

# PROPOSED NOVA

## DATABASE INTERFACE REQUIREMENTS

### BASED ON MINOS (AND SOUDAN2 AND SNO) EXPERIENCE

## Contents

Overview	2
Requirements	2
User Access	2
Performance	3
Support	3
Discussion	4
Basics of Table Structure	4
Primary table	4
Validity table	5
Design Philosophy	6
Other Important Issues	7
Text based override	7
Extended queries	8
Deep in the design	8
Further Discussion	9
Examples	10
Use Case 1: normal queries	10
Use Case 2: extended queries	10
Bibliography	11

# Overview

The original requirements for the MINOS offline software database interface were drawn up circa-1999 primarily based on the prior experience of Nick West, drawing on similar work for the Soudan2 and SNO experiments. Many of the fundamentals worked out in those experiments were carried over to the MINOS design while additions were made based on what was known to be lacking in those designs and the addition of new requirements from MINOS, notably a requirement to support multiple detector locations (Near, Far and CalDet test at CERN).

This document will try to summarize these requirements and provide rationales for the design and implementation what was the result. Many of the reasons for choosing a particular design of the system still hold for experiments such as NOvA, while actual implementation technology might need some modifications. Readers are urged to peruse the relevant MINOS “Package Rationale” and “User Manual” sections. I will endeavor to restate most of those requirements and lay out the reasoning behind them in this document while trying to remove biases that are not technology agnostic.

## Requirements

The primary requirements of the offline database interface had to do with delivering datasets to the reconstruction and analysis software in a performant manner. The canonical example is calibration data. The code is intended to satisfy the request: “give me all the data of that type which is valid for this event”. Alternative views, or *extended queries* of the data proved to also be of use, optimization for the first type of query it shouldn’t preclude the alternative.

### User Access

1. Data must be presented to the end-user as a convenient collection, or *result set*, of table-specific class-objects where objects represented a structured *row*. The system should not impose a particular scheme for segmenting the data; a row might represent a single channel or some collection of channels, whichever is sensibly sized for the table designer/end user.
  - a. The minimal specification of what data to return should require no more than the table name/class, a timestamp, which detector, and whether it was data or simulation. This should yield the “best” valid collection with no additional hints or selections on the user’s part. In a standard query this should present no ambiguities (i.e. the same channel should not appear more than once in the collection).
  - b. The class-objects should support table schema evolution in a convenient manner; generally this means the addition of columns, but could in extreme circumstances mean renaming or reordering columns. There should be support for updated code to be used by sites that haven’t transitioned their database table format and older code releases to sensibly handle newer layouts after a transition.
  - c. The collection structure should support an indexing scheme that avoids the need for the user to loop over all items in order to find the appropriate item (*natural indices*).

2. The system must have the ability for the end user (i.e. not the database administrator) to logically re-create the state-of-the-database at any time in the past (this is referred to as *rollback*). MINOS, with the potential for disruption of communication between the far site and FNAL, there were two concerns:
  - a. Capturing when data entered the system as a whole (*creation datetime*)
  - b. Capturing when data entered a particular database server (*insert datetime*)
3. Once retrieved, the data must be locally cached by a job process; the DBI code should *own* the objects and manage the memory, taking care of lifetime issues; choices of const-ness should protect the user from accidental local modifications.
4. It must be possible to inject database structured data into the system on a job-by-job basis without special rights. This allows users to *test* modifications to the database without actually having privileges to commit to the database.

## Performance

5. Tables and entries need to be structured to minimize duplication of the data in the database.
  - a. The system must support the ability to update only a part of the collection (*aggregate*) to allow such activities as partial calibrations for a chosen subset of all the channels; in NOvA parlance this might mean calibrating only individual blocks at a time, potentially on different schedules, rather than the detector as a whole.
  - b. The system must support the ability to put in entries for a limited time period (e.g. mask out a few channels for a limited time period without duplicating the entirety of the aggregate a second time).
  - c. There must be support for occasions where rows are valid for multiple detectors or valid for both real data and simulations, as well as cases where those sets are disjoint.
6. The access to the data must be performant; queries to the DB need to be structured to return as little data as necessary to satisfy the request without being so overly complex as to bog down the server; optimizations include indexing the appropriate table columns but no fancy Oracle-style hints.

## Support

7. The system must support a variety of connectivity issues:
  - a. The system should allow for a cascade of sources, i.e. if valid data isn't found at the first source then it should attempt retrieval from a second, etc. It should be easy to configure the cascade order on a job-by-job basis.
  - b. The system must be configurable to allow for (albeit constrained) access in situations where connectivity is limited (i.e. the *laptop on a plane* problem). While duplication of all the DB data might not be possible in such cases, one should be able to do processing (e.g. simulations or code development) by preloading a limited selection of the data into a local server. Setting up such a server should be do-able by a standard physicist.

- c. There must be a uniformity in what the end-user sees, independent of the database server technology (e.g. MySQL, Oracle, etc.).
  - d. Connections to the database (which for MINOS were direct) must be transient; there are inherent connection limits and license issues that precluded holding connections open.
8. There should be a basic level of uniformity in the layout of tables where the mechanism for dealing with the aforementioned requirements is a common feature and the only substantive difference in table layout is the payload data itself.
9. There must be a distribution system for efficiently synchronizing databases at remote sites:
- a. There must be support for the potential for data to flow bi-directionally (e.g. the far detector serves as a source for some tables but is not the *master* for most tables).
  - b. There should be a mechanism for validating that chunks of data are correctly duplicated at remote sites while allowing for the remote site not to be a complete duplicate (perhaps limiting their tables to a particular time period or detector).

## Discussion

### Basics of Table Structure

MINOS used a two-table structure for each type of data: a main table to hold the bulk of the data, and a validity table to map event info (timestamp, detector, etc) to the single element (SEQNO) key(s) for a collection of rows in the primary table. By splitting the two components the design avoids duplicating bulky validity information.

#### Primary table

The main table consists of a SEQNO column, a ROW\_COUNTER column, and one or more (generally more: 4-20) user specified columns. More than one row can have the same SEQNO in this table. Lookups are done based solely on the SEQNO column; queries on this table collect all entries matching a selection of one or more SEQNO values. Indexing the SEQNO column provided a significant performance improvement.

<i>FIELD</i>	<i>TYPE</i>	<i>NULL</i>	<i>KEY</i>
SEQNO	int	no	PRI
ROW_COUNTER	int	no	PRI
<i>data item 1</i>	<i>user specified</i>		
<i>data item 2</i>	<i>user specified</i>		
...			

The `ROW_COUNTER`, in conjunction with the `SEQNO`, allowed the pair to be indexed as a *primary key* (which require uniqueness) for integrity purposes. It was a late addition to the design and isn't absolutely necessary based on the requirements, but was implemented on recommendations from professional DBAs.

#### Validity table

The job of the validity (VLD) table is to allow for the selection of the correct set of `SEQNO` values based on the user supplied information (event time, detector, etc.). The `SEQNO` is the key that ties elements of the two tables together; it can appear in the VLD table only once (and thus is a primary key for the VLD table) and multiple times in the main table. This one-to-many mapping avoids much validity duplication.

<i>FIELD</i>	<i>TYPE</i>	<i>NULL</i>	<i>KEY</i>	<i>DEFAULT</i>	<i>EXTRA</i>
SEQNO	int	no	PRI		auto_increment
TIMESTART	datetime	no	MUL	0000-00-00 00:00:00	
TIMEEND	datetime	no	MUL	0000-00-00 00:00:00	
DETECTORMASK	tinyint	yes		NULL	
SIMMASK	tinyint	yes		NULL	
TASK	int	yes		NULL	
AGGREGATENO	int	yes		NULL	
CREATIONDATE	datetime	no		0000-00-00 00:00:00	
INSERTDATE	datetime	no		0000-00-00 00:00:00	

The [ `TIMESTART` , `TIMEEND` ) pair delineate the time for which the `SEQNO` is valid; `TIMEEND` not being part of the interval to avoid ambiguity on adjacent intervals. The requirement 5b for being able to patch limited time ranges without data duplication precludes the use of single-sided intervals (i.e. those implicitly extending from a start time off to "infinity" (and beyond)). In such a scheme if one has data valid from  $t_1$  to infinity, and need to patch from time  $t_2$  to  $t_3$ , one must put in the patch data as if it were valid from  $t_2$  to infinity and then duplicate original  $t_1$  data from  $t_3$  to infinity. This is wasteful and illogical (if the data is known to be valid only from  $t_2$  to  $t_3$ ). Having an end time also allows users to make entries that are limited to the actual time range used in calculating the values, or an estimated time when the data is likely to become invalid (a time by which a new entry should have been made).

The `DETECTORMASK` and `SIMMASK` are masks which are bit-wise tested against the user supplied detector and simulation flags suitably converted into a single bit, i.e. if the near detector is id 1, then the first bit is set, if the far detector is 2 then the second bit, etc. This allows a VLD row to be valid for more than one detector (similarly for data/simulation field). This allows the system to satisfy requirement 5c.

The `TASK` field (sometimes referred to in the design as “mode”) allows for further delineation of the data. It was anticipated that it would be used in two use cases:

- a tracking algorithm might have multiple configurations and use this to distinguish them.
- detector configuration could have two tasks, one for raw calibration one for refined.

MINOS made minimal (but non-zero) use of this task/mode capability; most tables uniformly set the default value of zero. It costs little to implement and provides additional flexibility.

The `AGGREGATENO` column allows a logical collection to be broken down into more manageable chunks in order satisfy requirement 5a. An example of an aggregate might be a plane, or a block. Such a unit would be calibrated at the same time. A finer segmentation of the aggregate (e.g. plane vs. block) means less data if a patch needs to be put in, but at the cost of more VLD table rows. Generally there is a natural unit or a happy medium can be reached. The query’s result collection contains the “best” dataset, decided independently for each aggregate.

The `CREATIONDATE` is the field used to resolve ambiguity of the “best” (requirement 1a) data meeting all other criteria. If multiple `SEQNO` rows satisfy the request (span the event’s time, match the detector flag, etc.) then one simply chooses the latest `CREATIONDATE` under the assumption that if newer data is available there must have been a reason for generating it (e.g. updated calibrations).

Finally the `INSERTDATE` tells when the row was put in this particular database instance. It is through the use of this field that *rollback* is possible (requirement 2). By telling the query to ignore all data inserted into the database after a particular timestamp one can re-create the state of the database at any previous time. One can preemptively use this to *freeze* a database during production processing, even while it continues to receive updates.

## Design Philosophy

One key aspect of the system is that decisions are based on time, not such things as run number. This is important for many reasons. First and foremost, time is monotonic and has a sensible ordering. Almost as important is that it is accessible to the users. They can use it to correlate with their experience with the detector, e.g. “channel X broke at time  $T_1$  and was fixed at time  $T_2$ ” or “I ran the calibrator on day D and there are the results”. They can correlate events they find in the log books or from talking with people with the database entries. Obviously, things happen independent of run boundaries, so run # is not a good candidate. Time can also serve to unify multiple detectors (run 1 in the near detector doesn’t correspond to run 1 in the far, but time is time -- time-of-flight being negligible at the quantization, seconds, of the database).

The only restriction is that the time needs to be UTC (or some equivalent, as long as everyone agrees on how leap seconds and such are handled); this avoids the two time a year daylight savings time shifts which can leave holes, or add ambiguity. If at all possible the database field should be structured to present itself to the user in ISO 8601 format: `YYYY-MM-DD hh:mm:ss`. Internally the database, ideally, would hold the data as a single integer (ala `time_t`, which is good until 2038-01-18 19:14:07) for efficiency of indexing (heavily used in this scheme).

Fundamental of the database management in this system is that data goes in and is never modified or deleted, at least for the master database; this allows for the reconstruction of the prior history of any table (requirement 2). To fix an error one makes new entries (with newer CREATIONDATE) to obscure the incorrect ones.

Rules are made to be broken, but MINOS saw less than a handful of infractions of the “no modifications” rule. One of few times this rule was broken in was when two sets went in with the same creation timestamp (to the second) . This the violated ambiguity resolution scheme and left the ordering for the “best” choice up to vagaries of how the server found the rows (semi-random/undefined in some cases). It should be very rare that two datasets are created covering the same collection at exactly the same second.

The collection class, or *result set*, as well as holding the individual row objects, must provide a means of reporting the validity of the collection in the form of a *validity range* (time range, mask of all legal detector and simflag mask bits). If a collection is to be made up from multiple aggregates (and thus multiple SEQNOs and VLD rows) then the reported validity range is must be valid for all the entries. While forming the collection from individual aggregates, the time range narrows down to starting the latest that any aggregate was valid and ending the earliest that still covers the requested time. All the aggregate DETECTORMASK and SIMMASK entries are taken as a logical-and (with each other). The result might be broader than the initial request; e.g. ask for data about the near detector and get back data that is valid for both the near and far.

Internal caching of the collection is done on a table-by-table basis; even after the user handle to the collection is dropped the collection is retained. Then when the next event’s *validity context* (timestamp, detector, simflag) is compared to the collection’s a decision can be made whether new data needs to be re-fetched from the server, i.e. once the validity context has moved out of range. This caching precludes reading a table that is undergoing updates in real time -- MINOS never had need of such facility. If NOvA does, one could probably slip in an mechanism for overriding the cache -- at the expense of continually querying the database.

Supported data column types need to include various sizes of signed and unsigned integers (tiny, small, large), floats and doubles, strings (text) and timestamps. The value retrieved from a floating point element must be the same (to within possible precision) to what was entered. This is important for checksumming,. The inability to achieve these goals was part of the reason MINOS abandoned Oracle after investing a lot of time trying to get it to work.

Ideally, a result set should be able to report how it was formed: which SEQNO entries and which entry in the cascade of sources. Class-object rows should be able to report which specific SEQNO they derived from and that SEQNO’s information such as task/mode, aggregate, creation and insertion timestamps. This kind of provenance information, while not generally stored into the event, can be extremely useful for debugging purposes.

## Other Important Issues

### Text based override

Support for non-privileged user to override of the database is an essential requirement. This allows users to test proposed entries to the database non-destructively. It also allows for “one-

off'' studies (e.g. do simulations where planes at the end were progressively ``turned off'' to simulate the potential need to scavenge electronic from the downstream part of the detector to prevent holes in the upstream). The number of instances of a need to do such things will surprised everyone.

In MINOS this capability took the form of structured ASCII text files that were copied into the database as temporary (only valid for the individual db connection) tables and then read back out via the normal mechanism. The details of implementation matter little as long as it is possible. Minimizing the discrepancy between the format of the real table and how the text file is structured will make user's lives easier (i.e. avoid requiring users to write SQL commands).

### Extended queries

Some times it is desirable to get back a larger set of rows than the minimum defined by the standard query, for such cases *extended queries* exist as an alternative. Extended queries have a variety of options for acceptance criteria for comparing a time range specified by the query to the time range of the `SEQNO`, which serve different purposes. One common use case in MINOS was an extra layer of caching when the underlying data entries had a very short validity span. For instance, the magnet current readback was recorded every 4 minutes or so. At the rate event records were processed this essentially meant that connections to the database were continually being made and closed. With the use of an extended context query, an additional layer of caching was introduced so that a full hour or more's worth of data was retrieved in a single query and the result wrapped up to loop over the set to find the right entry. This is a point where further development of the model would have been useful; there were a number of such nearly-identical instances that were re-invented semi-independently.

## Deep in the design

The MINOS interface eventually came to rely solely on the generic TSQL classes (`TSQLServer`, `TSQLStatement`, `TSQLColumnInfo`, etc) from ROOT to interface with the database. This set of classes wrapper the particular interface to MySQL. It is likely, though untested, that if the tables were duplicated in a PostgreSQL server and ROOT had the corresponding interface libraries built that the bulk of the code would work with just the change of an environment variable specifying the server URL. The only known MySQL-ism known to be in the code has to do with the loading of text files into the server as temporary tables.

The code made use of the concept of a handle or proxy, which served as the lightweight user-facing interface; data isn't fetched until one has a validity context in hand (as one would need to know which detector even for tables that might be static over the experiment's lifetime) and a query was made, generally from the handle constructor. Cache management behind the scene made the creation and destruction of the handles (`DbiResultPtr`) an inexpensive operation.

The MINOS DBI had an interface where the same row classes could be used to write into the database (given sufficient privileges) and logs of the writes would be made to a `EntryLog` table. This won't be further discussed in this memo.

There was also a provision for what was deemed a *L2Cache*, which essentially wrote the result set into a `.root` file for repeated use. While this worked it wasn't heavily used as there were issues with file management and scaling.

As mentioned earlier, the row class-objects were allowed to provide a *natural index* that allowed the result set to perform an efficient lookup. For example, if the result set were a set of rows representing something about individual planes the the natural index would be plane number -- this index was not required to be without gaps, only that there be no duplicates. There may be more efficient or C++'ee ways of doing the same thing now.

## Further Discussion

Originally MINOS table names used a StudlyCaps (or CamelCase) for the main table and the secondary table had ``vld`` appended to the name. Restrictions in Oracle forced MINOS to convert all table names (and column names) to upper case, which led to atrocities such as `BEAMMONSWICPEDSVLD` instead of a more readable `BeamMonSwicPedVld`. If NOvA is also forced (or desires for reasons of uniformity) to use solely upper (or lower) case it might behoove the table designers to enforce a scheme using underscores (e.g. `BEAM_MON_SWIC_PED_VLD`). Uniformity in naming styles helps to avoid user confusion by allowing them to predict how things will be formatted.

Other naming constraints (forced on MINOS by Oracle): limit of 30 characters for table name or field (including `VLD`); the need to avoid common words for names of fields (e.g. `VIEW`, `MODE`, many others)

## Examples

### Use Case 1: normal queries

```
TTimeStamp when(2011,9,9,12,13,11);
VldContext vc(Detector::kNear,SimFlag::kData,when);
DbiResultPtr<DbiDemoData1> myResPtr(vc);

for ( UInt_t irow = 0; irow < myResPtr.GetNumRows(); ++irow) {
    const DbiDemoData1* dddl = myResPtr.GetRow(irow);
    // process each row
}

// results pointers are lightweight, but also reusable
VldContext newvc(...);
UInt_t nrows = myResPtr.NewQuery(newvc);

// letting myResPtr handle (proxy) go out of scope does not
// invalidate the table cache behind it
```

### Use Case 2: extended queries

```
TTimeStamp tsStart(2001,9,9,15,0,0);
TTimeStamp tsEnd(2012,7,25,12,13,11);
DbiSqlContext context(DbiSqlContext::kStarts, tsStart, tsEnd,
                    Detector::kNear|Detector::kFar,SimFlag::kData);
// besides extended window, also sub-select w/ WHERE condition
DbiResultPtr<DbuSubRunSummary>
    runsResPtr("DBUSUBRUNSUMMARY",context,Dbi::kAnyTask,
              "RUNTYPENAME = 'NormalData'");

// get a row
const DbuSubRunSummary* srs = runsResPtr.GetRow(0);

// ask about the rows validity
const DbiValidityRec* vldRec = runResPtr.GetValidityRec(srs);
```

## Bibliography

Author Last Name, First Name. "Book Title or Reference Title." City: Publisher, Date.

MINOS DBI Package Overview

[http://www-numi.fnal.gov/offline\\_software/srt\\_public\\_context/WebDocs/Package\\_Overviews/DatabaseInterface.html](http://www-numi.fnal.gov/offline_software/srt_public_context/WebDocs/Package_Overviews/DatabaseInterface.html)

MINOS DBI Package Rationale

[http://www-numi.fnal.gov/offline\\_software/srt\\_public\\_context/DatabaseInterface/doc/dbi\\_PR.html](http://www-numi.fnal.gov/offline_software/srt_public_context/DatabaseInterface/doc/dbi_PR.html)

MINOS DBI User Manual

[http://www-numi.fnal.gov/offline\\_software/srt\\_public\\_context/doc/UserManual/node9.html](http://www-numi.fnal.gov/offline_software/srt_public_context/doc/UserManual/node9.html)

Representation of times and dates

[http://en.wikipedia.org/wiki/ISO\\_8601](http://en.wikipedia.org/wiki/ISO_8601)