DOCUMENT INFORMATION

TITLE	**A Lisp dialect for NDB interpreted code**
AUTHOR	**Max-Gerd Retzlaff, Datagraph GmbH**
DESCRIPTION	Slides for the presentation of the paper
CONFERENCE	European Lisp Symposium 2026, Kraków, May 11–12, 2026, https://european-lisp-symposium.org/2026
TIME SLOT	May 12th, 2026, 16:00–16:30
PAPER	Max-Gerd Retzlaff (2026). A Lisp dialect for NDB interpreted code. The 19th European Lisp Symposium (ELS'26), Kraków, Poland, May 11-12, 2026. European Lisp Symposium, ISSN: 2677-3465. doi:10.5281/zenodo.20290252
PUBL. LINK	https://matroid.org/id/3b52dbda-c2f8-477e-9553-9084220c749c
LICENSE	 Creative Commons Attribution–NoDerivatives 4.0 Intl.
COPYRIGHT ©	2026 Max-Gerd Retzlaff, Datagraph GmbH, Berlin, Germany.

A LISP DIALECT FOR NDB INTERPRETED CODE

Max-Gerd Retzlaff, Datagraph GmbH

European Lisp Symposium 2026, Kraków, May 11–12, 2026

Publication location: http://www.matroid.org/id/3b52dbda-c2f8-477e-9553-9084220c749c

DYDRA

- a graph store / RDF store / triplestore
  - SPARQL endpoint / processor SPOCQ in Lisp
  - data accession in Lisp
  - data storage in C++ and Lisp
- new cluster backend for Dydra,
  based on RonDB (NDB / MySQL NDB Cluster)
- cl-ndbapi: Common Lisp bindings for the C++ NDB API of RonDB,
  published at https://github.com/datagraph/cl-ndbapi

NDB INTERPRETED CODE

The language:

- allows to write NDB API interpreted programs
  that can be sent to the RonDB data nodes
  and evaluated within a RonDB cluster,
- is used to implement the `NdbScanFilter` functionality,
  that allows to define groups of conditions.

But the low-level interpreted code facility is fully documented and provided as part of the NDB API.

⇒ *You can just use it directly.*

NDB INTERPRETED CODE

The NDB API contains a low-level language:

- targets a simple register machine
- eight registers
- instructions to load constans, add and substract, and define labels
- branching commands such as `branch_ne` and `branch_gt`,
  and the unconditional jump `branch_label`
  also known as the infamous `go to` [Dijkstra 1968].
- no high level conditionals such as `if` and no loop constructs,
  you have to break it down to labels, low-level branching and jumps

⇒ *It looks like a simple assembly language.*

NDB INTERPRETED CODE

Powerful commands specific to an SQL data backend:

- read and write table columns: `read_attr` and `write_attr`
- a battery of twenty branch commands,
  all starting with `branch_col_*`,
  that allow to branch depending on table columns.

All these commands seemed to allow to just read or write whole columns,
that is table column values, which limits their use for us.
⇒ *That changed!*

NDB IC IN RONDB 24.10

Mikael Ronström implemented:

- not only the minimal set of my initial extension proposal
- but a substantial overhaul of the language:
  - interpreter memory
  - new instruction format to allow for more commands
  - dozens of new commands
  - new documentation: “RonDB Generic interpreter” https://docs.rondb.com/rondb_generic_interpreter/

NEW NDB IC EXTENSIONS

- more arithmetic operations
- reading whole columns into the new interpreter memory
- reading only parts of a column
- reading from the interpreter memory into registers
- the same for writing
- input parameters for NDB IC programs
- higher-level functions such as binary and interval search

cl-ndbapi FOR RONDB 24.10

introduced a Lisp dialect for NDB interpreted code (NDB IC):

- a cross-language compiler for NDB interpreted code
- with two backends:
  a) NDB interpreted code

  b) Common Lisp (CL) code for an implementation in Common Lisp of a subset of the NDB interpreted code commands

cl-ndbapi FOR RONDB 24.10

introduced a Lisp dialect for NDB interpreted code (NDB IC):

- a cross-language compiler for NDB interpreted code
- with two backends:
  a) NDB interpreted code

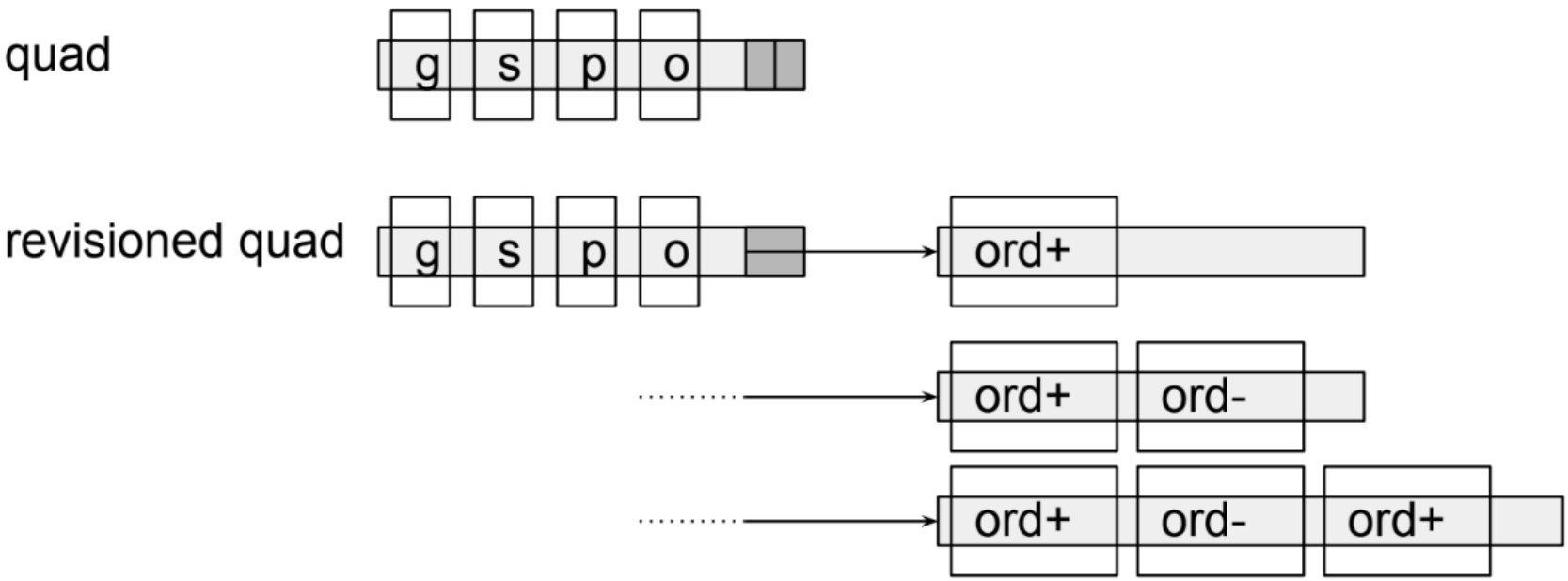
     → to be run within a RonDB cluster

  b) Common Lisp (CL) code for an implementation in Common Lisp of a subset of the NDB interpreted code commands

     → meant for testing and debugging new NDB IC in the more accessible environment of the Lisp image

QUADS WITH VISIBILITY

- Dydra is actually a revisioned RDF store.
- All RDF statements are associated with transaction-time metadata.
- This “visibility” is encoded as a sequence of revision ordinals.
- Ordinals identify transactions that added and removed each quad.



SIMPLE RDF STORE IN SQL

A triple store graph repository *with revisions*

could be created with this SQL call:

```
create table quad
  (g int unsigned not null, s int unsigned not null,
   p int unsigned not null, o int unsigned not null,
   primary key (g,s,p,o), index gpos (g,p,o,s),
   index gosp (g,o,s,p), index spog (s,p,o,g),
   index posg (p,o,s,g), index ospg (o,s,p,g),
   visibility varbinary(28000) not null);
```

spocq.i::%test-visibility - PART 1

```
1  (defun spocq.i-%test-visibility (ordinal %vector length)
2    "Locate the position of the highest ordinal less than that given.
3   Iff it is even, then the quad is visible in that revision.
4   Handle the degenerate cases without searching:
5   - length 0 implies unrevisioned
6   - length 1 implies presence at or after after that ordinal
7   - length 2 implies between the two"
8    (declare (optimize (speed 3) (debug 0) (safety 0))
9             (type (integer 0 4294967295) ordinal length))
10   (labels ((binary-search (start end)
11              (declare (type (integer 0 4294967295) start end))
12              (let* ((test-position (ash (the (integer 0 4294967295) (+ start end)) -1))
13                     (test-ordinal (cffi:mem-aref %vector :uint32 test-position)))
14                (cond
15                  ((< test-ordinal ordinal) (if (= test-position start)
16                                                start
17                                                (binary-search test-position end)))
18                  ((> test-ordinal ordinal) (if (= test-position start)
19                                                nil
20                                                (binary-search start test-position)))
21                  (t test-position)))))
```

spocq.i::%test-visibility – PART 2

```
22      (case length
23        (0 t)
24        (1 (when (>= ordinal (cffi:mem-aref %vector :uint32 0))
25             (values t 0)))
26        ;; for a very small repository, this special case was worth 0.17 elapsed time
27        (2 (when (>= ordinal (cffi:mem-aref %vector :uint32 0))
28             (if (< ordinal (cffi:mem-aref %vector :uint32 1))
29                 (values t 0)
30                 (values nil 1))))
31        (t (let ((position (binary-search 0 length)))
32             (when position
33               (cond ((evenp position)
34                      (values t position))
35                     ((and ;;(oddp position) ;; implicit, see above (20250310 mgr)
36                       (> length (1+ position))
37                       (= ordinal (cffi:mem-aref %vector :uint32 (1+ position))))
38                      ;; if the found value was a delete, but the next is an insert,
39                      ;; then it is present
40                      (values t (1+ position)))
41                     (t
42                      (values nil position)))))))))
```

MANUAL TRANSLATION

- Writing the NDB IC version was more complex than anticipated.
- Once written, it appeared to work.

MANUAL TRANSLATION

- Writing the NDB IC version was more complex than anticipated.
- Once written, it appeared to work.

But the real problem was testing it and how to test it:

- Running NDB IC programs in RonDB is opaque.
- no introspection, no printing or writing to log files
- only options:
  - writing intermediate results back to columns
  - conditionalized return of rows

MANUAL TRANSLATION

- Writing the NDB IC version was more complex than anticipated.
- Once written, it appeared to work.

But the real problem was testing it and how to test it:

- Running NDB IC programs in RonDB is opaque.
- no introspection, no printing or writing to log files
- only options:
  - writing intermediate results back to columns
  - conditionalized return of rows

New approach: write a compiler for NDB IC first

- It was faster to write it, test it and then implement the visibility algorithm in that,
- than the time it took to do the one manual conversion.

WRITING A SIMPLE COMPILER IN COMMON LISP IS EASY

In total:

- I do not have to parse the source language,
  as I stick with something Lisp-like, and
- I do not have to do the actual code generation,
  as that is done by the NDB API already.
- All that remains to be done is code transformation.
  ⇒ *And that is easy with Lisp's macros.*

DESIGN DECISIONS

- be pragmatic and keep the compiler simple
- higher-level conditionals such as `when`, `if`, and `cond` rather than defining labels with branching and jumps
- automatic label generation and jump resolution
- some form of recursion[1] to implement binary search efficiently
- Register allocation I did not bother to implement so far.

[1] NDB IC actually supports subroutines and also recursion. The maximum subroutine stack depth of 32 ought to be more than enough, but there was no information available on tail recursion or how it was implemented, so I rather wanted to supply something simple based on jumps that was guaranteed to be efficient.

MANUAL REGISTER ALLOCATION

- keep register constants in lexical bindings

Use the `:registers` argument of `ic:with-compiled-ic` form:

```
(ic:with-compiled-ic (:registers ((ordinal 0)
                                  (value 4))
                      :code code)
  (ic:load-const-u32 code ordinal 0)
  (ic:when (ic:null value)
    (ic:interpret-exit-nok code))
  (ic:interpret-exit-ok code))
```

LABEL ALLOCATION

needs to be automatic in order to implement the higher-level conditionals

- labels are just numbers in NDB IC
- use just a simple counter
- `ic.i::get-next-label` returns the next free number and increments the counter afterwards

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defvar *next-label* 0)

  (defun get-next-label ()
    ;; very simple for now...
    (prog1
        *next-label*
      (incf *next-label*))))
```

LABEL ALLOCATION

```
(ic:with-compiled-ic (:registers ((ordinal 0)
                                  (value 4))
                      :code code)
  (ic:load-const-u32 code ordinal 0)
  (ic:when (ic:null value)
    (ic:interpret-exit-nok code))
  (ic:interpret-exit-ok code))
```

which the `ic.ndbapi` backend transforms to:

```
(LET ((ORDINAL 0) (VALUE 4) (IC.I::*CODE* CODE))
  (NDBAPI.IC:LOAD-CONST-U32 CODE ORDINAL 0)
  (PROGN
   (NDBAPI.IC:BRANCH-NE-NULL IC.I::*CODE* VALUE 0)
   (NDBAPI.IC:INTERPRET-EXIT-NOK CODE)
   (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0))
  (NDBAPI.IC:INTERPRET-EXIT-OK CODE)
  (NDBAPI.IC:FINALISE IC.I::*CODE*))
```

LABEL ALLOCATION

```
(ic:with-compiled-ic (:registers ((ordinal 0)
                                  (value 4))
                      :code code)
  (ic:load-const-u32 code ordinal 0)
  (ic:when (ic:null value)
    (ic:interpret-exit-nok code))
  (ic:interpret-exit-ok code))
```

which the `ic.lisp` backend transforms to:

```
(BLOCK IC.LISP::IC-CODE-BLOCK
  (LET ((ORDINAL 0) (VALUE 4) (IC.I::*CODE* CODE))
    (DECLARE (TYPE (UNSIGNED-BYTE 32) ORDINAL VALUE))
    (SETQ ORDINAL 0)
    (TAGBODY
      (IF (/= VALUE 4294967295)
          (GO 0))
      (RETURN-FROM IC.LISP::IC-CODE-BLOCK NIL)
     0)))
```

HIGH-LEVEL CONDITIONALS

The NDB IC compiler defines:

- `ic:when`, `ic:if`, `ic:cond` and `ic:while`.

And as test for these conditionals it defines

- the binary tests: <, >, >=, <=, =, and /=
- as well as the unary tests:
  `ic:zerop`, `ic:onep`, `ic:twop`, and `ic:null`

Each test is transformed to a specific branch instruction

- such as: `ic:branch-ge` and `ic:branch-ne`
- and branch variants introduced by the NDB IC compiler such as:
  `ic:branch-ne-zero` and `ic:branch-ne-one`,
  which resolve to `ic:branch-ne-const`

ic:when, ic:if, ic:cond AND ic:while

- ic:when is implemented as:

```
(defmacro ic:when ((test &rest rest) &body forms)
  (let ((end-label (get-next-label)))
    `(,(if (ic:ic-lisp-mode-p)
           'tagbody
           'progn)
      (test-to-branch ,test ,rest ,end-label)
      ,@forms
      ,(def-label% '*code* end-label))))
```

- The others are quite similar.
- But ic:cond is implemented as a recursive macro and turns a ic:cond form in a series of nested ic:if clauses.

STEP BY STEP MACRO EXPANSION OF `ic:when`

```
(ic:when (ic:null value)
  (ic:interpret-exit-nok code))
⇒
(PROGN
 (IC.I::TEST-TO-BRANCH IC:NULL (VALUE) 0)
 (NDBAPI.IC:INTERPRET-EXIT-NOK CODE)
 (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0))
⇒
(PROGN
 (IC.I::TRANSFORM-FIRST-ARG IC.I::*CODE*
   NDBAPI.IC:BRANCH-NE-NULL (VALUE) 0)
 (NDBAPI.IC:INTERPRET-EXIT-NOK CODE)
 (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0))
⇒
(PROGN
 (NDBAPI.IC:BRANCH-NE-NULL IC.I::*CODE* VALUE 0)
 (NDBAPI.IC:INTERPRET-EXIT-NOK CODE)
 (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0))
```

ic:when WITH EXTRACTION

So one can write:

```
(ic:when (> length (ic:add-const-reg code value test-position 1))
  (ic:interpret-exit-ok code))
```

which will be resolved to:

```
(PROGN
 (PROGN
  (NDBAPI.IC:ADD-CONST-REG CODE VALUE TEST-POSITION 1)
  (NDBAPI.IC:BRANCH-LE IC.I::*CODE* LENGTH VALUE 0))
 (NDBAPI.IC:INTERPRET-EXIT-OK CODE)
 (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0))
```

Note: The transformation still needs an intermediate register, here `value`.

ic:when WITH EXTRACTION

So one can write:

```
(ic:when (> length (ic:add-const-reg code value test-position 1))
  (ic:interpret-exit-ok code))
```

which will be resolved to:

```
(PROGN
 (PROGN
  (NDBAPI.IC:ADD-CONST-REG CODE VALUE TEST-POSITION 1)
  (NDBAPI.IC:BRANCH-LE IC.I::*CODE* LENGTH VALUE 0))
 (NDBAPI.IC:INTERPRET-EXIT-OK CODE)
 (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0))
```

Note: The transformation still needs an intermediate register, here `value`.

⇒ allows to write long `ic:cond` forms consisting of many test forms, that each might require an extraction step first

⇒ more compact and idiomatic Lisp code

SIMPLE RECURSION WITH `ic:block`

- allows to jump back to the beginning of the code block by using a direct jump
- it introduce a local function with the specified name, which allows to recur:

```
(defmacro ic:block (recur &body forms)
  (let ((block-label (get-next-label)))
    `(,(if (ic:ic-lisp-mode-p)
           'tagbody
           'progn)
      ,(def-label% '*code* block-label)
      (labels ((,recur ()
                 (ic:branch-label *code* ,block-label)))
        ,@forms))))
```

ic:block EXAMPLE

A simple, rather nonsensical example might be:

```
(ic:block again
 (ic:when (ic:zerop value)
   (again))))
```

which the ic.ndbapi backend transforms to:

```
(PROGN
 (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0)
 (LABELS ((AGAIN ()
            (NDBAPI.IC:BRANCH-LABEL IC.I::*CODE* 0)))
   (PROGN
    (NDBAPI.IC:BRANCH-NE-CONST IC.I::*CODE* VALUE 0 1)
    (AGAIN)
    (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 1))))
```

ic:block EXAMPLE

A simple, rather nonsensical example might be:

```
(ic:block again
 (ic:when (ic:zerop value)
   (again))))
```

which the ic.ndbapi backend transforms to:

```
(PROGN
 (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 0)
 (LABELS ((AGAIN ()
            (NDBAPI.IC:BRANCH-LABEL IC.I::*CODE* 0)))
   (PROGN
    (NDBAPI.IC:BRANCH-NE-CONST IC.I::*CODE* VALUE 0 1)
    (AGAIN)
    (NDBAPI.IC:DEF-LABEL IC.I::*CODE* 1))))
```

- This code is still executed as Lisp code first and
- the NDBAPI.IC functions will produce the native NDB IC code.

ic:block EXAMPLE

A simple, rather nonsensical example might be:

```
(ic:block again
 (ic:when (ic:zerop value)
   (again))))
```

which the `ic.lisp` backend transforms to:

```
(TAGBODY
 0
  (LABELS ((AGAIN ()
             (GO 0)))
    (TAGBODY
      (IF (/= VALUE 0)
          (GO 1))
      (AGAIN)
     1)))
```

%test-visibility

```
1  (defun %test-visibility (vis-column &key memory-offset
2                                           (memory-header 4) ;; size of memory header in bytes
3                                           code
4                                      &aux (visibility-offset (+ memory-offset memory-header)))
5    "Locate the position of the highest ordinal less than that given.
6   Iff it is even, then the quad is visible in that revision.
7   Handle the degenerate cases without searching:
8   - length 0 implies unrevisioned
9   - length 1 implies presence at or after after that ordinal
10  - length 2 implies between the two"
11   (declare (optimize (speed 3) (debug 0) (safety 0))
12            ;; note: there is no cffi type for sb-alien:system-area-pointer
13            (type sb-alien:system-area-pointer vis-column)
14            (type (or null ndbapi:ndb-interpreted-code) code)
15            (type (unsigned-byte 32) memory-offset memory-header visibility-offset))
16   (declare (ignorable vis-column))
17   (ic:with-compiled-ic (:registers ((ordinal 0) ;; input: the ordinal to check for
18                                     (length 1)
19                                     (test-position 2)
20                                     (test-ordinal 3) ;; ordinal loaded from
21                                                      ;; vis-column-address at a position
22                                     (start 4)
23                                     (end 5)
24                                     ;; used to hold intermediate constants
25                                     (value 6))
26                         :code code)
```

```
27      ;; set input of algorithm:
28      (ic:read-interpreter-input code ordinal 0)
29      ;; memory-offset for ic:read-full needs to be stored in register:
30      (ic:load-const-u32 code value memory-offset)
31      (ic:read-full code vis-column value value)
32      ;;(format *trace-output* "read ~a bytes~%" extra)
33      (ic:read-u16-to-reg/const code length visibility-offset)
34      ;; divide by 4 to turn number of bytes to number of revision ordinals:
35      (ic:rshift-const-reg code length length 2)
36      (ic:load-const-u32 code start 0)
37      (ic:move-reg code end length)
38
39      (ic:cond
40        ((ic:zerop length)
41         (ic:interpret-exit-ok code))
42        ((ic:onep length)
43         (ic:read-u32-to-reg/const code test-ordinal (+ visibility-offset 2))
44         (ic:if (>= ordinal test-ordinal)
45                (ic:interpret-exit-ok code)
46                (ic:interpret-exit-nok code)))
47        ;; for a very small repository, this special case was worth 0.17 elapsed time
48        ((ic:twop length)
49         (ic:read-u32-to-reg/const code test-ordinal (+ visibility-offset 2))
50         (ic:when (>= ordinal test-ordinal)
51           (ic:read-u32-to-reg/const code test-ordinal (+ visibility-offset 6))
52           (ic:when (< ordinal test-ordinal)
53             (ic:interpret-exit-ok code)))
54         (ic:interpret-exit-nok code))
```

```
55      (t
56       (ic:block binary-search
57         ;; input: test-ordinal, start, end
58         ;; used constants: visibility-offset, numbersize=4, dataoffset=2
59         ;; used registers: value, test-position
60         ;; output: test-position (an index or null)
61         (ic:add-reg code test-position start end)
62         (ic:rshift-const-reg code test-position test-position 1)
63         (ic:lshift-const-reg code value test-position 2)
64         (ic:add-const-reg code value value (+ visibility-offset 2))
65         (ic:read-u32-to-reg/reg code test-ordinal value)
66         (ic:cond
67           ((< test-ordinal ordinal)
68            (ic:if (= test-position start)
69                   ;; position found
70                   (ic:move-reg code test-position start)
71                   (progn
72                     (ic:move-reg code start test-position)
73                     (binary-search))))
74           ((> test-ordinal ordinal)
75            (ic:if (= test-position start)
76                   ;; position not found
77                   (ic:load-const-null code test-position)
78                   (progn
79                     (ic:move-reg code end test-position)
80                     (binary-search))))
81           (t
82            ;; not: (copy-reg test-position test-position :reg-help value)
83            ;; just "return" test-position as is
84            )))
```

```
86        (ic:cond
87          ((ic:null test-position)
88           (ic:interpret-exit-nok code))
89          ;; complex argument is allowed for zerop as test
90          ((ic:zerop (ic:evenp-reg code value test-position))
91           (ic:interpret-exit-ok code))
92          ((> length (ic:add-const-reg code value test-position 1))
93              ;; and (oddp test-position), but this is implied by the previous test already
94           (ic:lshift-const-reg code value value 2)
95           (ic:add-const-reg code value value (+ visibility-offset 2))
96           (ic:read-u32-to-reg/reg code test-ordinal value)
97           (ic:if (= ordinal test-ordinal)
98                  ;; if the found value was a delete, but the next is an insert,
99                  ;; then it is present
100                 (ic:interpret-exit-ok code)
101                 (ic:interpret-exit-nok code))))))
102     (ic:interpret-exit-nok code)))
```

USING THE PROGRAM IN A SCAN

- The program generated by `%test-visibility` can be used in a scan as described in the previous paper.
- The code object can be allocated like this:

```
(ndbapi.ic:with-code (code table)
  (%test-visibility vis-column
                    :memory-offset memory-offset
                    :memory-header 4 ;; 4 byte header
                    :code code)
  ;; ... use code in a scan...
  )
```

EVALUATION

Dataset:

- a copy of a real production database “plm-stats” with 4.5 million quads
- It exhibits a complex revision history, that was the result of having been used in a real production system over some years.

Computer:

- a server with 1 TB of RAM and
- a single “AMD EPYC 7502P 32-Core Processor” with Hyper-Threading (HT), resulting in 64 virtual cores.

Test scenario:

- full index scans of the main index using a committed read
- but filtering for different revision ordinals

For the native Common Lisp implementation:

- The filtering is done by the visibility algorithm implemented in CL.
- All 4.5 M rows have to be transferred to the Lisp image running SPOCQ.
- That includes all visibility vectors!

For the implementation in the new NDB IC Lisp dialect:

- The filtering happens already on the RonDB data nodes.
- The cluster was configured to use just one active RonDB data node but the table was still divided into six partitions, resulting in six threads doing the filtering in parallel.
- Only the matching rows will be be transferred to SPOCQ.
- And the visibility vectors do not leave the data nodes at all.

- not a corner case of the algorithm
- returns 1,421,523 of the 4,474,991 rows

test	duration	type of revision check
1	1.440 s	implemented in Common Lisp
2	0.364 s	full NDB IC implementation (118 words)
3	0.324 s	NDB IC using search-interval-32 (14 words)
4	0.316 s	as test 3, but re-using ndb and code objects

- Test 2 needs 75 % less time then test 1, a speed-up of 4.0.
- Test 3 makes it another 11 % faster, a speed-up of 1.1.
- Test 4 saves additionally 2 %, a speed-up of 1.0.
- Overall, test 4 is 78 % faster than test 1, a speed-up of 4.6.

- ordinal “2” is actually the ordinal of the first import
  (“1” is always the empty, initial revision of any repository)
  ⇒ a corner case of the algorithm, handled quickly, no recursion
- returns 102,699 of the 4,474,991 rows, only 2.3 % of all rows,

test	duration	type of revision check
1	1.360 s	implemented in Common Lisp
2	0.324 s	full NDB IC implementation (118 words)
3	0.292 s	NDB IC using search-interval-32 (14 words)
4	0.288 s	as test 3, but re-using ndb and code objects

⇒ We can use it to compare scan rates.

test	duration	type of revision check
1	1.360 s	implemented in Common Lisp
2	0.324 s	full NDB IC implementation (118 words)
3	0.292 s	NDB IC using search-interval-32 (14 words)
4	0.288 s	as test 3, but re-using ndb and code objects

- Test 1 needs to transfer all 4,474,991 rows including the visibility vector to the API node. The resulting scan rate is 3.3 M rows/s.
- In test 2 only 102,699 rows are transferred to the API node, still the algorithm scans all 4,474,991 rows. The scan rate is 13.8 M rows/s.
- Test 3 is similar to test 2 just using a different implementation. The scan rate is 15.3 M rows/s.
- Test 4 is also similar to test 2. The scan rate is 15.5 M rows/s.

- In total we achieved an internal scan rate of up to 15.5 M rows/s.
- The most optimized version of the NDB IC implementation is 4.6 times as fast as the version that does the revision resolution in SPOCQ.
- Having the full algorithm with all distinctions of cases and binary search implemented as a rather long NDB IC implementation is already 4.0 times as fast.

UPDATED RESULTS FOR THE SCAN RATES NOVEMBER 25, 2025

New test release of RonDB 25.10.0:

- I had supplied multiple test programs to Mikael Ronström that he used
- parts of RonDB mainly connected to locking and sorting,
  now parallelized

Same computer:

- 1 TB of RAM and a single “AMD EPYC 7502P 32-Core Processor”

Dataset:

- a big database with 3.3 billion quads containing
  the human reference genome GRCh38 of TogoVar [Mitsuhashi 2022],
  chromosomes 1 through 22 plus X, Y and MT, as well as
  the ClinVAR (Clinical significance of variants) dataset.
- table devided into 18 instead of 6 partitions
- again full index scans of the main index using a committed read

UPDATED RESULTS FOR THE SCAN RATES NOVEMBER 25, 2025

- For ordinal “2”, just 0.002 % of all rows are returned.
  It took 86.287 s. That means an internal scan rate of 37.7 M rows/s.
- For ordinal “12” half of the rows (52.5 %) are returned.
  Just counting took 126.3 s — a scan rate of 25.8 M rows/s;
  with transfer of the row data: 180.4 s — a scan rate of 18.0 M rows/s.
- For ordinal “27” all rows are returned, 3,252,619,492 rows.
  Just counting took 187.2 s — a scan rate of 17.4 M rows/s;
  with transfer of the row data: 303.9 s — a scan rate of 10.7 M rows/s.

CONCLUSION

NDB IC IN RONDB 24.10

In collaboration of the teams behind RonDB and Dydra:

- NDB IC facility of RonDB was extended considerably
- to allow implementing the versioning algorithm of Dydra as NDB IC and push this visibility evaluation into the RonDB data nodes:
  - run in multiple parallel instances within the RonDB cluster
  - each data node returns only statements already filtered by the temporal logic

→ *Push the test to the data instead of pulling all data to the test.*

APPLICATION LOGIC AS NDB IC PROGRAMS

- Data that used to be opaque to RonDB

  and had needed to be retrieved from the data nodes

  to be interpreted by the Dydra query processor,
- is now analyzed by NDB IC within the RonDB cluster already.

IMPROVEMENTS FOR DYDRA

- Dydra can now run complex SPARQL queries more effectively.
- It improves running real queries, for example,
  a query to explore variant clinical significance[2],
  run against a database containing
  the human reference genome GRCh38 of TogoVar [Mitsuhashi2022TogoVar]
  as well as the ClinVAR (Clinical significance of variants) dataset.
- We can run queries now, that we simple could not two years ago.

[2] available online at https://github.com/togodx/togodx-sparqlet/blob/master/prod/variant_clinical_significance_togovar.md

OUTLOOK

- The evaluation was single host only,
  while the work presented should really make a difference
  when running on a big cluster and networking is involved.
- The write path when adding new content to a repository
  could now also be improved and simplified at the same time.
- E-Mail from Mikael Ronström on May 7, 2026:
  He made prototype of a compiler to turn NDB IC into machine code.

  A test shows that an interpreted program of 30 ops takes
  around 1 microsecond to compile on x86 and 3-5 micros on ARM64.

  The actual execution takes around 10ns, most likely a speed up of around 5x.

ACKNOWLEDGMENTS

- To Mikael Ronström, Hopsworks AB, and
  James Anderson, Dataraph GmbH,
  for their work, contributions and many discussions,
- to Moritz Hassert for comments, and
- Thorsten Schmidt for corrections.

BACKUP SLIDES

COMMON LISP MACROS

- Writing a *simple* compiler is embarrassingly easy in Common Lisp,
- as the language already provides everything you need:
- a code transformation machinery to extend the language.

READING THE CODE

- If you are willing to stick with Lisp syntax, or rather its *non-syntax*,
- then you can simply use the existing Lisp parser, called the *reader*,
- and only need to write the transformations and code generation.

CODE GENERATION

The C++ class `NdbInterpretedCode` contains everything:

- to create the NDB IC code objects,
- to add instructions to it and
- resolve internal branches and jumps to labels and subroutine offsets,

and `cl-ndbapi` implements access to it in the package `NDBAPI.IC`.

ic.i::test-to-branch

```
(defmacro test-to-branch (test rest label)
  (ccase test
     (<  `(transform-second-arg *code* ic:branch-ge ,rest ,label))
     (>  `(transform-second-arg *code* ic:branch-le ,rest ,label))
     (>= `(transform-second-arg *code* ic:branch-lt ,rest ,label))
     (<= `(transform-second-arg *code* ic:branch-gt ,rest ,label))
     (=  `(transform-second-arg *code* ic:branch-ne ,rest ,label))
     (/= `(transform-second-arg *code* ic:branch-eq ,rest ,label))
     (ic:zerop
      `(transform-first-arg *code* ic:branch-ne-zero ,rest ,label))
     (ic:onep
      `(transform-first-arg *code* ic:branch-ne-one  ,rest ,label))
     (ic:twop
      `(transform-first-arg *code* ic:branch-ne-two  ,rest ,label))
     (ic:null
      `(transform-first-arg *code* ic:branch-ne-null ,rest ,label))))
```

transform-first/second-arg

```
(defmacro transform-first-arg (code branch rest label)
  (destructuring-bind (car . cdr) rest
    (if (consp car)
        ;; if car is a form, move it before the branch
        ;; form and extract its reg-dest
        `(progn
           ,car
           (,branch ,code ,(caddr car) ,label ,@cdr))
        `(,branch ,code ,car ,label ,@cdr))))

(defmacro transform-second-arg (code branch rest label)
  (destructuring-bind (first second . rest) rest
    (if (consp second)
        ;; if second is a form, move it before the branch
        ;; form and extract its reg-dest
        `(progn
           ,second
           (,branch ,code ,first ,(caddr second) ,label ,@rest))
        `(,branch ,code ,first ,second ,label ,@rest))))
```