DOCUMENT INFORMATION

TITLE	**Efficient range estimation with 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 11th, 2026, 11:30–12:00
PAPER	Max-Gerd Retzlaff (2026). Efficient range estimation with 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.20288900
PUBL. LINK	https://matroid.org/id/29559dff-e465-4de5-88a7-fab9375fc9d2
LICENSE	
COPYRIGHT ©	

EFFICIENT RANGE ESTIMATION WITH NDB INTERPRETED CODE

Max-Gerd Retzlaff, Datagraph GmbH

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

Publication location: http://www.matroid.org/id/29559dff-e465-4de5-88a7-fab9375fc9d2

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

NEW RONDB CLUSTER BACKEND FOR DYDRA

- (needs to store statements in RonDB from Lisp)
- needs to scan statements in RonDB from Lisp
- needs to know cardinalities of graph patterns to steer query optimization in SPOCQ

$\Rightarrow$ efficient range estimation necessary

RESOURCE DESCRIPTION FRAMEWORK (RDF)

- extremely flexible data model
- with numerous serializations
- and a powerful query language

The RDF specification consists of a whole suite of W3C Recommendations and Working Group Notes.

RDF DATA MODEL

- RDF dataset: collection of RDF graphs
- RDF graph: set of RDF triples
- RDF triple: three RDF terms: subject–predicate–object
- RDF term: IRIs, blank nodes, or datatyped literals

RDF TRIPLE / RDF STATEMENT

- subject–predicate–object
- in principle, simple facts or statements:
  - “Bob is 35”
  - “Joe sells books”
  - “Bob knows Fred”

EXAMPLE RDF GRAPH

- in practice, a bit more structured:

```
@prefix  dc:  <http://purl.org/dc/elements/1.1/> .
@prefix  :    <http://example.org/book/> .
@prefix  ns:  <http://example.org/ns#> .

:book1  dc:title     "SPARQL Tutorial" .
:book1  ns:price     42 .
:book1  ns:discount  0.2 .

:book2  dc:title     "The Semantic Web" .
:book2  ns:price     23 .
:book2  ns:discount  0.25 .
```

format: RDF 1.1 Turtle (“Terse RDF Triple Language”)

SPARQL QUERY

```
:book1  dc:title     "SPARQL Tutorial" .
:book1  ns:price     42 .
:book1  ns:discount  0.2 .

:book2  dc:title     "The Semantic Web" .
:book2  ns:price     23 .
:book2  ns:discount  0.25 .
```

```
@prefix  dc:  <http://purl.org/dc/elements/1.1/> .
@prefix  ns:  <http://example.org/ns#> .

SELECT  ?title ?price
WHERE {
  ?x ns:price ?price .
  ?x dc:title ?title .
}
```

SPARQL QUERY

```
:book1  dc:title     "SPARQL Tutorial" .
:book1  ns:price     42 .
:book1  ns:discount  0.2 .

:book2  dc:title     "The Semantic Web" .
:book2  ns:price     23 .
:book2  ns:discount  0.25 .
```

```
@prefix  dc:  <http://purl.org/dc/elements/1.1/> .
@prefix  ns:  <http://example.org/ns#> .

SELECT  ?title ?price
WHERE {
  ?x ns:price ?price .
  ?x dc:title ?title .
}
```

title	price
"The Semantic Web"	23
"SPARQL Tutorial"	42

SPARQL QUERY #2

```
@prefix  dc:  <http://purl.org/dc/elements/1.1/> .
@prefix  ns:  <http://example.org/ns#> .

SELECT  ?title ?price ?discount
        (?price*(1-?discount) AS ?reduced_price)
{ ?x ns:price ?price .
  ?x dc:title ?title .
  ?x ns:discount ?discount
}
```

title	price	discount	reduced_price
"The Semantic Web"	23	0.25	17.25
"SPARQL Tutorial"	42	0.2	33.6

SPARQL QUERY AGAINST DBPEDIA

```
@prefix  rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix  dbo:   <http://dbpedia.org/ontology/>  .
@prefix  dbp:   <http://dbpedia.org/property/> .
@prefix  wn20:  <http://www.w3.org/2006/03/wn/wn20/instances/> .

SELECT DISTINCT ?name ?death ?cause
WHERE { ?person rdf:type dbo:Person .
        ?person foaf:name ?name .
        ?person dbp:wordnet_type wn20:synset-musician-noun-1 .
        ?person dbo:deathDate ?death .
        FILTER (?death > "1980-01-01"^^xsd:date) .
        OPTIONAL {?person dbp:deathCause ?cause} . }
```

SPARQL QUERY AGAINST DBPEDIA

```
@prefix  rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix  dbo:   <http://dbpedia.org/ontology/>  .
@prefix  dbp:   <http://dbpedia.org/property/> .
@prefix  wn20:  <http://www.w3.org/2006/03/wn/wn20/instances/> .

SELECT DISTINCT ?name ?death ?cause
WHERE { ?person rdf:type dbo:Person .
        ?person foaf:name ?name .
        ?person dbp:wordnet_type wn20:synset-musician-noun-1 .
        ?person dbo:deathDate ?death .
        FILTER (?death > "1980-01-01"^^xsd:date) .
        OPTIONAL {?person dbp:deathCause ?cause} . }
```

name	death	cause
"Prince"@en	2016-04-21	"Accidental fentanyl overdose"@en
"James Brown"@en	2006-12-25	
"John Lennon"@en	1980-12-08	<http://dbpedia.org/resource/Murder_of_John_Lennon>

EXAMPLE RDF DATASET

```
# Default graph
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example.org/bob>    dc:publisher  "Bob" .
<http://example.org/alice>  dc:publisher  "Alice" .
```

```
# Named graph: http://example.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Bob" .
_:a foaf:mbox <mailto:bob@oldcorp.example.org> .
```

```
# Named graph: http://example.org/alice
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example.org> .
```

format: RDF 1.1 Turtle

EXAMPLE RDF DATASET

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://example.org/bob>    dc:publisher  "Bob" .
<http://example.org/alice>  dc:publisher  "Alice" .

<http://example.org/bob> {
  _:a foaf:name "Bob" .
  _:a foaf:mbox <mailto:bob@oldcorp.example.org> .
}

<http://example.org/alice> {
  _:a foaf:name "Alice" .
  _:a foaf:mbox <mailto:alice@work.example.org> .
}
```

format: RDF 1.1 TriG (“Triples Graphs”)

EXAMPLE RDF DATASET

```
<http://example.org/bob>
 ↪ <http://purl.org/dc/elements/1.1/publisher>  "Bob" .
<http://example.org/alice>
 ↪ <http://purl.org/dc/elements/1.1/publisher>  "Alice" .

_:a <http://xmlns.com/foaf/0.1/name> "Bob"
 ↪ <http://example.org/bob> .
_:a <http://xmlns.com/foaf/0.1/mbox> <mailto:bob@oldcorp.example.org>
 ↪ <http://example.org/bob> .

_:a <http://xmlns.com/foaf/0.1/name> "Alice"
 ↪ <http://example.org/alice> .
_:a <http://xmlns.com/foaf/0.1/mbox> <mailto:alice@work.example.org>
 ↪ <http://example.org/alice> .
```

format: RDF 1.1 N-Quads

SPOG / SPOC

- subject–predicate–object–graph
- subject–predicate–object–context

SIMPLE RDF STORE IN SQL

A simple triple store graph repository without revisions could be created with this SQL call:

```
create table test
  (g int unsigned not null, s int unsigned not null,
   p int unsigned not null, o int unsigned not null,
   index gspo (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),
   unique (g,s,p,o));
```

SIMPLE RDF STORE IN SQL

As an optimization we can turn the first index into a primary index and get rid of the extra unique definition:

```
1 create table test
2   (g int unsigned not null, s int unsigned not null,
3    p int unsigned not null, o int unsigned not null,
4    primary key (g,s,p,o), index gpos (g,p,o,s),
5    index gosp (g,o,s,p), index spog (s,p,o,g),
6    index posg (p,o,s,g), index ospg (o,s,p,g));
```

SPARQL QUERIES TURN TO INDEX SCANS

```
select ?s ?p ?o
where {
  graph <http://data.dydra.com/graph/plm/typeSystem/1> {
    ?s ?p ?o
  }
}
```

- Dydra interns all terms as term identifier numbers (term IDs)
- Here, the term <http://data.dydra.com/graph/plm/typeSystem/1> is interned as ID `54316`.
- The SPARQL query
  will result scan for the quad pattern `(54316 0 0 0)`
  against the index `(0 1 2 3)`, which is the primary index
  also called GSPO, that is, its order is graph, subject, predicate, object.

SPARQL QUERIES WITH A GRAPH CLAUSE

- In SPOCQ, the SPARQL query

```
select ?s ?p ?o
where {
  graph <http://data.dydra.com/graph/plm/typeSystem/1> {
    ?s ?p ?o
  }
}
```

translates to an NDB API call `NdbTransaction::scanIndex()`

- with the `SF-ORDER-BY` flag set to true and
- one `NdbIndexScanOperation::IndexBound`:

```
bound 0:
  low-key:  (54316 0 0 0), low-key-count:  1, low-inclusive:  t,
  high-key: (54316 0 0 0), high-key-count: 1, high-inclusive: t,
  range-no: 0
```

SPARQL QUERY AGAINST THE DEFAULT GRAPH

- A SPARQL query without a graph clause will use the default graph:

```
1 select ?s ?p ?o
2 where { ?s ?p ?o }
```

- In SPOCQ, the default graph has the term ID `-1`,
  in the two's complement: `#xFFFFFFFF` or `4294967295` (decimal).
- Results in a scan for the quad pattern `(4294967295 0 0 0)`
  against the index `(0 1 2 3)`, which is again the primary index.

⇒ The same scan as with an explicit graph term.

SPARQL QUERY WITH GIVEN GRAPH AND PREDICATE

```
select ?s ?o
where {
  graph <http://data.dydra.com/graph/plm/typeSystem/1> {
    ?s a ?o
  }
}
```

- Adding a predicated, in this case the keyword “a”
  which is the shorthand for the RDF IRI
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, term ID 1:
- Results in a scan for the quad pattern `(54316 0 1 0)`
  against the index `(0 2 3 1)`, which is the GPOS index.

```
bound 0:
  low-key:  (54316 1 0 0), low-key-count:  1, low-inclusive:  t,
  high-key: (54316 1 0 0), high-key-count: 1, high-inclusive: t,
  range-no: 0
```

SPARQL QUERY WITH GIVEN GRAPH AND PREDICATE

```
select ?s ?o
where {
  graph <http://data.dydra.com/graph/plm/typeSystem/1> {
    ?s a ?o
  }
}
```

- Adding a predicated, in this case the keyword “a”
  which is the shorthand for the RDF IRI
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, term ID 1:
- Results in a scan for the quad pattern `(54316 0 1 0)`
  against the index `(0 2 3 1)`, which is the GPOS index.

```
bound 0:
  low-key:  (54316 1 0 0), low-key-count:  1, low-inclusive:  t,
  high-key: (54316 1 0 0), high-key-count: 1, high-inclusive: t,
  range-no: 0
```

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

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

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.

⇒ *More on that in my second presentation.*

A MINIMAL NDB INTERPRETED CODE PROGRAM

```
1 (ndbapi.ic:with-code (code (cffi:null-pointer))
2   (ndbapi.ic:interpret-exit-last-row code)
3   (ndbapi.ic:finalise code)
4   ;; ... use code ...
5   )
```

- `interpret-exit-last-row` indicates that:
  - the current row is to be returned as part of the scan and
  - that no more rows should be returned.
- `finalise` finishes the interpreted program:
  - resolves all jumps, branch and subroutine calls
  - is not an instruction that is going to be part of the program

$\Rightarrow$ only one instruction in total

ROW COUNT OF A TABLE

Why would such a program be useful?

- instead of the actual columns use *pseudo columns*,
- retrieve as an attribute value using `NdbOperation::getValue()`

One such candidate is:

> *ROW_COUNT*
>
> *This returns the current number of rows in the partition.*
> *A row is seen in this view when the row insert has committed.*
> *It is a 64-bit column.*

⇒ The ndb tool `ndb_select_count` does exactly that
to determine “the number of rows in one or more NDB tables”.

RANGE ESTIMATION

For Dydra we need three things differently:

1. scan against an index not the full table,
2. allow for bounded scans, and
3. use the newer `Record` interface of the NDB API and not the older `RecAttr` interface.

ROW COUNTS OF BOUNDED INDEX SCANS

- instead of `NdbOperation::getValue()`
  use the `NdbOperation::OperationOptions` option `extraGetValues`
- instead of `ROW_COUNT`
  use the 2nd value of `RECORDS_IN_RANGE`,
  which contained the number of rows in the range:

  RECORDS_IN_RANGE

  When executing an ordered index scan this column will calculate an estimate of how many rows exist in the range specified in the scan. The intended use for this is to calculate index statistics used to optimize queries in the MySQL Server. The column type is an array of 4 32-bit values.

  The first value is the number of entries in the index (all rows in the partition except rows that have a NULL value in an index column). The second is the number of rows in the range, the third value is the number of rows before the range and the last value is the number of rows after the range. All these values are estimates.

WHY IS `records_in_range` AN ESTIMATE?

- `RECORDS_IN_RANGE` estimates rows in a range based on counts from tree properties on each index fragment.
- Errors come from imperfectly balanced tree and
  from uncommitted entries.
- In case the range fits inside a single tree node, the number of rows returned is actually exact for that fragment.
  But if the range spans multiple tree nodes, it is calculated based on values that are only correct for perfectly balanced trees.

⇒ tolerable for us, as we just want to steer SPOCQ's query optimization

ZERO IS EXACT

Important property of `RECORDS_IN_RANGE`:

- A returned zero for the number of rows is exact.
- Zero is only returned when a range is actually empty,
- otherwise, for each fragement at least one is returned.

⇒ SPOCQ can eleminate parts of the query during query optimization when the estimation returned zero for a graph pattern.

ndbapi-scan-count.lisp

```
1  (in-package :ndb.scan-count)
2  (defun scan-count (&key connection-string database-name table-name index-name
3                          bound-specs ;; list of specs: (&key low high (low-inclusive t) (
   ;;; high-inclusive t))
4                          (lock-mode :+lm-committed-read+))
5    "WARNING: scan-count just gives an estimation of the matching rows.
6    For an exact count call simple-scan with :just-count t."
7
8  ;;; Ensure that the NDB API library has been loaded and ndb_init has been called, make an
   ;;; ndb-cluster-connection and make an ndb object, which is bound to "ndb" for the context of
   ;;;  the ndbapi:with-ndb form:
9    (ndbapi:ensure-ndb-api-initialisation)
10   (ndbapi:with-ndb-cluster-connection (ndbapi:*connection* (connection-string) :name "
   ;;; ndbapi-scan-count")
11     (ndbapi:with-ndb (ndb (ndbapi:*connection* database-name))
12
13 ;;; Create the interpreted code object:
14       (ndbapi.ic:with-code (code (cffi:null-pointer))
15         (ndbapi.ic:interpret-exit-last-row code)
16         (ndbapi.ic:finalise code)
```

https://github.com/datagraph/cl-ndbapi/blob/main/examples/ndbapi-scan-count.lisp

```
;;; Create the transaction and retrieve some objects:
      (ndbapi:with-ndb-transaction (transaction ndb)
        (let* ((dict (ndbapi:ndb-get-dictionary ndb))
               (table (ndbapi:dictionary-get-table dict table-name))
               (index (ndbapi:dictionary-get-index dict index-name table-name))
               (index-default-record (ndbapi:index-get-default-record index))
               (table-default-record (ndbapi:table-get-default-record table)))

;;; Allocate some memory to hold the result of records-in-range, allocate struct
;;; get-value-spec, introduce the local variable "extra-get-values" for it and make it to
;;; retrieve the column "records-in-range":
          (cffi:with-foreign-object (records-in-range-ptr :uint32 4)
            (ndbapi:with-foreign-struct
             (extra-get-values (list :column (ndbapi:column-records-in-range)
                                     :app-storage records-in-range-ptr
                                     :rec-attr (cffi:null-pointer)
                                     ;; the next two initializations are strictly
                                     ;; necessary starting with rondb 24.10
                                     :m-start-pos 0 :m-size 0)
                               '(:struct ndbapi:get-value-spec))
```

```
;;; Allocate a struct scan-options, call it scan-options with scan options so-scanflags,
;;; so-getvalue and so-interpreted and set those options accordingly:
                (ndbapi:with-foreign-struct
                 (scan-options (list :options-present
                                     '(:+so-scanflags+ :+so-getvalue+ :+so-interpreted+)
                                     ;; allow multiple bounds
                                     :scan-flags '(:+sf-multi-range+)
                                     :extra-get-values extra-get-values
                                     :num-extra-get-values 1
                                     :interpreted-code (ndbapi:foreign-pointer code))
                               '(:struct ndbapi:scan-options))

;;; Allocate a result-mask that will not retrieve any columns:
                  (cffi:with-foreign-object (result-mask :unsigned-char)
                    (setf (cffi:mem-ref result-mask :unsigned-char) #b00000000)

;;; Call scanIndex() and name the returned scan index operation "scan":
                    (ndbapi:with-ndb-transaction-scan-index
                        (scan (transaction index-default-record
                                           table-default-record
                                           lock-mode
                                           result-mask
                                           (cffi:null-pointer)
                                           scan-options
                                           (cffi:foreign-type-size
                                            '(:struct ndbapi:scan-options)))
                              (t))
```

```
;;; Set the bounds for the bounded scans for the quad pattern, as a bound specification of
;;; form (&key low high (low-inclusive t) (high-inclusive t)):
                    (let ((bounds-count (length bound-specs)))
                      (cffi:with-foreign-objects
                          ((low-quads '(:struct ndb.quads:quad) bounds-count)
                           (high-quads '(:struct ndb.quads:quad) bounds-count)
                           (bounds '(:struct ndbapi:index-bound) bounds-count))
                        (loop for bound-spec in bound-specs   for i from 0 do
                              (destructuring-bind (&key low high (low-inclusive t)
                                                        (high-inclusive t))
                                  bound-spec
                                (setf
                                 (cffi:mem-aref low-quads '(:struct ndb.quads:quad) i)
                                 (ndb.quads:list-to-quad low)
                                 (cffi:mem-aref high-quads '(:struct ndb.quads:quad) i)
                                 (ndb.quads:list-to-quad high)
                                 (cffi:mem-aref bounds '(:struct ndbapi:index-bound) i)
                                 (list :low-key (cffi:mem-aptr low-quads
                                                               '(:struct ndb.quads:quad) i)
                                       :low-key-count (length low)
                                       :low-inclusive low-inclusive
                                       :high-key (cffi:mem-aptr high-quads
                                                                '(:struct ndb.quads:quad) i)
                                       :high-key-count (length high)
                                       :high-inclusive high-inclusive
                                       :range-no i))
                                (let ((bound (cffi:mem-aptr bounds
                                                            '(:struct ndbapi:index-bound) i)))
                                  (ndbapi:ndb-index-scan-operation-set-bound
                                   scan index-default-record bound))))))
```

```
94  ;;; Execute the transaction and thus do the index scan:
95                            (ndbapi:ndb-transaction-execute transaction :+NO-COMMIT+)))
96
97  ;;; Introduce a local variable total-row-count, allocate a pointer "row-data" to point at the
    ;;;  current row and collect results:
98                        (let ((total-row-count 0))
99                          (cffi:with-foreign-object (row-data :pointer)
100                           (loop ;; We have to loop as there will be one scan per partition.
101                                 ;; Call nextResult with options fetchAllowed set to "true"
102                                 ;; and forceSend to "false":
103                                 for rc = (ndbapi:ndb-scan-operation-next-result
104                                           scan row-data t nil)
105                                 while (zerop rc) ;; return code 0 means:
106                                                  ;;  "Another tuple has been received"
107                                 ;; Access the range-count at second position:
108                                 for range-count = (cffi:mem-aref records-in-range-ptr
109                                                                  :uint32 1)
110                                 ;; Add up the number of rows in the current partition:
111                                 do (incf total-row-count range-count)
112                                 ;; Check at the end that all is well:
113                                 finally (assert (= rc 1) () ;; return code 1 means:
114                                                 ;;   "There are no more tuples to scan".
115                                                 "scan-operation-next-result() failed: ~a"
116                                                 (ndbapi:get-ndb-error transaction
117                                                  #'ndbapi:ndb-transaction-get-ndb-error))))
118 ;;; Return the total count:
119                         total-row-count)))))))))))
```

RUN TIMES OF RANGE ESTIMATION QUERIES

Comparison of:

a) count estimation with NDB IC, `scan-count`
b) count via full index scan, implementated as `simple-scan`, still does not transfer any actual column data from the RonDB data node to the SPOCQ API node but counts each individual row.

Implementations:

- https://github.com/datagraph/cl-ndbapi/blob/main/examples/ndbapi-scan-count.lisp
- https://github.com/datagraph/cl-ndbapi/blob/main/examples/ndbapi-simple-scan.lisp

EVALUATION #1

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.

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.

EVALUATION #1

Count all quads with the predicate “a”, or RDF IRI <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, term ID 1 in Dydra.

a) count estimation:

```
* (time (ndb.scan-count:scan-count
          :database-name "redacted/togovar-rondb-subject"
          :table-name "quads"
          :index-name "psog"
          :bound-specs '((:low (1) :low-inclusive t
          :high (1) :high-inclusive t))))
Evaluation took:
  0.004 seconds of real time
  0.001416 seconds of total run time (0.001416 user, 0.000000 system)
  25.00% CPU
  6,708,350 processor cycles
  2 page faults
  0 bytes consed
292056083
```

EVALUATION #1

b) full count of the same quads:

```
* (time (ndb.simple-scan:simple-scan :just-count t
         :database-name "redacted/togovar-rondb-subject"
         :table-name "quads"
         :index-name "psog"
         :bound-specs '((:low (1) :low-inclusive t
         :high (1) :high-inclusive t))))
Evaluation took:
  129.628 seconds of real time
  21.025530 seconds of total run time (19.667508 user, 1.358022
    system)
  16.22% CPU
  323,413,452,025 processor cycles
  195,840 bytes consed
331013545
```

Result: 129.628 vs. 0.004 seconds. The estimation is 32,000 times faster, while the result is off by 38,957,462 entries, which is 11.8 %.

EVALUATION #2

Dataset:

- the more typical production database “plm-stats”
  with 4.5 million quads copied over to one of our development servers.

Computer:

- one of our development servers

EVALUATION #2

Again count all quads with the predicate “a”, but we use the index POSG, as this database happens to have a different set of indexes defined.

a) count estimation:

```
* (time (ndb.scan-count:scan-count
          :database-name "mgr/plm-stats"
          :table-name "quads" :index-name "posg"
          :bound-specs '((:low (1) :low-inclusive t
          :high (1) :high-inclusive t))))
Evaluation took:
  0.003 seconds of real time
  0.001207 seconds of total run time (0.000497 user, 0.000710 system)
  33.33% CPU
  6,280,160 processor cycles
  32,640 bytes consed
591102
```

EVALUATION #2

b) full count of the same quads:

```
* (time (ndb.simple-scan:simple-scan :just-count 1
          :database-name "mgr/plm-stats"
          :table-name "quads" :index-name "posg"
          :bound-specs '((:low (1) :low-inclusive t
          :high (1) :high-inclusive t))))
Evaluation took:
  0.183 seconds of real time
  0.129964 seconds of total run time (0.129964 user, 0.000000 system)
  71.04% CPU
  441,008,240 processor cycles
  0 bytes consed
1110182
```

Result: 0.183 seconds vs. 0.003 seconds. the estimation is 61 times faster, while the result is off by 519,080 quads, which is 47 %.

The error is quite high, but it is still good enough for our purpose.

CONCLUSION

- This paper presents the implementation of an efficient range estimation based on NDB interpreted code.
- Even on a large database with 3.3 billion statements it runs in constant time and returns almost instantly.
- It depends just on the number of partitions, which is constant, and not on the number of statements stored in the database.
  $\Rightarrow$ The estimation's bound of growth can be expressed as $O(1)$ in the number of statements, in Bachmann-Landau notation [Bachmann 1894, Landau 1909], commonly referred to as the Big *O* notation.
- This estimation is used to steer the query optimization of the SPARQL processor SPOCQ when the Dydra graph store is using the new RonDB backend.

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

EXAMPLE RDF GRAPH

- in practice, a bit more structured:

```
<http://example.org/book/book1>
 ↪ <http://purl.org/dc/elements/1.1/title> "SPARQL Tutorial" .
<http://example.org/book/book1>
 ↪ <http://example.org/ns#price> 42 .
<http://example.org/book/book1>
 ↪ <http://example.org/ns#discount> 0.2 .

<http://example.org/book/book2>
 ↪ <http://purl.org/dc/elements/1.1/title> "The Semantic Web" .
<http://example.org/book/book2>
 ↪ <http://example.org/ns#price> 23 .
<http://example.org/book/book2>
 ↪ <http://example.org/ns#discount> 0.25 .
```

format: RDF 1.1 N-Triples

ESSENTIAL PATTERNS

Category	Patterns	Count
Singles	G, S, P, O	4
Pairs	GS, GP, GO, SP, SO, PO	6
Triples	GSP, GSO, GPO, SPO	4
Quads	GSPO	1
Total		15

▸ back

COVERAGE OF PATTERNS BY INDEXES

Pattern (Fixed)	Covered By (Prefix)
G	GPSO, GSPO
S	SOPG
P	POGS, PSOG
O	OGSP
GP	GPSO
GS	GSPO
GO	OGSP (via OG prefix)
SP	PSOG (via PS prefix)

Pattern (Fixed)	Covered By (Prefix)
SO	SOPG
PO	POGS
GSP	GSPO
GPO	POGS (via POG prefix)
GSO	SOPG (via SOP prefix)
SPO	PSOG (via PSO prefix)
GSPO	Any (Full key match)

SIX INDEXES - OUR STANDARD SELECTION

- GSPO: Covers patterns starting with G, GS, GSP, and GSPO.
- GPOS: Covers patterns starting with GP, GPO, and GPOS.
- GOSP: Covers patterns starting with GO, GOS, and GOSP.
- SPOG: Covers patterns starting with S, SP, SPO, and SPOG.
- POSG: Covers patterns starting with P, PO, POS, and POSG.
- OSPG: Covers patterns starting with O, OS, OSP, and OSPG.

SIX INDEXES - SUPPORT STAR PATTERNS

- GPSO: Covers patterns starting with GP, GPS, and GPSO.
- GSPO: Covers patterns starting with G, GS, GSP, and GSPO.
- OGSP: Covers patterns starting with O, OG, OGS, and OGSP.
- POGS: Covers patterns starting with PO, POG, and POGS.
- PSOG: Covers patterns starting with P, PS, PSO, and PSOG.
- SOPG: Covers patterns starting with S, SO, SOP, and SOPG.