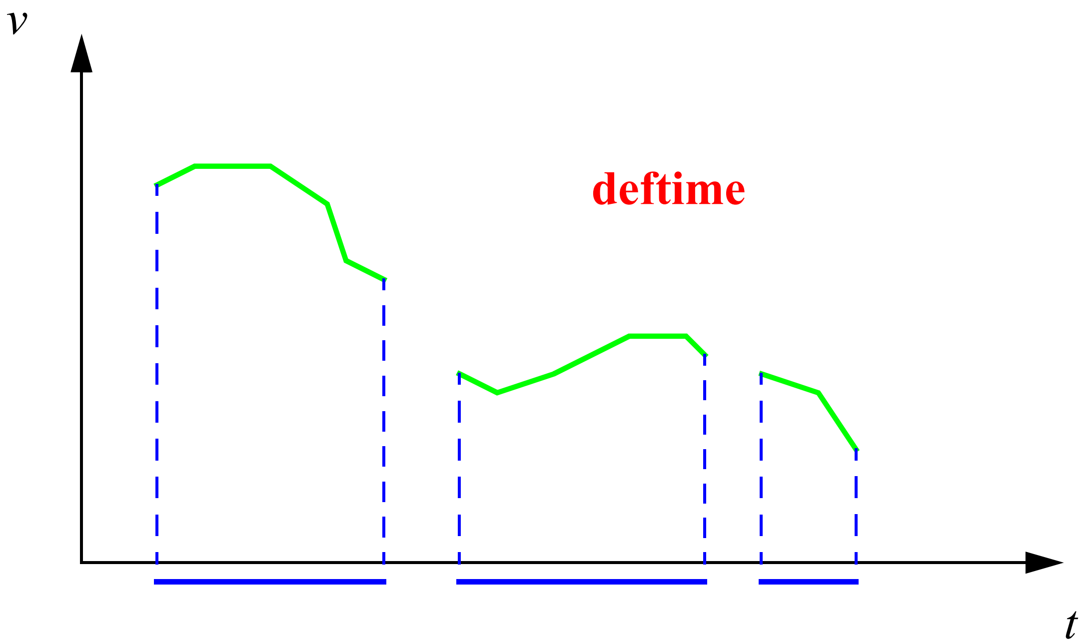
moving(α)	→ *periods*	**deftime**
moving(α)	→ *range*(α)	**rangevalues** [1D]
moving(*point*)	→ *points*	**locations**
moving(*point*s)	→ *points*	
moving(*point*)	→ *line*	**trajectory**
moving(*point*s)	→ *line*	
moving(*line*)	→ *line*	**routes**
moving(*line*)	→ *region*	**traversed**
moving(*region*)	→ *region*	

Operations on Temporal Types

Values of these types are partial functions

$f\colon A_{\underline{instant}} \rightarrow A_{\alpha}$

Projection to Domain / Range



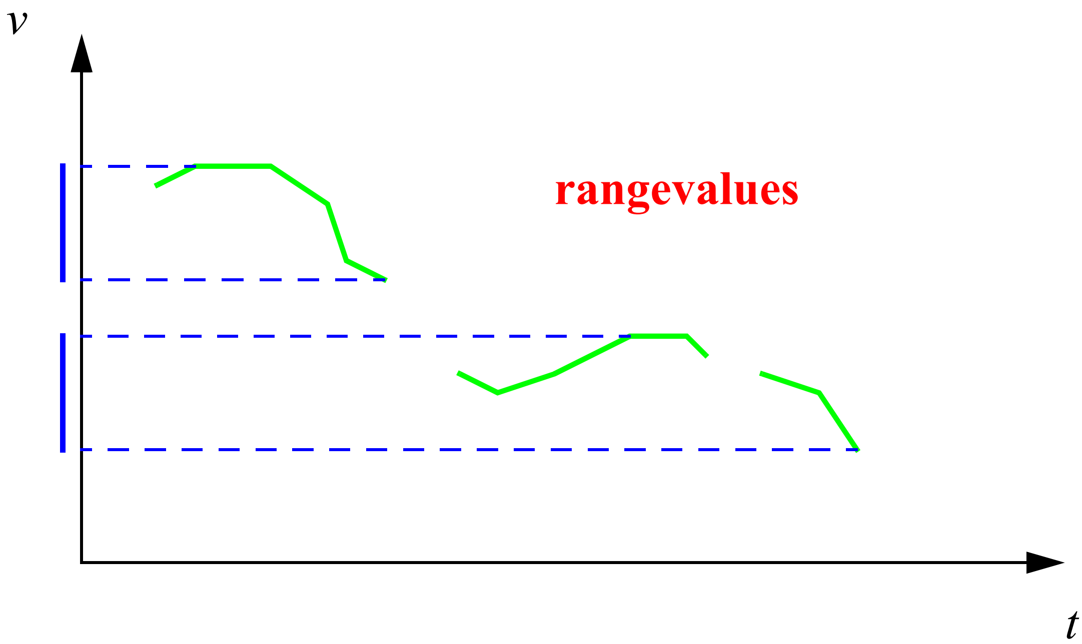
moving(α)	→ *periods*	**deftime**
moving(α)	→ *range*(α)	**rangevalues** [1D]
moving(*point*)	→ *points*	**locations**
moving(*point*s)	→ *points*	
moving(*point*)	→ *line*	**trajectory**
moving(*point*s)	→ *line*	
moving(*line*)	→ *line*	**routes**
moving(*line*)	→ *region*	**traversed**
moving(*region*)	→ *region*	

Operations on Temporal Types

Values of these types are partial functions

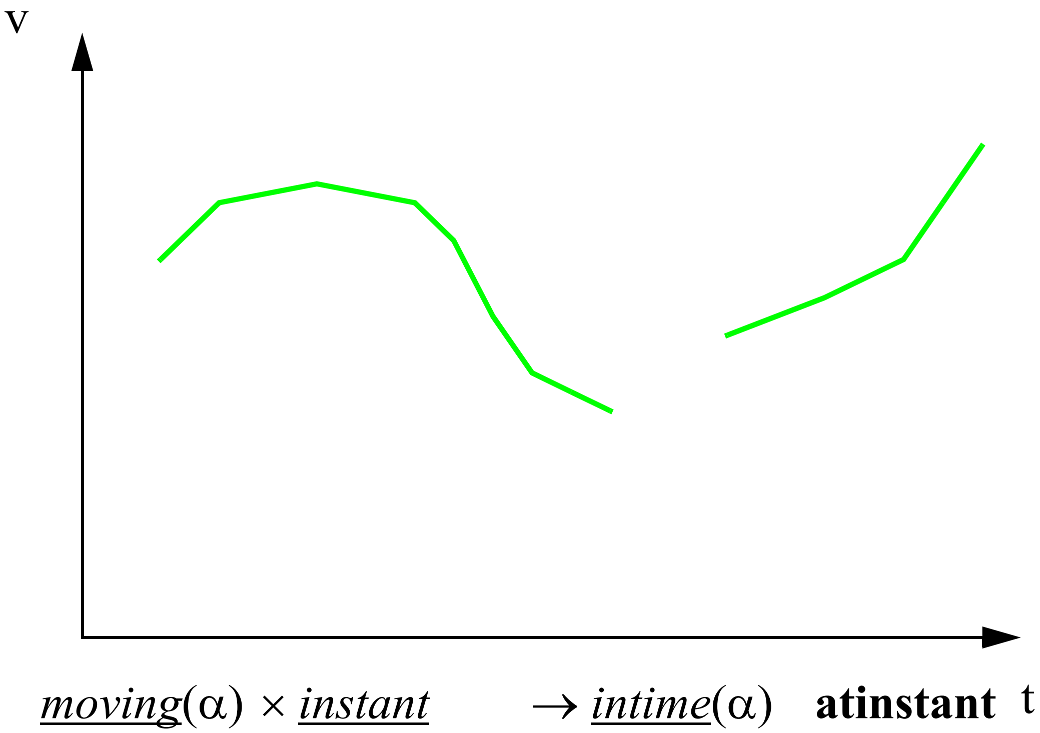
$f: A_{\underline{instant}} \to A_{\alpha}$

Projection to Domain / Range



moving(α)	→ *periods*	**deftime**
moving(α)	→ *range*(α)	**rangevalues** [1D]
moving(*point*)	→ *points*	**locations**
moving(*point*s)	→ *points*	
moving(*point*)	→ *line*	**trajectory**
moving(*point*s)	→ *line*	
moving(*line*)	→ *line*	**routes**
moving(*line*)	→ *region*	**traversed**
moving(*region*)	→ *region*	

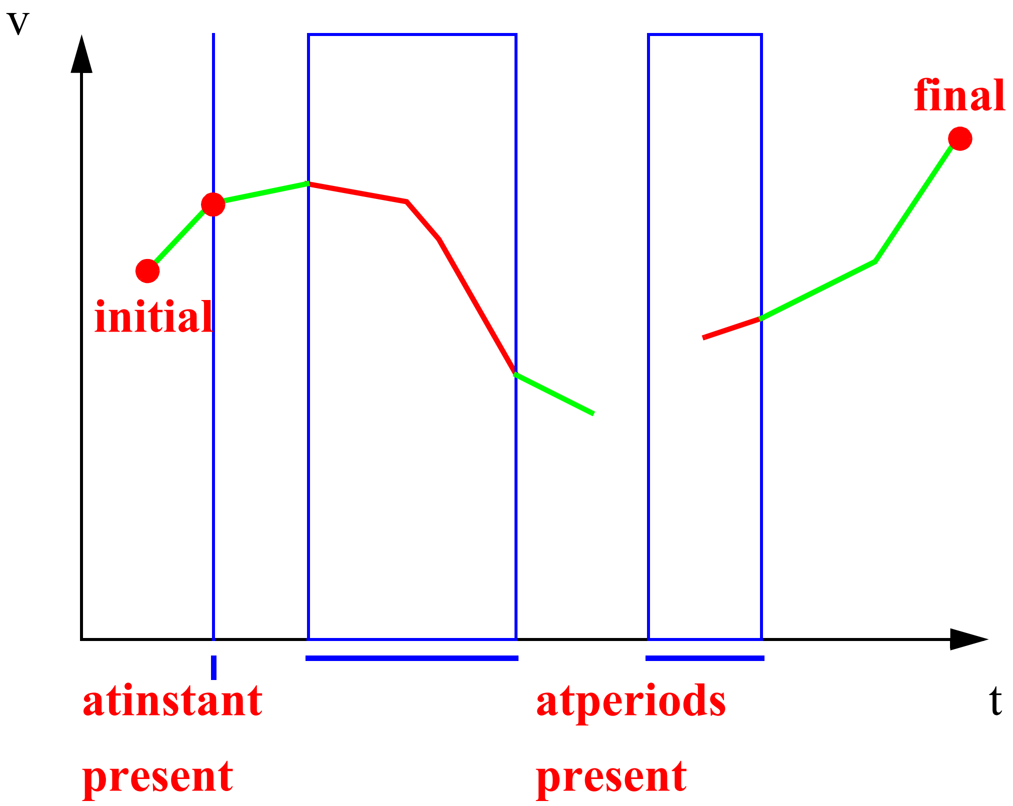
Interaction with Points and Point Sets in Domain and Range



moving(α) × *instant*	→ *intime*(α)	**atinstant**
moving(α) × *periods*	→ *moving*(α)	**atperiods**
moving(α)	→ *intime*(α)	**initial**, **final**
moving(α)× *instant*	→ *bool*	**present**
moving(α)× *periods*	→ *bool*	

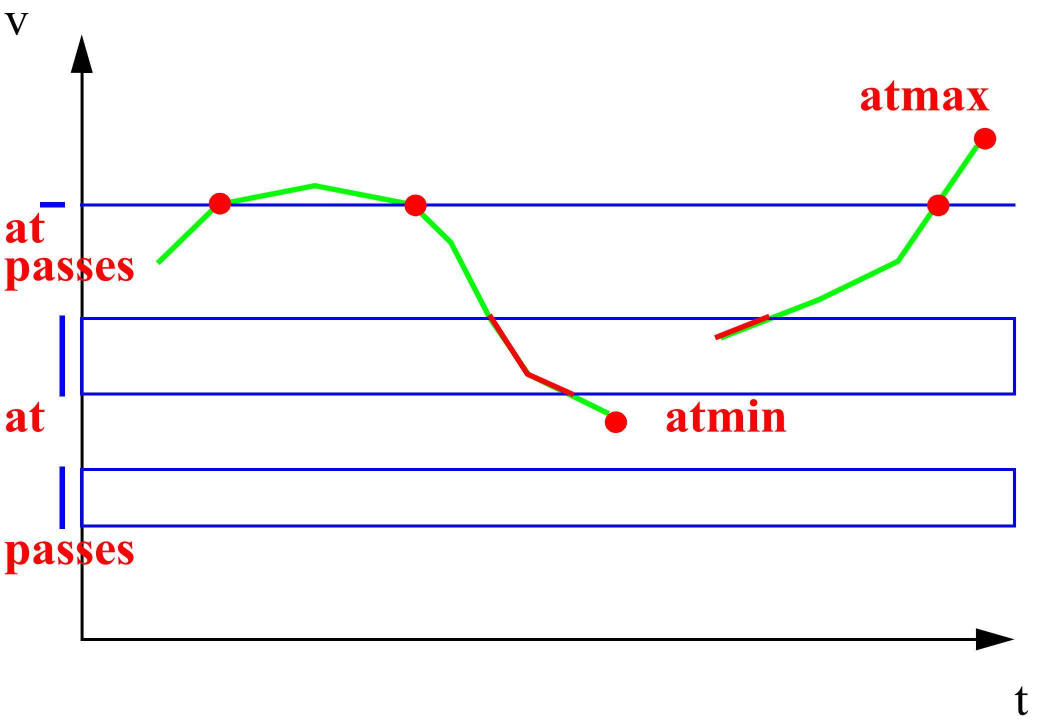
moving(α) × β	→ *moving*(α)	**at** [1D]
moving(α) × β	→ *moving*(min(α, β))	**at** [2D]
moving(α)	→ *moving*(α)	**atmin**, **atmax** [1D]
moving(α) × β	→ *bool*	**passes**

Interaction with Points and Point Sets in Domain and Range



moving(α) × *instant*	→ *intime*(α)	**atinstant**
moving(α) × *periods*	→ *moving*(α)	**atperiods**
moving(α)	→ *intime*(α)	**initial, final**
moving(α)× *instant*	→ *bool*	**present**
moving(α)× *periods*	→ *bool*	

Interaction with Points and Point Sets in Domain and Range



<u>*moving*</u>(α) × β	→	<u>*moving*</u>(α)	**at** [1D]
<u>*moving*</u>(α) × β	→	<u>*moving*</u>(min(α, β))	**at** [2D]
<u>*moving*</u>(α)	→	<u>*moving*</u>(α)	**atmin**, **atmax** [1D]
<u>*moving*</u>(α) × β	→	<u>*bool*</u>	**passes**

Rate of Change

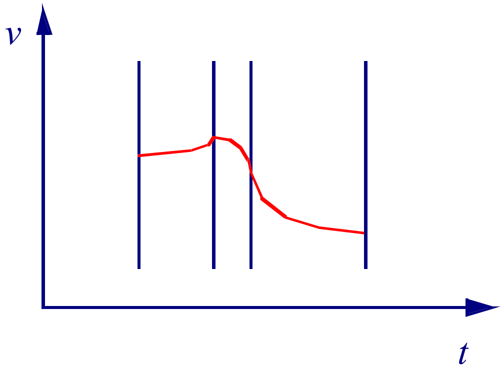
Concept of derivative: $f'(t) = \lim_{\Delta t \to 0} \frac{f(t+\Delta t) - f(t)}{\Delta t}$

Applicable to which data types? Needed: difference, division by real number..

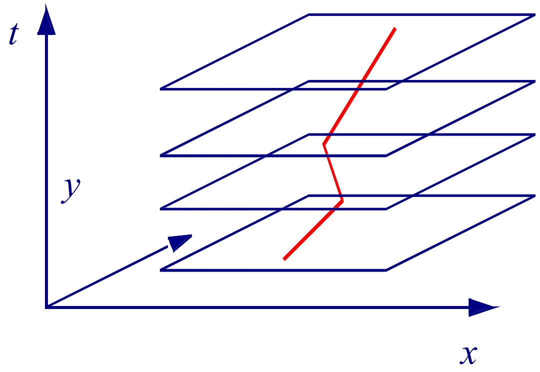
Operation	*Signature*	*Semantics*
derivative	$\underline{mreal} \to \underline{mreal}$	μ' where $\mu'(t) = \lim_{\delta \to 0}(\mu(t+\delta) - \mu(t)) / \delta$
speed	$\underline{mpoint} \to \underline{mreal}$	μ' where $\mu'(t) = \lim_{\delta \to 0} f_{\textbf{distance}}(\mu(t+\delta), \mu(t)) / \delta$
mdirection	$\underline{mpoint} \to \underline{mreal}$	μ' where $\mu'(t) = \lim_{\delta \to 0} f_{\textbf{direction}}(\mu(t+\delta), \mu(t)) / \delta$
turn	$\underline{mpoint} \to \underline{mreal}$	μ' where $\mu'(t)$ $= \lim_{\delta \to 0}(f_{\textbf{mdirection}}(\mu(t+\delta)) - f_{\textbf{mdirection}}(\mu(t))) / \delta$
velocity	$\underline{mpoint} \to \underline{mpoint}$	μ' where $\mu'(t) = \lim_{\delta \to 0}(\mu(t+\delta) - \mu(t)) / \delta$

Discrete Model / Data Structures

Representation of types *moving*(α): Represent the temporal development of the value of type a by decomposing the time dimension into a set of disjoint time intervals (“slices”) such that within each slice the development can be described by some “simple” function. Called the **sliced representation**.

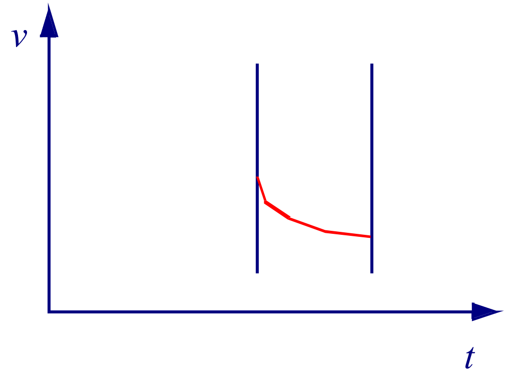


moving(real)



moving(point)

Real Units



$$D_{\underline{ureal}} = Interval(Instant) \times$$

$$\{(a, b, c, r) \mid a, b, c \in \text{real}, r \in \text{bool}\}$$

Semantics (value at instant *t*):

$$\iota((a,b,c,r),t) = \begin{cases} at^2 + bt + c & \text{if } \neg r \\ \sqrt{at^2 + bt + c} & \text{if } r \end{cases}$$

Distance between two moving points:

$$\sqrt{at^2 + bt + c}$$

Perimeter of a moving region:

$$bt + c$$

Area of a moving region:

$$at^2 + bt + c$$

Not closed under derivative!

A Moving Objects DBMS Prototype

Demo: GUI and Moving Objects

```
[start Javagui]

open database berlintest
query UBahn
query train7
query trajectory(train7)
query deftime(train7)
query train7 atinstant six30
query val(train7 atinstant sixthirty)
query theminute(2003,11,20,6,25)
query train7 atperiods
  theperiod(theminute(2003,11,20,6,25), theminute(2003,11,20,6,39)
query thecenter
query train7 at thecenter
query deftime(train7 at thecenter)
```

A Moving Objects DBMS Prototype

Demo: GUI and Moving Objects

```
query Trains count
query Trains feed filter[.Trip present eight] consume

[start optimizer server]

query mehringdamm
select * from trains
  where [trip present eight, trip passes mehringdamm]

select [val(trip atinstant eight) as ateight] from trains
  where [trip present eight, trip passes mehringdamm]
```

A Moving Objects DBMS Prototype

Example: Create a time dependent density animation

```
observe x-extension 30000, y-extension 20000
create a 6 x 4 raster of squares of size 5000
lower left corner at (-4000, 1000)

let r1 = [const rect value (-4000.0 1000.0 1000.0 6000.0)]
  rect2region
query seqinit(1)

let raster = r1 feed transformstream
  ten feed filter[.no < 7] {t1}
  ten feed filter[.no < 5] {t2}
  product product
  projectextend[; No: seqnext(),
    Field: .elem translate[(.no_t1 - 1) * 5000.0,
      (.no_t2 - 1) * 5000.0] ]
  consume
```

A Moving Objects DBMS Prototype

Example: Create a time dependent density animation

```
query Trains feed filter[.Line < 5]
  raster feed
  symmjoin[.Trip passes ..Field]
  sortby[No asc]
  groupby[No;
    Field: group feed extract[Field],
    Occupation: group feed
      extend[Time: periods2mint(deftime(.Trip at .Field))]
      aggregate[Time; fun(m1:mint, m2:mint) m1 + m2; zero()]
  ]
consume
```

Highlights

- A generic implementation model based on type constructors and operators.
- A type-checked, comfortable language for writing query plans, extensible by operators.
- A module concept for extensibility: algebra modules
- A strictly modular architecture with clean interfaces separating kernel, optimizer and GUI
- An optimizer capable to deal with complex systems of types and operations (expensive predicates, selectivity estimation, extensibility)
- A flexible language for querying moving objects beyond just a few query types (range queries, nearest neighbor, …)