Linköping Electronic Articles in
Computer and Information Science
Vol. 4(1999):nr 1

QDB - A Query Processor for the High Performance, Parallel Data Server NDB Cluster

Martin Sköld

Department of Computer and Information Science
Linköpings Universitet
Linköping, Sweden

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/1999/001/

Published on April/7/1999 by
Linköping University Electronic Press
581 83 Linköping, Sweden

Linköping Electronic Articles in
Computer and Information Science
ISSN 1401-9841
Series Editor: Erik Sandewall

© 1999 Martin Sköld
Typeset by the author using FrameMaker

Recommended citation

Martin Sköld. QDB - A Query Processor for the High Performance, Parallel Data Server NDB Cluster. Linköping electronic articles in computer and information science, Vol 4(1999): nr 1. http://www.ep.liu.se/ea/cis/1999/001. April/7/1999.

This URL will also contain a link to the author's home page.

The publishers will keep this article on-line on the Internet (or its possible replacement network in the future) for a period of 25 years from the date of publication, barring exceptional circumstances as described separately.

The on-line availability of the article implies a permanent permission for anyone to read the article on-line, to print out single copies of it, and to use it unchanged for any non-commercial research and educational purpose, including making copies for classroom use.
This permission can not be revoked by subsequent transfers of copyright. All other uses of the article are conditional on the consent of the copyright owner.

The publication of the article on the date stated above included also the production of a limited number of copies on paper, which were archived in Swedish university libraries like all other written works published in Sweden.
The publisher has taken technical and administrative measures to assure that the on-line version of the article will be permanently accessible using the URL stated above, unchanged, and permanently equal to the archived printed copies at least until the expiration of the publication period.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/ *or by conventional mail to the address stated above.*

Abstract

This report describes a query language front-end developed for the data server NDB Cluster. NDB Cluster is a high-performance, parallel data server that is being developed by Ericsson for telecom applications.

The query processor QDB is an extension of the AMOS DBMS with an interface to NDB Cluster. The query language of AMOS, AMOSQL, supports an Object-Relational data model and makes it possible to support both a relational model of data in NDB Cluster, as well as an Object-Oriented (OO) data model.

NDB Cluster supports a flat relational data model and QDB supports transparent translation of OO queries to flat relations. QDB utilizes the foreign function mechanism of AMOS which makes it possible to integrate foreign data sources transparently and with an extensible query optimizer.

The report describes the translation process, query processing and optimization of QDB queries, and details about the interface between the systems.

Introduction

In the telecom area, databases are increasingly being used in the operation of telecom networks and as part of telecom applications. Database systems are needed in many parts of telecommunication networks such as in managing subscriber data, billing, and in network management. In the past, databases have been used off-line for most of these applications, but here we are considering on-line databases systems. These database systems will be tightly integrated into the network architecture and are here called *network databases*[1]. The first network databases were Service Control Points (SCP) which provided mostly number translations for various services. Also number translation databases for mobile telecommunications (Home Location Registers, HLR) were early starters. SCPs and HLRs are now becoming the platform for the execution of telecommunication services. SCPs and HLRs could also be envisioned to include event monitoring of the telecom network, both for on-line charging services and for various network management services (see Fig. 1).

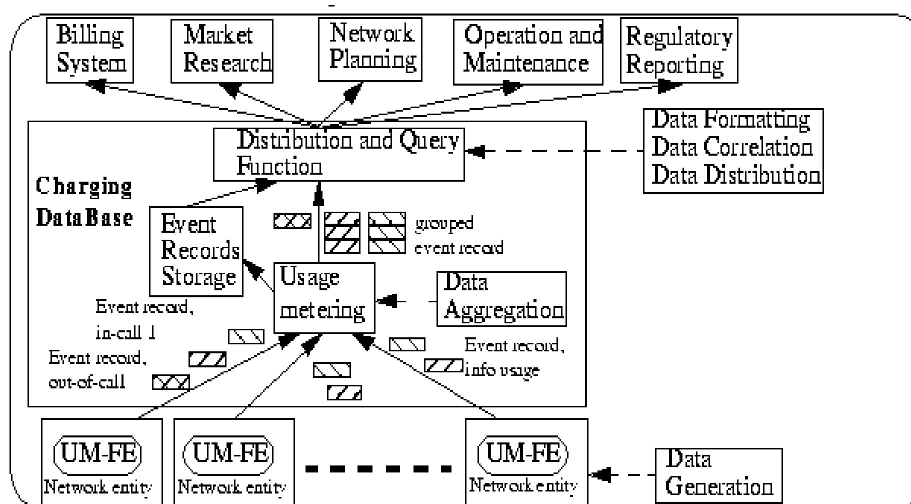

Figure 1: Databases in Telecommunication Networks

UMTS (Universal Mobile Telecommunications System) is defined to be a next-generation mobile telecommunications system and aims to integrate message services, multimedia services, and broadband (up to 2 Mbit/sec) accesses even in mobile networks. This means that it will be necessary to have access to Mail Servers and various types of Information Servers in the telecommunication network. All these servers require network data-

1. Not to be confused with database systems based on the network data model (a predecessor to the relational data model) in which relationships are represented by a network of links.

bases. They have the common characteristics that they have to answer massive amounts of rather simple queries with predictable and short response times. Sometimes very large object are returned. Furthermore, there is also need for complex queries which are issued less frequently, e.g. for complex billing schemes and for network monitoring functions. Most the network databases have extremely high reliability requirements. Both software and hardware changes are necessary to cope with without interrupting the system.

There is an on-going project at Ericsson [8] to develop a reliable and high-performing database system targeted for use in telecom systems. The project, called NDB Cluster (Network DataBase Cluster), comprises novel work in fast network data servers. The need for high-performance database systems in future telecom networks has spawned a number of research projects with the aim of developing high-performance main-memory database systems for supporting functionality and applications in future broad-band telecommunications networks. Examples of such systems include DataBlitz [1] by Lucent and TimesTen [10], a spin-off from HP-Labs. Ericssons NDB Cluster is based on parallel processing using a distributed system architecture based on a high-speed communication and special distributed data structures. This architecture provides better reliability than both DataBlitz and TimesTen, without sacrificing performance. NDB Cluster does not have a query language interface and the purpose of this project was to study requirements and develop a query interface for NDB Cluster.

AMOS

AMOS [3] stands for Active Mediating Object System and is an Object-Relational main-memory DBMS. Active symbolizes that AMOS is an active DBMS [9] with support for monitoring changes to data using active rules. Mediating symbolizes that AMOS is a multi-database DBMS with support for accessing foreign data sources, both other database systems and other data sources such as files or sensor data. AMOS is an open database architecture with support for integrating applications and accessing foreign data sources.
The extensibility of the architecture stems from an extensible storage manager that supports the introduction of new data structures and an extensible query language and query optimizer that supports transparent access to application data and foreign data sources through foreign functions, imported object classes and proxy objects. Foreign functions can be used for accessing foreign relational tables and table attributes. Imported classes and proxy objects can give access to foreign Object-Oriented data and also for extending a foreign relational data source with an OO data model, which is done in QDB. In QDB, AMOS has been extended with

NDB Cluster as a foreign DBMS that can be accessed more or less transparently through AMOSQL, the query language of AMOS.

NDB Cluster and QDB

NDB Cluster (hereafter referred to as just NDB) is a distributed database architecture and the integration of AMOS as a query front-end was done by letting AMOS (hereafter referred to as QDB) become one node in the distributed architecture (see Fig. 2)..

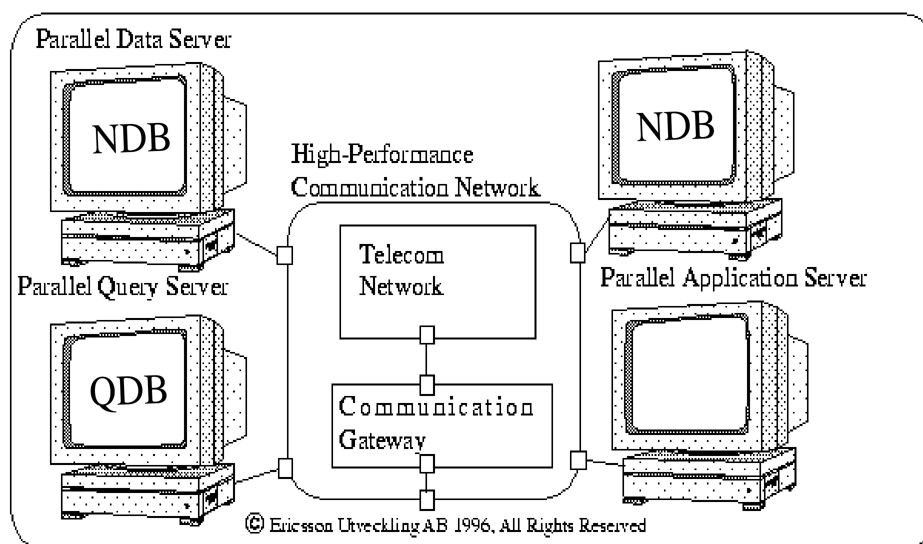

Figure 2: The QDB/NDB Cluster Client-Server Architecture

This was done using the NDB Application Programming Interface (NDB-API) which is a low level client-server interface for accessing NDB from C++. The NDB-API isolates the application (and QDB) from the server in order not to compromise the reliability of the server. The communication over the NDB-API can be done using TCP/IP (mostly for developing purposes) or using SCI [4], a high-performance interconnect protocol based on shared memory and special hardware[1] (a more likely solution for real-time client-server communication). Which communication protocol that is chosen is totally transparent for applications using the NDB-API Writing applications using the low-level NDB-API is very tedious and requires recompilation of the application every time you make a change. By integrating the NDB-API into QDB, no recompilation is required since the interaction using AMOSQL is incremental. Stating complex queries in AMOSQL is fairly easy, but can be very complicated using just the NDB-API. In QDB the query optimizer makes it easier for the user writing que-

1.NDB Cluster also supports communication based on shared memory managed by the operating system.

ries since the way the query is written (like considering the order accessing tables and using available indexes) is more or less unimportant. Such choices are made by the AMOSQL query optimizer when the actual *query plan* is chosen. AMOSQL views (called derived functions) can also be *reoptimized* in the case of changes to the size of data or changes to the cost model. In direct access to the NDB-API complex queries have to be hand-optimized and can never be reoptimized (without rewriting the application code).

Schema Definition in QDB

In QDB the user can directly define and access a purely relational data model stored in NDB or an Object-Oriented data model where QDB translates schema definitions and queries from the Object-Oriented model to a flat relational data model stored in NDB.

Relational Schemas in QDB

Relational tables and views can be created as special AMOSQL functions[1].

```
create function
subscriber(snb charstring key,
           gname charstring, /* given name */
           fname charstring, /* family name */
           address charstring) -> boolean
as foreign "ndb-table";
```

This creates a table with four attributes and where the string `snb` is primary key. In the current implementation the literal data types `charstring` is mapped to C/C++ `char*`, `integer` is mapped to C/C++ `unsigned int`, and `real` is mapped to C/C++ `double`. Extensions with any additional literal types that NDB can store can be made fairly easily since QDB has an extensible storage manager.

The table can be populated by inserting data using `add`:

```
add subscriber("1234",
               "Joe",
               "Smith",
               "Main Street 1") = true;
```

1. The syntax for the pure relational model is a little awkward since no extensions have been made to the syntax of AMOSQL. A future extension can easily be made with more SQL-like syntax.

```
add subscriber("5678",
               "Albert",
               "Smith",
               "Park Avenue 5") = true;
add subscriber("9999",
               "George",
               "Williams",
                "Circle Drive 17") = true;
```

Updates can be made using `set`, e.g. append ", NDBCity" to each subscriber address:

```
for each charstring snb,
         charstring gname,
         charstring fname,
         charstring address,
         charstring new_address
where subscriber(snb, gname, fname, address) and
      new_address = address + ", NDBCity"
do set subscriber(snb,
                  gname,
                  fname,
                  new_address) = true;
```

and deletions using `remove`:

```
remove subscriber("9999",
                  "George",
                  "Williams",
                  "Circle Drive 17, NDBCity")
                  = true;
```

Any AMOSQL queries can be made over NDB-tables, e.g. fetch the name and address of the subscriber with the number "1234":

```
select gname, fname, address
from charstring snb,
     charstring gname,
     charstring fname,
     charstring address
where subscriber(snb, gname, fname, address) and
      snb = "1234";
<"Joe", "Smith", "Main Street 1, NDBCity">
```

Fetch all subscribers with the last name "Smith":

```
select gname, fname, snb
from charstring snb,
     charstring gname,
     charstring fname,
```

```
     charstring address
where subscriber(snb, gname, fname, address) and
      fname = "Smith";
<"Joe", "Smith", "1234">
<"Albert", "Smith", "5678">
```

The first query searches the subscriber table using the primary key which is very efficient and the second query has to scan the whole table which is much less efficient. If the AMOSQL query optimizer has the choice of using keys instead of scanning tables the query plan will be reordered to use available keys (see section Query Processing and Query Optimization below).

Arbitrary complex relational views can be defined in QDB by defining derived functions over NDB-tables, e.g. create a function `snb` that takes the name and address of a subscriber and returns the subscriber number:

```
create function snb(charstring gname,
                    charstring fname,
                    charstring address)
                    -> <charstring snb> as
select snb
where subscriber(snb, gname, fname, address);

snb("Joe", "Smith", "Main Street 1, NDBCity");
<"1234">
```

Create a function that finds all the subscribers with the same family name:

```
create function same_fname()
                    -> <charstring snb,
                        charstring gname,
                        charstring fname,
                        charstring address> as
select gname1, fname1, address1, snb1
from charstring snb1,
     charstring gname1,
     charstring fname1,
     charstring address1
where fname = fname1 and
      snb(gname, fname, address) !=
      snb(gname1, fname1, address1);

same_fname();
<"Joe", "Smith", "Main Street 1, NDBCity",
"1234">
<"Albert", "Smith", "Park Avenue 5, NDBCity",
"5678">
```

Object-Oriented Schemas in QDB

The Object-Oriented (OO) data model of AMOSQL can be used for defining an OO schema for NDB. This also fits better with the use of AMOSQL since it was designed for primarily modeling OO data. The OO schema in QDB is mapped to a flat relational data model that can be stored in NDB tables. This is done by using translated types and proxy objects. The type (equivalent to an object class) hierarchy is stored in QDB and all object attributes of a type are mapped to a table that stores the data of all the instances of the type. A special table in NDB, called `ndbtypes`, stores the Object Identifiers (OIDs) of all NDB objects that can be accessed as proxy objects in QDB (see Fig. 3). Foreign functions are used for accessing attributes of objects stored in QDB. Derived functions in QDB can be used for defining complex views of objects stored in NDB.

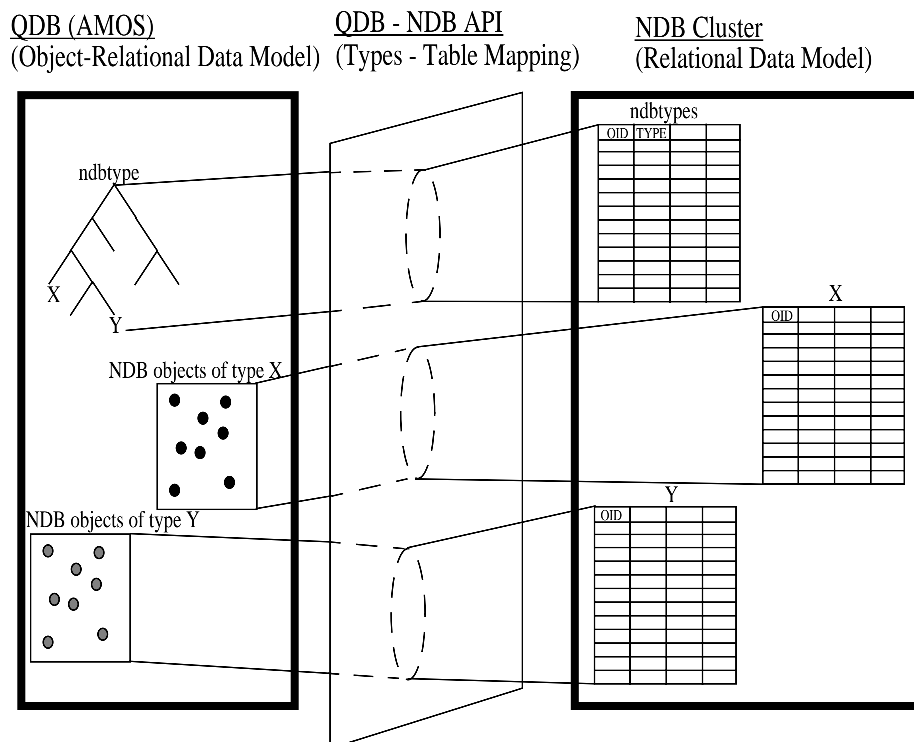

Figure 3: Mapping of QDB Object Types (Classes) to NDB Tables

New types created as subtypes of `ndbtype` will have their instances stored in NDB:

```
create type subscriber subtype of ndbtype
properties (snb charstring, /* subscriber number */
```

```
          gname charstring,  /* given name */
          fname charstring, /* family name */
          address charstring);
```

This creates a table in NDB with the type subscriber as primary key.

Instances of an `ndbtype` can be created with the attribute values inserted as tuples into the corresponding NDB table:

```
create subscriber(snb, gname, fname, address)
instances
:snb1("1234", "Joe", "Smith", "Main Street 1"),
:snb2("9999", "George", "Williams", "Circle Drive
17");
```

Objects are stored in the NDB table corresponding to their type and with their OID as primary key (represented by an `unsigned int`). Objects stored in NDB are handled as proxy objects which are transient (non-persistent) objects that are created temporarily during transactions in QDB. The persistency of objects is maintained in NDB by storing the object OID numbers in a special `ndbtypes` table and storing the object attribute values in a tuple of the table for the specific `ndbtype`. The type integrity of attributes of a certain type (i.e. type `ndbtype` or subtypes of type `ndbtype`) is maintained by QDB which automatically transforms OIDs of proxy objects of type `ndbtype` to integers representing their OID number.

Attributes can be directly modified (inserted, deleted, or updated), e.g. append ", NDBCity" to each subscriber address:

```
for each subscriber s, charstring address
where new_address = address(s) + ", DBBCity"
do set address(s) = new_address;
```

Objects of `ndbtype` can also be stored as object attribute vales of other `ndbtype` objects:

```
create type account subtype of ndbtype
          properties ( owner subscriber,
                       balance real);

create account(owner, balance) instances
          :acc1(:snb1, 10.0),
          :acc2(:snb2, 17.7);
```

QDB supports full inheritance over ndbtypes since the type hierarchy is

managed by AMOS. Each new subtype results in a new NDB table that contains any new or overloaded attributes.

```
create type mobile_subscriber
          subtype of subscriber
          properties (msnb charstring);

create mobile_subscriber(gname, fname, snb, msnb)
instances
:mb("Kalle", Karlsson", "1717", "07061717");
```

The attributes belonging to the `subscriber` type are stored in the subscriber table[1] and attributes of the new type are stored in the `mobile_subscriber` table.

Query Processing in QDB

NDB objects are stored temporarily as proxy objects in QDB, e.g. fetch the value of the temporary interface variable `:snb1`:

```
select :snb1;
#[OID 317]
```

Search on primary key (OIDs are primary keys as default), e.g. fetch the subscriber who owns the account represented by `:acc1`:

```
owner(:acc1)[2];
#[OID 317]
```

Search on non-primary key (other than OID) by scanning the table, e.g. fetch the name of the subscriber with the number “1234”:

```
select gname(s), fname(s) from subscriber s
where snb(s) = "1234";
<"Joe", "Smith">
```

OIDs that are returned from queries will create proxy objects (if the proxy objects don’t already exist):

```
select s from subscriber s where gname(s) = "Joe";
#[OID 317]
```

Complex queries (find the full name of the subscriber with the highest balance)

```
select gname(s),fname(s)
```

---

1. Attributes that are not set, e.g. the `address` attribute, are given NULL values.
2. Short form for `select owner(:acc1);`

```
from subscriber s, account a1
where  owner(a1) = s and
       balance(a1) = maxagg(select balance(a2)
                            from account a2);
<"George","Williams">
```

The AMOSQL query compiler chooses inherited attributes at compile time whenever possible:

```
select gname(x), fname(x), snb(x), msnb(x)
from mobile_subscriber x;
<"Kalle", Karlsson", "1717", "07061717">
```

Complex queries can be stated easily in AMOSQL. Here follows an example of a complex search for a persons ancestors that would be quite difficult to hard-code using the low level API:

```
create type person subtype of ndbtype
        properties(name charstring,
                   mother person,
                   father person);

create person(name) instances
        :p1("Albert"),
        :p2("Svea"),
        :p3("Johanna"),
        :p4("Karl"),
        :p5("Anna"),
        :p6("Johan"),
        :p7("Pelle");

set mother(:p3) = :p2;
set father(:p3) = :p1;
set mother(:p5) = :p3;
set father(:p5) = :p4;
set mother(:p7) = :p5;
set father(:p7) = :p6;

create function grandparent(person p) -> person gp
as
select gp where gp = mother(mother(p)) or
                gp = father(mother(p)) or
                gp = mother(father(p)) or
                gp = father(father(p));

create function greatgrandparent(person p)
          -> person ggp as
select ggp
```

```
where ggp = mother(grandparent(p)) or
      ggp = father(grandparent(p));

select name(greatgrandparent(p)) from person p
where name(p) = "Pelle";
“Albert”
“Svea”
```

The NDB Cluster Interface

The QDB query interface uses the low-level NDB API by automatically generating foreign functions that directly interact with NDB. The foreign functions are defined with different binding patterns that are included in query plans as foreign predicates.

Access patterns define which arguments are assumed to be bound or unbound during query processing and makes it possible to generate different API calls (see Appendix: The NDB Cluster Application Programming Interface) depending on if primary key attributes (or attributes with secondary indexes) are bound or unbound. A bound attribute value for a primary key can utilize direct access to a tuple in a table while an unbound key requires scanning the table to find matching tuples. The QDB interface to NDB uses typed registers which are small buffers managed by the interface (i.e. they are not stored in the database) and are used for passing and retrieving data to and from NDB.

The Relational Interface

Lets take a look at the following definition again:

```
create function
subscriber(snb charstring key,
           gname charstring,  /* given name */
           fname charstring,  /* family name */
           address charstring) -> boolean
as foreign “NDB-table”;
```

This directly defines a table called `subscriber` in NDB with `snb` as primary key and with three other attributes. The interaction with NDB looks as follows:

```
(let ((sop (ndb_get-schema-op *ndb-schema-trans-
action*)))
  (ndb_create-table sop “subscriber” 10 ’tuplekey
2 ’all)
```

```
   (ndb_create-attribute sop “snb” ‘tuplekey nil
nil ‘string)
   (ndb_create-attribute sop “gname” ‘nokey nil
nil ‘string)
   (ndb_create-attribute sop “fname” ‘nokey nil
nil ‘string)
   (ndb_create-attribute sop “address” ‘nokey nil
nil ‘string)
   (ndb_execute-schema-transaction *ndb-schema-
transaction*))
```

The `*ndb-schema-transaction*` variable is a handle to the current schema transaction on NDB.

Two internal access functions[1] are created in QDB for accessing this table and that return found tuples to the AMOSQL query processing system using the `osql`[2]`-result` function. One for fetching a single tuple given a primary key value:

```
(defun subscriber-+++ (_obj _snb _gname _fname
_address)
  (let ((top (ndb_get-operation *ndb-transaction*
"subscriber"))
        __gname __fname __address)
   (ndb_read-tuple top)
   (ndb_equal top "snb" _snb)
   (setq __gname
      (ndb_make-register ’string)
   (setq __fname
      (ndb_make-register ’string))
   (setq __address
      (ndb_make-register ’string))
   (ndb_get-value top "gname" __gname)
   (ndb_get-value top "fname" __fname)
   (ndb_get-value top "address" __address)
   (ndb_execute-transaction *ndb-transaction*
‘nocommit)
   (osql-result _snb
      (ndb_get-register __gname)
      (ndb_get-register __fname)
```

1. The names of the functions are appended with access patterns where “-” symbolizes a bound argument and “+” symbolizes unbound.
2. OSQL[7] was the query language of Iris [2] and is the predecessor of AMOSQL, the function name `osql-result` has been kept for backward compatibility reasons.

```
      (ndb_get-register __address))))
```

and one access function for fetching several tuples when the primary key value is not known:

```
(defun subscriber++++ (_obj _snb _gname _fname
_address)
  (let ((top (ndb_get-operation *ndb-transaction*
"subscriber"))
        __snb __gname __fname __address)
    (ndb_open-scan top)
    (setq __snb
       (ndb_make-register 'string))
    (setq __gname
       (ndb_make-register 'string))
    (setq __fname
       (ndb_make-register 'string))
    (setq __address
       (ndb_make-register 'string))
    (ndb_get-value top "snb" __snb)
    (ndb_get-value top "gname" __gname)
    (ndb_get-value top "fname" __fname)
    (ndb_get-value top "address" __address)
    (while (not (ndb_scan-is-eof top))
       (ndb_scan-get-next top)
       (ndb_execute-operation top)
       (osql-result
          (ndb_get-register __snb)
          (ndb_get-register __gname)
          (ndb_get-register __fname)
          (ndb_get-register __address)))
    (ndb_close-scan top)))
```

The `*ndb-transaction*` variable is a handle to the current read/write transaction on NDB.

Tuples are inserted into table using the AMOSQL add expression:

```
add subscriber("1234", "Joe", "Smith", "Main
Street 1") = true;
```

which is translated into the following sequence of operations towards the NDB-API:

```
(let ((top (ndb_get-operation *ndb-transaction*
"subscriber"))
```

```
      (attrv (listtoarray (list "1234" "Joe"
"Smith" "Main Street 1")))
      __gname __fname __address)
    (ndb_insert-tuple top)
    (ndb_equal top "snb"
       (elt attrv 0))
    (setq __gname
       (ndb_make-register 'string))
    (setq __fname
       (ndb_make-register 'string))
    (setq __address
       (ndb_make-register 'string))
    (ndb_set-register __gname
       (elt attrv 1))
    (ndb_set-register __fname
       (elt attrv 2))
    (ndb_set-register __address
       (elt attrv 3))
    (ndb_set-value top "gname" __gname)
    (ndb_set-value top "fname" __fname)
    (ndb_set-value top "address" __address)
    (ndb_execute-transaction *ndb-transaction*
`nocommit))
```

Similar instruction sequences are generated for updates and deletions of tuples (`ndb_insert-tuple` is substituted by `ndb_update-tuples` or `ndb_delete-tuples`).

The Object Oriented Interface

In the OO interface type definitions in QDB are translated into a flat relational model in NDB.

Let's take a look at the following definition again:

```
create type subscriber subtype of ndbtype
properties
(snb charstring,    /* subscriber number */
 gname charstring,  /* given name */
 fname charstring,  /* family name */
 address charstring);
```

This will create a new proxy type `subscriber` (here represented by the type object `#[OID 377 SUBSCRIBER]`) and the following definitions:

```
create function
subscriber(oid subscriber key,
           snb charstring,
```

```
            gname charstring,
            fname charstring,
            address charstring) -> boolean
as foreign “NDB-table”;
create function snb(oid subscriber) -> charstring
as foreign “NDB-attribute”;
create function gname(oid subscriber)
            -> charstring
as foreign “NDB-attribute”;
create function fname(oid subscriber)
            -> charstring
as foreign “NDB-attribute”;
create function address(oid subscriber)
            -> charstring
as foreign “NDB-attribute”;
```

The NDB table is created in a similar manner as for the relational interface, except that the subscriber type is translated into an `unsigned int`. Two access functions similar as presented for the relational interface described above are automatically created for the subscriber table. The only difference is that this table has an `unsigned int` that represents the OID of the subscriber and which is the primary key.

In the direct index access function the bound key attribute is a proxy OID and the OID number is fetched with the function `get_proxy_oid` and is used for accessing the primary index:

```
(defun subscriber-++++ (_obj _oid _snb _gname
_fname _address)
  (let ((top (ndb_get-operation *ndb-transaction*
"subscriber"))
        __snb __gname __fname __address)
   (ndb_read-tuple top)
   (ndb_equal top "oid"
      (get_proxy_oid _oid))
   (setq __snb
      (ndb_make-register ’string))
   (setq __gname
      (ndb_make-register ’string))
   (setq __fname
      (ndb_make-register ’string))
   (setq __address
      (ndb_make-register ’string))
   (ndb_get-value top "snb" __snb)
   (ndb_get-value top "gname" __gname)
   (ndb_get-value top "fname" __fname)
```

```
   (ndb_get-value top "address" __address)
   (ndb_execute-operation top)
   (osql-result _oid
      (ndb_get-register __snb)
      (ndb_get-register __gname)
      (ndb_get-register __fname)
      (ndb_get-register __address))))
```

In the index-scan access function this integer is found during scanning and is transformed to an OID (using the `find-ndb-object` function) for every tuple that is returned:

```
(defun subscriber+++++ (_obj _oid _snb _gname
_fname _address)
  (let ((top (ndb_get-operation *ndb-transaction*
"subscriber"))
        __oid __snb __gname __fname __address)
   (ndb_open-scan top)
   (setq __oid
      (ndb_make-register 'unsigned))
   (setq __snb
      (ndb_make-register 'string))
   (setq __gname
      (ndb_make-register 'string))
   (setq __fname
      (ndb_make-register 'string))
   (setq __address
      (ndb_make-register 'string))
   (ndb_get-value top "oid" __oid)
   (ndb_get-value top "snb" __snb)
   (ndb_get-value top "gname" __gname)
   (ndb_get-value top "fname" __fname)
   (ndb_get-value top "address" __address)
   (while (not (ndb_scan-is-eof top))
      (ndb_scan-get-next top)
      (ndb_execute-operation top)
      (osql-result
         (find-ndb-object #[OID 377 SUBSCRIBER]
            (ndb_get-register __oid))
         (ndb_get-register __snb)
         (ndb_get-register __gname)
         (ndb_get-register __fname)
         (ndb_get-register __address)))
   (ndb_close-scan top)))

(defun subscriber.snb-+ (_obj _oid _snb)
```

```
  (let ((top (ndb_get-operation *ndb-transaction*
"subscriber"))
        __oid __snb)
    (ndb_read-tuple top)
    (ndb_equal top "oid"
       (get_proxy_oid _oid))
    (setq __snb
       (ndb_make-register 'string))
    (ndb_get-value top "snb" __snb)
    (ndb_execute-operation top)
    (osql-result _oid
       (ndb_get-register __snb))))

(defun subscriber.snb++ (_obj _oid _snb)
  (let ((top (ndb_get-operation *ndb-transaction*
"subscriber"))
        __oid __snb)
    (ndb_open-scan top)
    (setq __oid
       (ndb_make-register 'unsigned))
    (setq __snb
       (ndb_make-register 'string))
    (ndb_get-value top "oid" __oid)
    (ndb_get-value top "snb" __snb)
    (while (not (ndb_scan-is-eof top))
      (ndb_scan-get-next top)
       (ndb_execute-operation top)
       (osql-result
          (find-ndb-object #[OID 377 SUBSCRIBER]
             (ndb_get-register __oid))
          (ndb_get-register __snb)))
    (ndb_close-scan top)))
```

The functions `subscriber.gname-+`, `subscriber.gname++`, `subscriber.fname-+`, `subscriber.fname++`, `subscriber.address-+`, and `subscriber.address++`. Functions `account.owner-+`, `account.owner++`, `account.balance-+`, and `account.balance++`. Insertion of data is performed in a similar manner as explained for the relational interface above, but with the translation of OIDs to unsigned integers before they are inserted into the NDB-table.

Type extents (the set of instances of a type) are managed by an NDB-table called `ndbtypes`. In an expression such as:

```
select o for each subscriber o;
```

a QDB function called `typesof` is called which is overloaded for objects of `ndbtype`:

```
(defun ndbtype.typesof->type+-(obj nobj tp)
  (mapfunction1 _ndboid_ (list tp)
      #'(lambda (resl)
            (osql-result (find-ndb-object tp
(first resl)) tp)))
  (mapfunction _subtypes_ (list tp)
      #'(lambda (resl)
           (let ((sbtp (first resl)))
               (mapfunction _ndboid_
                   (list sbtp)
                   #'(lambda (resl)
                        (osql-result
                           (find-ndb-object sbtp
(first resl))
                                          tp)))))))
```

The variable `_ndboid_` is a handle to the AMOSQL function `ndboid` that accesses the table ndbtypes and which is defined as:

```
create function ndboid(type ndbtype)
           -> integer ndboid as
select ndboid
where ndbtypes(ndboid, name(ndbtype));
```

This function finds all the instances of a given type as well as the instances of the subtypes of the given type.
The variable `_substypes_` is a handle to the AMOSQL function `subtypes` which returns all subtypes of a given type.

Query Processing and Query Optimization

Take the query:

```
select gname(s), fname(s)
from subscriber s, account a
where balance(a) > 15.0 and owner(a) = s;
```

1. `mapfunction` is an internal AMOS function that iterates over an AMOSQL function and passes each matching tuple to a given functional argument.

This generates a query plan in ObjectLog[1] with the name `*select*`:

```
*select*(_arg0, _arg1) ←
     subscriber.gname++(_gname, s _arg0) ∧
     subscriber.fname-+(_fname, s _arg1) ∧
     account.owner++(_owner, a s) ∧
     account.balance-+(_balance, a _arg2) ∧
     gt--(_gt,_arg2 15.0)
```

where

```
_gname = #[oid 388 subscriber.gname->charstring]
_fname = #[oid 390 subscriber.fname->charstring]
_owner = #[oid 400 account.owner->subscriber]
_balance = #[oid 402 account.balance->real]
_gt = #[oid 86 object.object.>->boolean]
```

This query plan finds all subscribers and their `gname` by index scanning and then finds their `fname` using the primary key. It then finds the account for this particular subscriber (by scanning) then the account `balance` (using the primary key). The functions[2] `subscriber.gname++` and `account.owner++` are defined in a similar manner as `subscriber.snb++` (described in the previous section). The functions `subscriber.fname-+` and `account.balance-+` are defined in a similar manner as `subscriber.snb-+`.
Notice here that the order between `balance` and `owner` has been swapped by the AMOSQL query optimizer [6] since we don't have to find the `balance` of every `account`, just the `balance` for the current `subscriber`. If we would have a secondary index on the `owner` attribute (i.e. an `account.owner+-` function), the query optimizer would have chosen that function (since `s` is bound by `subscriber.gname++`). A function `costhint` is available in AMOSQL for providing cost information to the query optimizer, for example:

```
costhint('account.owner->subscriber', 'fb', {2,
1});
```

tells the optimizer that the expected cost for `account.owner+-` is 2 and the fan-out is 1. The string `'fb'` specifies that the costhint is for the

1. ObjectLog [6] is an Object Oriented extension of Datalog and is the internal logical language for query plans in AMOS.
2. Each AMOSQL function has a logical predicate whose different attribute bindings are associated with different foreign functions.

predicate of function `account.owner->subscriber` where the first argument is free `'f'` and the second argument is bound `'b'`. The cost represents a cost measure for executing a function given one specific input binding. The fan-out is the expected number of output tuples for one specific input binding. In the QDB interface default costhints are defined for the automatically defined functions that can be overridden be calling `costhint`. When new costhints are provided all old query plans that reference that predicate has to be reoptimized. This can be done by calling the AMOSQL function `reoptimize`.

Here the > operation is executed by QDB using the `gt--` function, but one future extension could be to push the comparison operator into the low-level instruction sequence that is sent to NDB. Work on multi-database support in AMOS [5] can here be utilized to specify rules or patterns of what partial query plans that can be pushed down to NDB.

QDB Transactions

In NDB there are two types of transactions, schema transactions and read/write transactions. In schema transactions new tables and table attributes can be defined (and dropped or altered). Read/write transactions can do insert, delete, or update of tuples or table attributes. In AMOS no distinction is made between schema transactions and read/write transactions. Transactions in QDB uses AMOS transactions and attaches one schema transaction and one read/write transaction in NDB. See the NDB-API description in the appendix.

Summary and Future Work

This paper presented QDB, a query processing front-end to the high-performance, parallel data server NDB Cluster. QDB is built on-top of AMOS and uses the extensibility of the AMOS storage manager and foreign functions of the query language AMOSQL. QDB gives both a relational interface and an Object-Oriented (OO) interface to NDB Cluster.

In the relational interface the user can define NDB-tables (seen as updateable foreign functions in AMOSQL), populate them (insert, update, and delete tuples), and make any AMOSQL queries over the tables. Derived functions (relational views) can be defined over the NDB-tables.

In the OO interface the user can create new types (object classes of class NDB-type) and sub-types (sub-classes) with attributes of literal type (integers, real numbers, or strings) or attributes of other NDB-types. The attributes can be set (inserted, updated, or deleted) using the OIDs of instances of NDB-types and are stored as one tuple in a table that represents all the object data of the specific NDB-type and where the OID (rep-

resented as an integer) is the primary key. The user can make any AMOSQL queries over the NDB-types and their attributes. Derived functions (object-relational views) can be defined over the NDB-types.

The current implementation does not use all the functionality in the NDB-API. Comparison operators (<, >, >=, <=) are currently executed in the QDB query plan, but could be pushed down into the NDB instruction sequence.

In the current implementation of QDB a special table ndbtypes is used for storing OIDs in NDB. This is a potential bottleneck and future work should investigate how OIDs can be better supported by NDB. The ndbtypes table could be replicated to several nodes to increase read access (fetching OIDs) at the cost of slowing down write access (creation of OIDs). NDB has tuple identifiers, but these are currently not documented and it is unclear if these are sufficient as substitutes for OIDs.

Acknowledgements

I would like to thank Mikael Ronström, the creator of NDB CLuster, for providing us with versions of the system and for reviewing this paper. Prof. Tore Risch, the creator of AMOS, supervised the work and helped in technical discussions about QDB.

References

[1] DataBlitz Architectural Overview, Technical White Paper, *Lucent Technologies*, 1998.

[2] Fishman D. et. al: Overview of the Iris DBMS, Object-Oriented Concepts, Databases, and Applications, *ACM press, Addison-Wesley Publ. Comp.*, 1989.

[3] Flodin S., Josifovski V., Karlsson J. S., Risch T., Sköld M., and Werner M.: AMOS II User's Guide, Tech. Report, *Dept. of Comp. Science, Linköping University*, Sweden, 1998.

[4] IEEE SCI Draft 2.00, SCI Scalable Coherent Interface, SCI: D2.00 P1596-18Nov91-doc233, 1991.

[5] Josifovski V. and Risch T.: Transformation of Queries over Object-Oriented View in a Database Mediator System, *IFCIS CoopIS '98*, New York, NY, 1998.

[6] Litwin W. and Risch T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, Dec. 1992.

[7] Lyngbaek P., OSQL: A Language for Object Databases, tech. rep. HPL-DTD-91-4, *Hewlett-Packard Company*, Jan. 1991.

[8] Ronström M.: The NDB Cluster - A parallel data server for telecom-

munications applications, *Ericsson Review: The Telecommunications Technology Journal* - No. 4, 1997.

[9] Sköld M.: Active Database Management Systems for Monitoring and Control, Linköping Studies in Science and Technology, Dissertation No. 494, *Dept. of Comp. Science, Linköping University*, Sweden, 1997.

[10] TimesTen Main-Memory Data Management, Technical White Paper, *TimesTen Performance Software*, November, 1998.

Appendix: The NDB Cluster Application Programming Interface

The NDB-API is a low-level Applications Programming Interface in C++ that has been linked together with QDB. Most API calls have been made available to be called from the internal Lisp language of QDB. Here follows a short description of the used API calls:

`ndb_connect(dbname)`
Connect to database named `dbname`, initialize it, open one read/write transaction and one schema transaction.

`ndb_disconnect()`
Disconnect from the current database and close the associated read/write and schema transactions.

`ndb_init(db nrcon nrop)`
Initialize a database `db` with number of allowed connections `nrcon` and number of allowed operations `nrop`.

`ndb_open_schema-transaction(db)`
Open a schema transaction for database `db`.

`ndb_close-schema-transaction(db stran)`
Close schema transaction `stran` on database `db`.

`ndb_open-transaction(db)`
Open a read/write transaction for database `db`.

`ndb_close-transaction (db tran)`
Close read/write transaction `stran` on database `db`.

`ndb_execute-transaction (tran exectype)`
Execute all associated instructions and operations of read/write transaction `tran`. Attribute `exectype` can be `'commit`, `'nocommit`, or `'noexectypedef`.

`ndb_execute-schema-transaction (stran)`
Execute all associated instructions and operations of schema transaction `stran`.

`ndb_get-schema-op(stran)`
Creates a schema operation for schema transaction `stran`.

`ndb_get-operation(tran table)`
Create an operation for read/write transaction `tran`.

`ndb_execute-operation(op)`
Execute one specific read/write operation `op`. Used in scans where the same operation is executed repeatedly.

`ndb_create-table(sop tname tsize tkey nrpages ftype nrbackup nrstandby kvalue minload maxload memtype)`
Creates a table with name `tname`. The argument `tkey` specifies the type of key for the table. See NDB documentation for the other arguments.

`ndb_create-attribute(sop attrname tkey attrsize arraysize attrtype safetype storagemode nullattr)`
Creates an attribute with name `attrname` and type `attrtype` for a previously created table. The attribute types can currently be `'unsigned`, `'float`, or `'string`. This easily be extended in the future. The argument `tkey` specifies if the attribute is a key (value `'tuplekey)` or not (value `'nokey)`. See NDB documentation for the other arguments.

`ndb_read-tuple (op)`
Associate a read tuple instruction with the read/write operation `op`.

`ndb_insert-tuple (op)`
Associate an insert tuple instruction with the read/write operation `op`.

`ndb_update-tuples (op)`
Associate an update tuples instruction with the read/write operation `op`.

`ndb_delete-tuples (op)`
Associate a delete tuples instruction with the read/write operation `op`.

`ndb_equal(op attr value)`
Associate an equal test with the read/write operation `op` that checks if the attribute `attr` has the value `value`.

`ndb_set-value(op attr reg)`
Associate a set value instruction with read/write transaction op that sets attribute `attr` to the value of register `reg`.

`ndb_get-value(op attr reg)`
Associate a get value instruction with read/write transaction op that gets the value of attribute `attr` and puts it in register `reg`.

`ndb_open-scan(op)`
Opens a scan on table associated with operation `op`.

`ndb_close-scan(op)`
Closes a scan on table associated with operation `op`.

`ndb_scan-get-next (op)`
Fetches next tuple in scan on table associated with operation `op` and puts values from the tuple in registers associated with previous `ndb_get-value` instructions.

`ndb_scan-is-EOF(op)`
Checks is the scan on table associated with operation `op` has reached the end of the table.

The following instructions are not part of the NDB-API, but are used by QDB to create registers and to read/write values from/to NDB into/from QDB registers:

`ndb_make-register(regtype)`
Creates a register of type `regtype`. The register types can currently be `'unsigned`, `'float`, or `'string`.

`ndb_get-register(reg)`
Gets the value of register `reg`. If a NULL valued attribute is fetched from NDB (with an `ndb_get-value` instruction) `nil` is returned.

`ndb_set-register(reg value)`
Sets a value of a register `reg` to `value`. Performs type checking so that the value of the type and the register match. If value is `nil` the register will cause an attribute to be set to NULL (with an `ndb_set-value` instruction).