A Short Guide to Writing Executable Queries in SECONDO

Ralf Hartmut Güting, April 2011

SECONDO was built by the *SECONDO team*.

Contents

1 Introduction

Queries can be formulated to SECONDO at two levels:

1. in an SQL-like language
2. by typing query plans directly, called the executable language.

Queries at level 1 are given to the optimizer which produces a plan written in the executable language of level 2. A unique feature of SECONDO is that the user can also type queries at level 2. A query at level 2 is essentially a term composed of database objects and operators of the active algebras in the system.

Writing queries at the executable level has disadvantages and advantages. Disadvantages are:

- This language is more complex than SQL. One has to understand what methods are available, e.g. for executing joins.
- The user decides how to efficiently execute the query, hence there is no cost estimation involved and the user's decision may be wrong.

Advantages are:

- The full range of the system's execution capabilities is available. The optimizer is only able to create a subset of possible plans and is restricted to a relational data model (including abstract data types at the attribute level such as moving objects). At the executable level also other data types than relations such as networks, graphs, nested relations, arrays of relations as well as parallel execution models are available.
- In research, adding new concepts to a database system proceeds from the executable level to the query optimization level. That is, the first step is to implement for example a new kind of index structure as a data type and to implement a new query processing algorithm as an operator. Once this is done, these methods are available in the executable language. It is a further step to integrate such methods into query optimization by extending query plan generation, cost estimation and possibly selectivity estimation techniques. Because of this, capabilities available at the SQL level will always lag behind those at the executable level.

The purpose of this document is to provide a brief introduction to the query processing operations available at the executable level.

2 Example Database

We use the *berlintest* database available within the SECONDO distribution and within it the following objects:

- relations

```
Trains(Id: int, Line: int, Up: bool, Trip: mpoint)
strassen(Name: string, Typ: string, geoData: line)
Kinos(Name: string, Strasse: string, geoData: point)
```

  Trains is a relation describing trips of underground trains in Berlin, modeling the movement in data type *mpoint* (short for *moving*(*point*)). *strassen* means roads, a relation describing the road network of Berlin. *Kinos* is a relation with cinema locations.

- an R-tree index on the *geoData* attribute of *strassen*:

```
strassen_geoData_rtree
```

- atomic objects

```
train7: mpoint
mehringdamm: point
tiergarten: region
```

3 Creating a Stream of Tuples

3.1 From a Relation

A relation can be read from disk and made available as a stream of tuples by the operations

- *feed*
- *feedproject*

Feed puts tuples into the stream as they are. *Feedproject* creates only the attributes given in a projection list, hence is more efficient for large tuples.

Examples:

```
Trains feed
Trains feedproject[Id, Trip]
```

These operations have signatures:

```
feed:         rel(Tuple) -> stream(Tuple)                        _ #
feedproject:  rel(Tuple) x AttrList -> stream(tuple(AttrList))   _ # [ _ ]
```

Here words starting with a capital letter represent type variables, in contrast to type constructors starting with a lower case letter. At the end of the line is a syntax pattern where "_" represents an argument and "#" the operator. Parentheses, square brackets, commas, etc. have to be put as shown in the pattern. For stream operations usually postfix syntax is used, hence *feed* is applied to the argument in front of it. The notation *tuple*(*AttrList*) is assumed to construct the correct tuple type for the given *AttrList*.

Note that to use the expressions in a query you have to put the keyword `query` in front. Furthermore, a stream cannot be a final result of a query, so the query has to be completed by an operation such as *count* or *consume* (see Section 6). Hence a complete query for one of the examples above would be

```
query Trains feedproject[Id, Trip] consume
```

3.2 From a Stream of Atomic Values

One can also create a stream of tuples from a stream of values delivered by some operation. For example, the *units* operation can be used to convert a moving point into a stream of moving point units. Its signature is:

```
units: mpoint -> stream(upoint)          # ( _ )
```

Hence the expression

```
units(train7)
```

returns a stream of point units. Another example is the *intstream* operator creating a stream of integers, (all integers between the two arguments) with signature:

```
intstream: int x int -> stream(int)      # ( _ , _ )
```

The operations

- *transformstream*
- *namedtransformstream*

can be used to transform such a stream of values into a stream of tuples.

Examples:

```
units(train7) transformstream
units(train7) namedtransformstream[UTrip]
intstream(1, 10000) namedtransformstream[No]
```

Signatures:

```
transformstream: stream(Data)    -> stream(tuple([elem: Data]) )      _ #
namedtransformstream: stream(Data) x AttrName
                                 -> stream(tuple([AttrName: Data]))   _ #
```

Hence these operations create for each value in the input stream one tuple with a single attribute. For *transformstream* this attribute name is fixed (*elem*) whereas for *namedtransformstream* it can be specified. The type represented by type variable *Data* must belong to the kind DATA, that is, data types suitable as attribute types.

4 Operations on a Single Tuple Stream

4.1 Selecting Tuples

One can select a subset of the tuples from a given stream by operations:

- *filter*
- *head*, *tail*
- *ksmallest*, *kbiggest*

Filter implements selection by a predicate. *Head* and *tail* reduce the stream to the first or last *k* elements, respectively. *Ksmallest* and *kbiggest* return for a stream only the *k* smallest or largest tuples with respect to some sort order. This is done by maintaining a heap of *k* elements, hence is more efficient than sorting the entire tuple stream.

Examples:

```
Trains feed filter[.Id = 7]
Trains feed filter[.Trip passes mehringdamm]

Trains feed head[10]
Trains feed tail[10]

Trains feed ksmallest[Id; 10]
Trains feed ksmallest[Line, Id; 50]
```

Signatures:

```
filter: stream(Tuple) x (Tuple -> bool) -> stream(Tuple)     _ # [ _ ]
head, tail: stream(Tuple) x int -> stream(Tuple)            _ # [ _ ]
ksmallest, kbiggest: stream(Tuple) x AttrList x int
                                 -> stream(Tuple)           _ # [ _ ]
```

Note that the second parameter of filter is a function (mapping a tuple into a boolean value) which is written in the example in an abbreviated form. Without abbreviation it could be written as follows:

```
Trains feed filter[fun(t: TUPLE) attr(t, Id) = 7]
```

Here `TUPLE` is actually a function determining the type of a tuple in the input stream of the filter operator. The abbreviation allows one to omit the function head `fun(t: TUPLE)` and to refer to the argument tuple by the symbol "." and to any of its attributes "attr" by the notation ".attr". Hence the following notation is also valid.

```
Trains feed filter[attr(., Id) = 7]
```

4.2 Changing the Order of Tuples

A stream of tuples can be sorted by one or more attributes using operations

- *sortby*
- *sort*

The latter operation sorts lexicographically ascending by all attributes.

Examples:

```
Trains feed sortby[Line asc]
Trains feed sortby[Up asc, Id desc]
Trains feed sort
```

Signatures

```
sortby: stream(Tuple) x ((attr1 x dir1) x ... x (attr_n x dir_n))
                                  -> stream(Tuple)          _ # [ _ ]
sort: stream(Tuple)               -> stream(Tuple)          _ #
```

4.3 Removing Duplicates

Sorting is often combined with removing duplicates.

- *rdup*
- *krdup*

The first operation *rdup* is applicable to a totally ordered stream of tuples obtained by *sort*. The second allows one to specify some attributes for comparison and return from a group of tuples equal in these attributes only the first one. This operation is applicable to a stream ordered by a subset of the attributes, obtained by *sortby*.

Examples:

```
Trains feed sort rdup
Trains feed sortby[Line asc] krdup[Line]
```

Signatures

```
rdup: stream(Tuple)                -> stream(Tuple)          _ # [ _ ]
krdup: stream(Tuple)               -> stream(Tuple)          _ # [ _ ]
```

4.4 Tuple to Tuple Transformations

In this section we discuss operations that map each input tuple in a stream into a single output tuple.

Adding and Removing Attributes

This can be done by operations

- *extend*
- *project*, *remove*
- *projectextend*

Extend adds new attributes to a tuple derived from the values in the existing attributes, by specifying for each new attribute an attribute name and an expression (a function mapping the tuple into an atomic value). *Project* is the standard relational projection. *Remove* is similar to *project* but allows one to mention the attributes that should be removed rather than those that should stay. Finally, *projectextend* allows one to combine *project* and *extend* into a single operation.

Examples:

```
Trains feed extend[Dist: distance(.Trip, mehringdamm)]
Trains feed project[Id, Trip]
Trains feed remove[Trip]

Trains feed projectextend[Id, Line, Up
  ; Id1000: .Id * 1000,
    Mindist: minimum(distance(train7, mehringdamm))]
```

Signatures

```
extend: stream(Tuple) x
  ((NewAttr1 x (Tuple -> Data1)) x ... x (NewAttr_n x (Tuple -> Data_n)))
  -> stream(Tuple o tuple([NewAttr1: Data1, ..., NewAttr_n: Data_n]))
                                                          _ # [ _ ]

project: stream(Tuple) x AttrList
                              -> stream(tuple(AttrList))    _ # [ _ ]

remove: stream(Tuple) x AttrList
                      -> stream(tuple(Tuple - AttrList))    _ # [ _ ]

projectextend: stream(Tuple) x AttrList x
  ((NewAttr1 x (Tuple -> Data1)) x ... x (NewAttr_n x (Tuple -> Data_n)))
  -> stream(tuple(AttrList)
    o tuple([NewAttr1: Data1, ..., NewAttr_n: Data_n]))     _ # [ _ ; _ ]
```

Here o denotes concatenation of two tuple types and *NewAttrList* used in *projectextend* has the same form as the second argument in *extend*. The types *Data_1* through *Data_n* must be atomic data types in the kind DATA, i.e., suitable as attribute types. The notation *tuple*(*Tuple* - *AttrList*) is assumed to construct the correct tuple type obtained by removing the attributes in *AttrList* from type *Tuple*.

Again, in the examples parameter functions of *extend* have been written in abbreviated form; a full version is

```
Trains feed
  extend[Dist: fun(t: TUPLE) distance(attr(t, Trip), mehringdamm)]
```

4.5 Tuple to Tuple Stream Transformations

4.5.1 Combining Each Tuple With a Stream of Values

- *extendstream*
- *projectextendstream*

These operations take a tuple from the input stream and add an attribute whose value is obtained by an operation generating a stream of values. Examples of such operators are *units* and *intstream* introduced in Section 3.2. Since every output tuple can take only one of the values of the stream, one copy of the input tuple is made for each value. The operator *extendstream* just adds the new attribute, creating the required number of copies of the input tuple. The operator *projectextendstream* additionally allows one to specify a projection so that only selected attributes of the input tuple are copied into the result tuples.

Examples:

```
Trains feed extendstream[UTrip: units(.Trip)]
Trains feed projectextendstream[Id, Line, Up; UTrip: units(.Trip)]
```

Signatures

```
extendstream: stream(Tuple) x (NewAttr x (Tuple -> stream(Data)))
  -> stream(Tuple o tuple([NewAttr: Data]))                  _ # [ _ ]
projectextendstream: stream(Tuple) x AttrList x
  (NewAttr x (Tuple -> stream(Data)))
  -> stream(tuple(AttrList) o tuple([NewAttr: Data]))    _ # [ _ ; _ ]
```

Note that these operations add only a singe stream-valued attribute, in contrast to *extend* and *projectextend* which can add several attributes.

4.5.2 Combining Each Tuple With a Stream of Tuples

Analogously to *extendstream* and *projectextendstream* one can also combine each tuple of the input stream with a stream of tuples generated from the input tuple. This is in fact a join operation called *loopjoin*. The stream generated from the input tuple can, for example, be obtained by retrieving tuples from an index based on attributes of the input tuple.

- *loopjoin*
- *loopsel*

The *loopsel* operation generates for each input tuple a stream of tuples in the same way as *loopjoin*. However, it does not concatenate the input tuple with each output tuple but instead simply returns all the streams created for tuples concatenated into a single stream. Hence it can be used to implement a semijoin.

Examples:

```
Trains feed loopjoin[fun(t: TUPLE)
  Trains feed filter[sometimes(.Trip = attr(t, Trip))] {t2}]
```

This is a simple nested loop join. The inner relation is scanned once for every tuple of the outer stream. Here `{t2}` is a renaming needed to make attribute names distinct.

```
strassen feed loopjoin[
  strassen_geoData_rtree strassen windowintersects[.geoData] {s2}]
```

This is an index nested loop join. For every tuple of the *strassen* relation, a range query on the R-tree index is executed to retrieve roads whose bounding box intersects the bounding box of the road in the input tuple.

```
Trains feed loopsel[
  fun(t: TUPLE) strassen feed filter[attr(t, Trip) passes .geoData]]
```

This query finds all roads that are passed by underground trains. Hence a subset of the strassen relation is returned, containing duplicates.

Signatures:

```
loopjoin: stream(Tuple1) x (Tuple1 -> stream(Tuple2)
                        -> stream(Tuple1 o Tuple2)            _ # [ _ ]
loopsel: stream(Tuple1) x (Tuple1 -> stream(Tuple2)
                        -> stream(Tuple2)                     _ # [ _ ]
```

4.6 Tuple Stream to Tuple Transformations

Grouping

- *groupby*

Obviously, grouping is the operation that returns for each group of tuples a single tuple. The *groupby* operator is applicable to a stream of tuples ordered by the grouping attributes; for each group, it allows one to derive some new attributes.

Examples:

```
Trains feed sortby[Line asc, Up asc]
  groupby[Line, Up
  ; Cnt: group count,
    MinId: group feed min[Id],
    MaxId: group feed max[Id]]
```

```
Trains feed sortby[Line asc]
  groupby[Line
  ; PassMehr: group feed filter[.Trip passes mehringdamm] count]
```

The first query groups trains by line and direction and computes for each such group the number of trains and the minimal and maximal *Id*. The second determines for each train line how many of its trains pass through *mehringdamm*.

Signature:

```
groupby: stream(Tuple) x AttrList x
  ((NewAttr1 x (stream(Tuple) -> Data1)) x ... x
  (NewAttr_n x (stream(Tuple) -> Data_n)))
    -> stream(tuple(AttrList) o
        tuple([NewAttr1: Data1, ..., NewAttr_n: Data_n])   _ # [ _ ; _ ]
```

5 Operations on Two Tuple Streams (Joins)

This section considers symmetric operations on two tuple streams, i.e., joins. Note that *loopjoin* is also available as an asymmetric join technique (Section 4.5.2).

5.1 Generic Join

- *symmjoin*

The symmjoin operator implements a symmetric, non-blocking, nested loop join technique, so the complexity is $m \cdot n$ for tuple streams of sizes m and n, respectively. The main advantage of *symmjoin* is that it admits arbitrary join conditions, hence can always be used to implement a join.

Examples:

```
Trains feed {t1} Trains feed {t2}
  symmjoin[sometimes(distance(.Trip_t1, ..Trip_t2) < 1000.0)]
```

This finds all pairs of trains whose time dependent distance function at some time has a value less than 1000. The parameter function of *symmjoin* has two argument tuples, one from the first and one from the second argument stream. In the abbreviated form shown above, one can refer to the first argument as "." and the second as ".."; similarly one can refer to an attribute *attr* of the first argument tuple as "*.attr*" and an attribute of the second argument tuple as "*..attr*". The full form of the query above is

```
Trains feed {t1} Trains feed {t2}
  symmjoin[fun(t1: TUPLE, t2: TUPLE2)
    sometimes(distance(attr(t1, Trip_t1), attr(t2, Trip_t2)) < 1000.0)]
```

Signature:

```
symmjoin: stream(Tuple1) x stream(Tuple2) x (Tuple1 x Tuple2 -> bool)
  -> stream(Tuple1 o Tuple2)                                  _ _ # [ _ ]
```

5.2 Equi-Join

- *sortmergejoin*
- *mergejoin*
- *hashjoin*

Three methods are available to perform an equijoin. These are the standard techniques from the literature. *Mergejoin* is applicable if the two input streams are already ordered by the join attribute.

Examples:

```
Trains feed {t1} Trains feed {t2} sortmergejoin[Line_t1, Line_t2]
Trains feed {t1} Trains feed {t2} mergejoin[Id_t1, Id_t2]
Trains feed {t1} Trains feed {t2} hashjoin[Id_t1, Id_t2]
```

Note that before SECONDO version 3.1, hashjoin had an additional parameter for the number of buckets, so would have to be written as

```
Trains feed {t1} Trains feed {t2} hashjoin[Id_t1, Id_t2, 99997]
```

Signatures:

```
sortmergejoin: stream(Tuple1) x stream(Tuple2) x AttrName1 x AttrName2
  -> stream(Tuple1 o Tuple2)                                  _ _ # [ _ , _ ]
mergejoin: stream(Tuple1) x stream(Tuple2) x AttrName1 x AttrName2
  -> stream(Tuple1 o Tuple2)                                  _ _ # [ _ , _ ]
hashjoin: stream(Tuple1) x stream(Tuple2) x AttrName1 x AttrName2
  -> stream(Tuple1 o Tuple2)                                  _ _ # [ _ , _ ]
```

5.3 Spatial Join

- *spatialjoin*

The *spatialjoin* operator determines efficiently pairs of tuples from the input streams for which the (2D or 3D) bounding boxes of the mentioned spatial attributes overlap. It implements a grid based spatial join technique similar to (Patel & DeWitt, Partition-Based Spatial Mergejoin, SIGMOD 1996).

Examples:

```
strassen feed {s1} strassen feed {s2}
  spatialjoin2[geoData_s1, geoData_s2]

Trains feed extend[Box: bbox(.Trip)] {t1}
Trains feed extend[Box: bbox(.Trip)] {t2}
  spatialjoin2[Box_t1, Box_t2]
```

Attributes of simple 2D spatial data types (such as *point*, *line*, *region*) can be mentioned directly whereas for temporal types (*mpoint*, *upoint*) bounding boxes have to be added explicitly.

Note that a robust operator *spatialjoin* comes only with SECONDO version 3.1; unfortunately the previous implementation *spatialjoin* (now renamed *spatialjoin0*) crashed occasionally.

Signature:

```
spatialjoin: stream(Tuple1) x stream(Tuple2) x SpatialAttr1 x
  SpatialAttr2 -> stream(Tuple1 o Tuple2)                _ _ # [ _ , _ ]
```

6 Consuming a Tuple Stream

6.1 Into a Relation

- *consume*
- *tconsume*

These two operations collect a tuple stream into a relation. *Consume* creates a persistent relation suitable to be kept and indexed. *Tconsume* is useful for temporary results and tries to keep tuples in memory as far as possible.

Examples:

```
Trains feed filter[.Trip passes mehringdamm] consume
Trains feed filter[.Trip passes mehringdamm] tconsume
```

Signatures:

```
consume: stream(Tuple) -> rel(Tuple)
tconsume: stream(tuple) -> trel(Tuple)
```

6.2 Into an Aggregate Value

- *count*
- *sum*, *min*, *max*, *avg*
- *extract*
- *aggregateB*

Aggregate functions can be applied to a tuple stream to derive a single atomic value. For *sum*, *min*, *max*, and *avg* one has to specify an attribute for which the respective function is applied. *Extract* allows one to get a typed atomic value of any attribute data type out of a tuple; it simply extracts this attribute frome the first tuple in the stream.

Examples:

```
Trains feed filter[.Trip passes tiergarten] count
strassen feed extend[Length: size(.geoData)] avg[Length]
Trains feed filter[.Id = 322] extract[Trip]
```

Hence the result of the third expression is of type *mpoint*.

The *aggregateB* operator provides an efficient implementation of a generic aggregate function. One has to specify an attribute of the tuple stream, a function combining two attribute values, and a value to be returned if the stream is empty. The operator uses a stack to merge partial results of equal sizes, rather than combining a single value with the aggregation of all previous values.

Examples:

```
strassen feed aggregateB[geoData
  ; fun(l1: line, l2: line) l1 union l2; [const line value ()]]
```

This computes the entire road network of Berlin as a single *line* value.

Signatures:

```
count: stream(Tuple) -> int                                  - #
min, max: stream(Tuple) x (AttrName: Data) -> Data           _ # [ _ ]
avg: stream(Tuple) x (AttrName: TNum) -> real                _ # [ _ ]
aggregateB: stream(Tuple) x (AttrName: Data)
  x (Data x Data -> Data) x Data -> Data                     _ # [_ ; _ ; _ ]
```

Here (*AttrName: Data*) denotes an attribute name referring to an attribute of a type Data (and type *Data* must be in the kind DATA, that is, suitable as an attribute type of a relation). Type *TNum* must be a numeric type, i.e., *int* or *real*.

Aggregate functions are especially useful in connection with grouping. Examples are shown in Section 4.6.

6.3 Into a Stream of Values

- *transformstream*

A stream of tuples that have only one attribute can be transformed into a stream of values. This enables one to further apply operators taking a stream of values as an argument. One such operator is *collect_points* which takes a stream of point values and collects them into a *points* value (the data type *points* can represent a set of points). We can the use operations applicable to *points* values. *Convex_hull* is one of those. The two mentioned operations have signatures:

```
collect_points: stream(point) x bool -> points               _ # [ _ ]
convexhull: points -> region                                 # ( _ )
```

The boolean parameter in *collect_points* tells whether undefined point values in the input stream should be ignored or should make the whole value undefined.

Example:

```
convexhull(Kinos feed project[geoData] transformstream
  collect_points[TRUE])
```

This computes the convex hull of the Kino positions and so roughly defines a “cinema area”.

Signature:

```
transformstream: stream(tuple([Attr: T])) -> stream(T)       _ #
```