LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Technical Note **LIGO-T990101-02 - E** 11/30/99

Table Definitions for LDAS Metadata / Event Database

P. Shawhan

Distribution of this draft:

LDAS Group; LIGO Scientific Collaboration

This is an internal working note
of the LIGO Project

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

Contents

1 INTRODUCTION

The LIGO Data Analysis System (LDAS) includes a database system which is intended to store information of various types, including metadata and event lists. "Metadata" may be defined as information which facilitates access to, or interpretation of, the regular data stream. For example, it includes a record of the configuration of the apparatus at any given point in time, as well as a lookup table which indicates where the raw data from a given time interval may be found. We will also consider the term to include summary information derived from the data, although some would argue that this should still be considered to be data, not metadata. LIGO "events" will be generated by various programs, both on-line and off-line, which examine the gravity-wave channel for signatures of astrophysical interest or which monitor instrumental or environmental channels for transients of terrestrial origin.

An earlier draft document, LIGO-T980070, "LIGO Metadata, Event and Reduced Data Requirements—Preliminary", laid out a conceptual data-usage model and a set of requirements for the LDAS database. In the course of the full design process, some of these concepts and requirements have been modified, so this document should be considered to supersede the earlier one.

The LDAS database system has commercial database software at its core, with LIGO software layers on top to provide a more convenient and uniform user interface, do format translations, etc. The IBM DB2 "Universal Database" has been selected for the underlying database software, at least for the foreseeable future. This is a relational database which uses the SQL query language to insert and retrieve information. All information is stored in "tables", each with a fixed number of pre-defined columns and a variable number of rows which represent database entries. SQL provides ways to ensure the self-consistency of the database contents, in the form of uniqueness and "referential integrity" constraints; the latter may be thought of as defining certain relationships between different tables. It also provides a flexible query syntax to retrieve information from one or more tables; however, LDAS will provide a graphical user interface so that users can retrieve information in various ways without having to know SQL.

This document presents a specific design for the underlying database, in the form of table definitions expressed as SQL scripts, plus some discussion. It builds on an earlier implementation by Xiao Hu and others, correcting a number of shortcomings and providing additional capabilities, in consultation with members of the LDAS group and with John Zweizig. This is intended to be a stable, usable design; however, with DB2 it is possible to add tables or to modify table definitions and constraints (with some limitations) even after data has been inserted, should this prove to be necessary or advantageous.

The conclusion of this documentation includes a list of open issues which need to be addressed.

2 SCOPE OF THE LDAS DATABASE

The design presented in this document covers the following database purposes:

- Metadata concerning raw frame data (including lists of "framesets" and "segments", to be described in Section 5)
- Summary information (statistics, spectra, etc.) for time intervals of arbitrary length

- List of events generated by the Global Diagnostics System (GDS) representing transients detected in instrumental/environmental channels, etc.
- Lists of astrophysical event candidates

The database will likely be used for a variety of other purposes as well. For example, it may be used to log information (statistics, etc.) from LDAS and/or data-acquisition processes. It could also be used to record events from other sources, such as lists of earthquakes, gamma-ray bursts, neutrinos observed in underground detectors, etc. On the other hand, it may be the case that these things are adequately stored elsewhere, e.g. in searchable catalogs or parsable ASCII files, in which case it may not be necessary to ingest them into the LDAS database; there is a trade-off between the advantage of being able to use SQL queries to retrieve events (by time interval, magnitude, etc.) vs. the disadvantage of having to set up DB2, the metadataAPI, and the user interface to handle these other event types. Decisions on these other possible database functions can be deferred, since they do not impact the design for the functions described in this document.

The exact scope of the LDAS database will be determined by the LIGO Scientific Collaboration (LSC) in consultation with LDAS personnel. Once the database design is finalized and put into use, a change to the design (for example, the addition of a new table) generally would require corresponding changes to the LIGO metadataAPI and user-interface software layers, and should not be undertaken without good reason. Any proposed change should be presented in a document with the following information:

- Motivation for the proposed change
- Description of the proposed change
- Contact person(s)
- Name(s) of program(s) which will write to the database
- Description of output generated by the program(s) (event detection algorithm, method used to calculate each output variable, etc.), with references to other documentation if appropriate
- Description of new database table(s) needed (if any)
- Volume of data to be stored in the database (essential to ensure adequate storage space)
- User-interface requirements (common queries, presentation of results, etc.)
- Requirements for replication to other database installations

Approved changes will be implemented by LDAS personnel, with help from the proposer(s) if needed. The proposal document will be made accessible from the LDAS web server for permanent reference. The tables in the current design, presented below, should be subjected to the same documentation requirement before they are put into active use. For tables which are used by multiple programs (e.g. astrophysics event lists), each program should be documented.

General guidelines for including information in the database, and the procedure for considering proposed changes, should be established by the LSC.

3 OVERVIEW OF DATABASE DESIGN

The rest of this document presents table definitions to store appropriate information for the functions listed at the beginning of Section 2. A graphical overview of the tables, indicating the relationships between them, is shown in Figure 1. The SQL code to define these tables, as it appears in the later sections of this document, is currently located on the Caltech LIGO cluster in `/home/pshawhan/metadb/design_v02`. (Besides the individual files, there is a file called `all_tables.sql` which contains the code for all of the tables.) If you are viewing this document as a pdf file, then clicking on a table name in Figure 1 should cause the table definition to be displayed by your web browser.

Besides listing the columns contained within each table, one major design consideration is the choice of the "primary key", which is required to be unique for each row. DB2 maintains an index based on the primary key columns, so queries based on the columns in the primary key will be performed more efficiently. In many cases, additional indexes are created to speed up common queries which use columns other than those in the primary key. (These indexes are generally used to promote "clustering" of related rows in the same physical pages on disk, further improving access speed.) We also make extensive use of "foreign keys", which help ensure the integrity of the database by checking that the foreign key value(s) in one table match an entry in a unique index (usually the primary key) of another table. All of these indexes and constraints add a certain amount of overhead for inserting rows into the database, but our philosophy has been to optimize more for retrieval than for insertion. (If necessary, indexes and constraints can be added or deleted later on, even after data has been inserted.) Along the same lines, we have sometimes made the database tables somewhat larger than strictly necessary (with extra columns, or with fixed-length rather than variable-length character strings) in the interest of faster querying.

Programs which write into the database must provide certain necessary information for each entry. LDAS will need to provide a software interface which makes it easy for the user to assemble this information, format it appropriately (i.e. as an XML file) and transmit it to the database. Currently it is thought that there will be two available methods for doing this:

- A program may submit multiple XML files while it is running. The first file, generated during initialization, contains basic process information along with the values of input parameters and descriptions of any filters used. Other files, generated and submitted periodically, contain blocks of events, summary information, or whatever. This method is appropriate for programs which run continuously, such as on-line event searches or monitoring programs. It requires the program to query the database once (before submitting the second XML file) in order to retrieve the unique ID which has been assigned to it; there are some timing issues here to be worked out.
- A program may assemble all generated database records in a single XML file, then submit it after the program finishes. This method does not have to carry out any synchronous communication with the database. Thus, it is appropriate for off-line programs which operate on a fixed amount of data. It also allows the user to check the output of the program before submitting the XML file to the database.

One issue is the mechanism for generating unique values for certain entries; this is nontrivial because it is often the case that a given unique value must appear in more than one database table,

LDAS Metadata / Event Database Tables

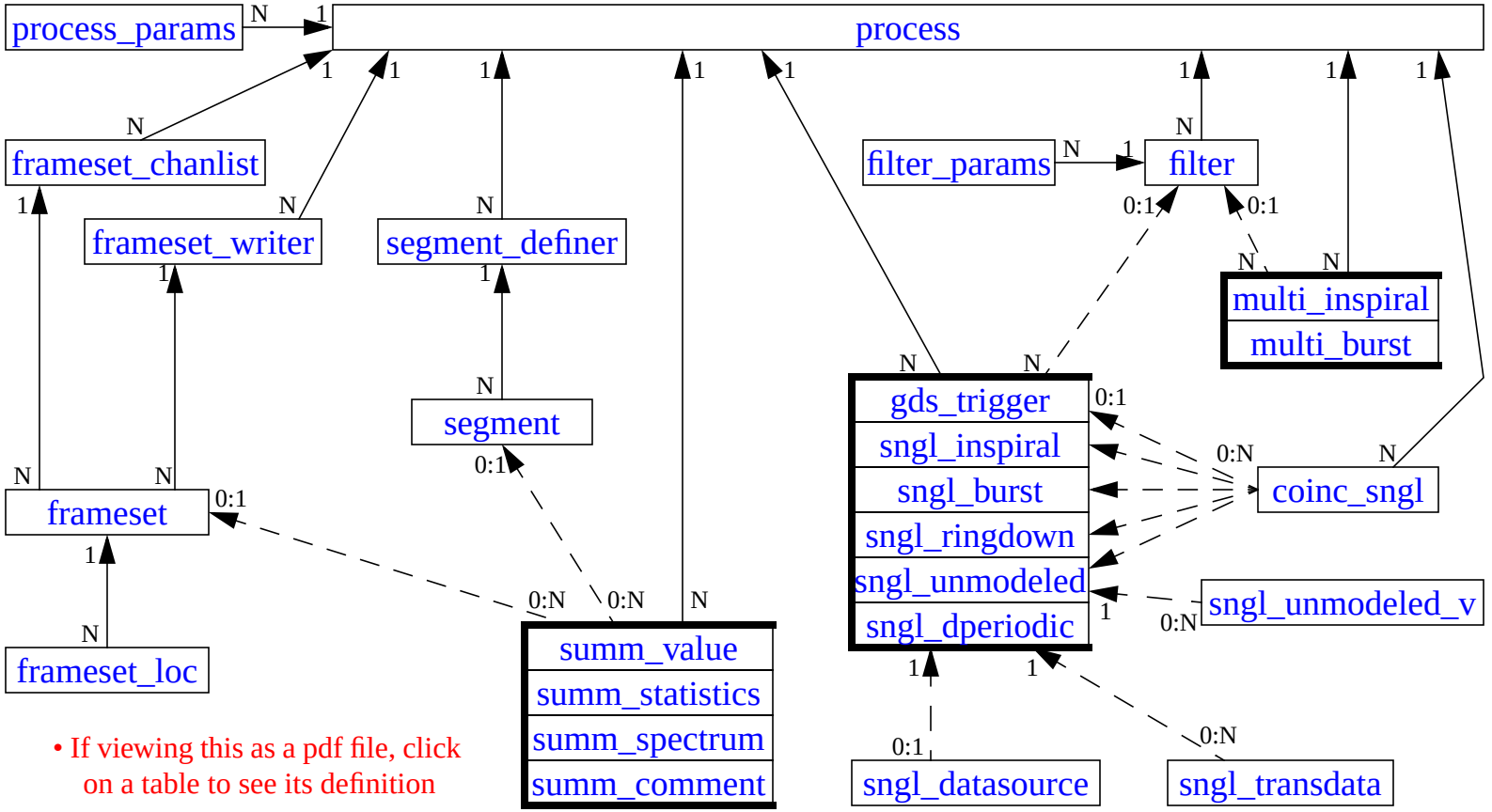

Figure 1: Graphical overview of database tables defined in this document.

Arrows indicate “foreign key” referential integrity constraints. Values near the ends of the arrows (1, N, etc.) indicate the possible multiplicities. Dashed lines indicate optional relationships. Stacked tables (grouped by thick lines) have common relationships with other tables, except for relationship arrows connecting along the right edge. Examples: 1) Each segment is related to one segment_definer; 2) Each segment_definer is (generally) related to many segments; 3) A frameset is related to one frameset_chanlist entry and to one frameset_writer; 3) A summ_value (or summ_statistics, etc.) entry may or may not be related to a segment and/or a frameset; 4)A single-interferometer event (gds_trigger, sngl_inspiral, etc.) entry may be related to up to one sngl_datasource and/or any number of sngl_transdata entries.

but it is unreasonable to expect the user program to carry out two-way communication with the database for each event candidate, etc. Thus, the metadataAPI will include code to request unique values from DB2 as necessary and insert them into each of the relevant database entries. A site-specific "creator database" tag (which is necessary to ensure uniqueness with certainty, since it is possible for different database servers to generate the same "unique value") is automatically set by DB2. (We need to check whether this is propagated correctly when data is replicated.)

4 PROCESS INFORMATION

The `process` table stores information about a specific invocation of a program. Any program which intends to insert information into the database must make an entry in the `process` table during its initialization stage. A unique `process_id` will be assigned to the program; all database entries made by the program will then include this `process_id`, to keep a "paper trail" of who has modified the database. All other tables have a foreign-key relationship with the `process_id` table, either directly or through another table. Programs which merely read from the database do not have to add an entry to the `process` table.

Input parameters for a given process (e.g. from a configuration file) are stored in the `process_params` table. Each row in this table stores the name, type, and value of one parameter. Typically there are many parameters for a given process, leading to many rows in the `process_params` table. For uniformity, we will always use this "generic" method of recording process parameters, even though one could imagine more efficient storage schemes customized for individual processes.

When a process exits, it should modify its entry in the `process` table to record the end time.

4.1. `process` Table Definition

```
CREATE TABLE process
(
-- This table contains information about a specific invocation of a program.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- BASIC INFORMATION ABOUT THE PROGRAM
-- Program name
      program            CHAR(16) NOT NULL,
-- Version of the program
      version            CHAR(8) NOT NULL,
-- Where the program is stored in the cvs repository
      cvs_repository     VARCHAR(64),
-- Time when the program was entered into the cvs repository (GPS seconds)
      cvs_entry_time     INTEGER,
-- User comment which describes the program
      comment            VARCHAR(240),

-- INFORMATION ABOUT THIS INVOCATION OF THE PROGRAM
-- Flag to indicate whether it was run on-line (1) or off-line (0)
      is_online          INTEGER NOT NULL WITH DEFAULT 0,
-- Node on which it was run
      node               VARCHAR(48) NOT NULL,
```

```
-- Unix username
      username          CHAR(16) NOT NULL,
-- Unix process ID
      unix_procid       INTEGER NOT NULL,
-- Start time (GPS seconds)
      start_time        INTEGER NOT NULL,
-- End time (GPS seconds); not filled initially, but filled when process
-- exits gracefully
      end_time          INTEGER,

-- Unique id to identify this program, generated by DB2 (not unix process ID)
      process_id        CHAR(13) FOR BIT DATA NOT NULL,
-- Parameter set identifier.  Permits an association between multiple
-- invocations of a program which use the same set of input parameters.
-- Probably not filled initially (because of timing issues if multiple
-- processes start at about the same time), but updated later.
      param_set         INTEGER,

-- INFORMATION ABOUT THE DATA HANDLED BY THIS PROGRAM
-- Interferometer(s) from which data comes.  In general, a process
-- knows in advance what interferometer's data it is analyzing.
-- (This is necessary to retrieve frameset metadata, for instance.)
-- Make this variable long enough to indicate multiple interferometers.
-- (e.g. "H1H2L1V1")
      ifos              CHAR(12),

      CONSTRAINT process_pk
      PRIMARY KEY (program, start_time, node, unix_procid),

-- Need to explicitly create a unique index on process_id (since we are not
-- using it as the primary key) so that other tables can use it as a foreign
-- key.
      CONSTRAINT process_uni_pid
      UNIQUE (creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
```

4.2. `process_params` Table Definition

```
CREATE TABLE process_params
(
-- This table contains input parameters for programs.

-- Database which created this entry
      creator_db        INTEGER NOT NULL WITH DEFAULT 1,

-- Program name
      program           CHAR(16) NOT NULL,
-- Unique process ID (not unix process ID)
      process_id        CHAR(13) FOR BIT DATA NOT NULL,

-- The triplet will store the value of a single parameter.
-- One example might be param = "mass", type="REAL"
-- and value = "12345.6789"
      param             VARCHAR(32) NOT NULL,
```

```
      type              VARCHAR(16) NOT NULL,
      value             VARCHAR(64) NOT NULL,

-- The program name is not necessary to make the primary key unique, but
-- one will generally make a query about a particular program.
      CONSTRAINT procparams_pk
      PRIMARY KEY (program, creator_db, process_id, param),

-- Foreign key relationship to process table.  The 'ON DELETE CASCADE'
-- modifier means that if a row in the process table is deleted, then
-- all its associated parameters are deleted too.
      CONSTRAINT procparams_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
          ON DELETE CASCADE
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
```

5 METADATA ABOUT RAW DATA

When preparing to do a diagnostic or science analysis, there are two main things that one wants to know:

- Where can I get the raw data for my analysis?
- What sections of the data are appropriate to use for my analysis?

These complementary concepts are addressed by two separate parts of the database design, which we call “framesets” and “segments”, respectively.

5.1. Physical Data Units: Framesets

The data acquisition system (DAQS) builds raw data into the standard frame format and writes files containing multiple frames. The DAQS writes three different types of frame files (“full”, “analysis”, and “trend”) for each interferometer; off-line programs may repackage the data in various other ways. We refer to each file as a “frameset”; it has a unique name (e.g. `H1-627332233.F`) which indicates the interferometer, the GPS time at the beginning of the file, and a suffix indicating the contents (`F` for “full DAQS output” in this case). Framesets from the same interferometer and with the same general contents are grouped together as a “frameset group” for easy specification; for example, a user may request all framesets in the group H1.F between the times 627332233 and 627418633. (The channel list, sampling rates, and instrument configuration *must* be the same for all frames in a given frameset. However, minor differences in channel lists should not prevent two framesets from being considered as part of the same frameset group, if they were created for the same purpose. Also, a frameset is permitted to be missing one or more frames, though in practice we may decide to start a new frameset when we encounter a missing frame.) Basic information about each frameset is recorded in the `frameset` table, while information about its origin and contents is stored in the `frameset_writer` and `frameset_chanlist` tables, respectively. Frameset locations (on disk, on tape, in the hpss

archive, etc.) are listed in the `frameset_loc` table; note that a given frameset may be located in more than one place, and the place(s) may change over time. For example, the physical location of a tape should be updated when it is sent from a site to the central archive, while framesets which are deleted from disk should have their `frameset_loc` entries deleted as well.

Each process which intends to create framesets must, at initialization time, make an entry in the `frameset_writer` table. This table can then be queried to get a list of frameset groups, though one must use the `DISTINCT` modifier with the query to get the desired list since there can be multiple processes which created parts of the same frameset group. (We might want to add a `frameset_groups` table to allow a more straightforward listing and to keep summary information about each frameset group, such as the number of framesets, earliest start time, latest stop time; it should be possible to set up DB2 triggers to maintain such a table automatically.)

In the current design, the channel list applicable for a given frameset is stored in the `frameset_chanlist` table in a character large object ("CLOB"), along with the sampling rate for each channel. Derived pseudo-channels are perfectly acceptable as long as each is given a unique descriptive name. DB2 allows one to ask whether the CLOB includes a specific channel, using the `LIKE` predicate, although there is no good way to check the sampling rate.

Note that the frameset-related tables support the creation of new framesets with arbitrary contents. Thus, for example, one might extract the seismic channels from the full data stream and write them to framesets with names like `H0-627332233.SEIS` (i.e. frameset group `H0.SEIS`). Similarly, any "reduced data sets" are treated the same as the standard DAQS output streams as far as the database is concerned. For any given analysis, the list of channels needed can, in principle, be used to construct a database query to determine what framesets should be retrieved from the archive. This makes it possible to consider building a "smart" archive which splits up the input data into groups of related channels for storage on tape, then assembles a customized data stream in response to a user request with minimal waste of input/output resources. Whether to implement such a scheme has yet to be determined.

5.1.1. `frameset_chanlist` Table Definition

```
CREATE TABLE frameset_chanlist
(
-- List of channels included in a frameset.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Unique process ID of the process which submits the chanlist info
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- Frameset group for which this channel list applies
-- (But note that one frameset group could have several different channel
-- lists for data taken at different times)
      frameset_group     CHAR(48) NOT NULL,
-- Validity range (in GPS seconds) for this channel list
      start_time         INTEGER,
      end_time           INTEGER,

-- Unique identifier for this channel list
      chanlist_id        CHAR(13) FOR BIT DATA NOT NULL,
```

```
-- Channel list, with modifiers and sampling rates.  Separated by spaces.
-- Examples of items in the list:
--      'H2:PEM-LVEA_SEISX 256'                Raw data stream
--      'H2:PEM-LVEA_SEISX 16'                 Decimated
--      'H2:PEM-LVEA_SEISX.LOWPASS(10.0) 16'   Filtered and decimated
-- List is stored in a Character Large OBject (CLOB).
      chanlist          CLOB(512K),
-- Length of channel list in bytes (0 if the CLOB is empty)
      chanlist_length   INTEGER NOT NULL,

      CONSTRAINT fschanlist_pk
      PRIMARY KEY (creator_db, chanlist_id),

      CONSTRAINT fschanlist_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- The following line ensures that replication will work properly on any
-- LONG VARCHAR columns that we might add to this table in the future.
ALTER TABLE frameset_chanlist
    DATA CAPTURE CHANGES INCLUDE LONGVAR COLUMNS
;
```

5.1.2. `frameset_writer` Table Definition

```
CREATE TABLE frameset_writer
(
-- List of processes which create framesets.  Note that multiple processes
-- can write framesets in the same frameset_group.

-- Database which created this entry
      creator_db        INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROGRAM WHICH IS WRITING FRAMESETS
-- Program name
      program           CHAR(16) NOT NULL,
-- Unique process ID
      process_id        CHAR(13) FOR BIT DATA NOT NULL,

-- INFORMATION ABOUT THE FRAMESETS BEING WRITTEN
-- Base name for this group of framesets (e.g. 'H2.F')
      frameset_group    CHAR(48) NOT NULL,
-- Source of this data.  Use 'DAQS' for original raw data, otherwise the
-- name of the frameset_group from which this new frameset_group is derived.
-- If the new frameset_group is derived from multiple frameset_groups, list
-- them all, separated by spaces.
      data_source       VARCHAR(240) NOT NULL,
-- Interferometer(s) with information in the frames
      ifos              CHAR(12),

-- Optional user comment about this frameset_group
      comment           VARCHAR(240),
```

```
      CONSTRAINT fswriter_pk
      PRIMARY KEY (frameset_group, creator_db, process_id),

      CONSTRAINT fswriter_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Also create a clustering index for quicker scanning
CREATE INDEX fswriter_ind_fsgrp ON frameset_writer(frameset_group) CLUSTER;
```

5.1.3. `frameset` Table Definition

```
CREATE TABLE frameset
(
-- A “frameset” is a set of data frames contained in the same file.  It is
-- the smallest unit of raw data which can be read and analyzed, since
-- dictionary information stored at the beginning of the file is needed to
-- interpret frames appearing later in the file.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Unique process ID of the process which wrote this frameset
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- Base name for the group to which this frameset belongs (e.g. ‘H2.F’)
      frameset_group     CHAR(48) NOT NULL,

-- INFORMATION ABOUT THE CONTENTS OF THIS FRAMESET
-- Database which created chanlist entry (which may be different from the
-- database creating this frameset entry)
      chanlist_cdb       INTEGER NOT NULL,
-- Channel set identifier
      chanlist_id        CHAR(13) FOR BIT DATA,
-- Frameset start and end times, in GPS seconds and nanoseconds.
-- Note that end_time is the time at the END of the last frame
-- included in this frameset.  Thus, these two times always differ
-- by at least the length of one frame.
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,
-- Number of frames in frameset
      n_frames           INTEGER NOT NULL,
-- Number of missing frames within frameset
      missing_frames     INTEGER NOT NULL,
-- Size of the frameset, in bytes
      n_bytes            INTEGER NOT NULL,

-- FILENAME FOR THIS FRAMESET
-- This uniquely identifies the frameset.  No two different framesets
-- can have the same name (but if there are multiple copies of a
-- frameset in different places, they would have the same name).
-- Normally the name will consist of an interferometer code, a GPS time,
```

```
-- and a suffix which indicates the frame type, e.g. 'H2-628318531.F'.
      name              VARCHAR(80) NOT NULL,

-- Note that (frameset_group,start_time) should be sufficient to uniquely
-- identify a frameset, but we include end_time in the primary key to
-- facilitate faster queries.
      CONSTRAINT frameset_pk
      PRIMARY KEY (frameset_group, start_time, end_time),

-- Also create a unique index for the frameset name
      CONSTRAINT frameset_uni_name
      UNIQUE (name),

      CONSTRAINT frameset_fk_fswrit
      FOREIGN KEY (frameset_group, creator_db, process_id)
          REFERENCES frameset_writer(frameset_group, creator_db, process_id),

      CONSTRAINT frameset_fk_chanli
      FOREIGN KEY (chanlist_cdb, chanlist_id)
          REFERENCES frameset_chanlist(creator_db, chanlist_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
```

5.1.4. `frameset_loc` Table Definition

```
CREATE TABLE frameset_loc
(
-- Table to keep track of frameset locations.  There can be more than one
-- location for a given frameset.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Frameset name
      name               VARCHAR(80) NOT NULL,

-- Media type (disk, hpss, 8mm, dvd,...)
      media_type         CHAR(16) NOT NULL,
-- The node with the data (i.e. computer on which disk is mounted, or hpss
-- server); OR, in the case of removable media, this should be the media label
      node               VARCHAR(48) NOT NULL,
-- Physical location of removable media (e.g. LHO, LLO, transit, CACR, ...)
      media_loc          VARCHAR(48),
-- Status of the media (e.g. OK, broken, ...)
      media_status       CHAR(8) WITH DEFAULT 'OK',
-- Full path and actual file name.  OR, in the case of a tape without named
-- files, this should just be the frameset name again.
      fullname           VARCHAR(128) NOT NULL,
-- File number (for tapes without named files)
      filenum            INTEGER,
-- Decompression command (e.g. 'gunzip *.tar.gz; tar xf *.tar', where '*'
-- is replaced by the frameset name)
      decompress_cmd     VARCHAR(128),

      CONSTRAINT fsloc_pk
```

```
      PRIMARY KEY (node, fullname),

      CONSTRAINT fsloc_fk_frameset
      FOREIGN KEY (name) REFERENCES frameset(name)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Also create a clustering index for quicker name lookup
CREATE INDEX fsloc_ind_name ON frameset_loc(name) CLUSTER;
```

5.2. Logical Data Units: Segments

Framesets are not a very suitable unit for characterizing the data, since the boundaries between framesets are often arbitrary, determined by trying to make files of a reasonable size. Also, a given frameset refers to a specific set of channels, so there can be multiple framesets for any given time interval. Therefore, we define a separate construct called a "segment" which is intended to indicate a time interval with a common property, e.g. an interval during which the Hanford 2K interferometer is locked. Segments with the same property belong to a "segment group" with a descriptive name (e.g. "H2-locked"). (We also allow for a version number in case it is necessary to redetermine the segments in a segment group; the default should be to use the latest existing version.) The database contains `segment_definer` and `segment` tables which are analogous to the `frameset_writer` and `frameset` tables described above. (It would be nice if there were also a `segment_group` table, with one row per segment group, containing information like the number of segments, first start time and last end time, and the fraction of the data included in segments.)

A typical analysis would loop over the segments in some segment group, determine what framesets need to be read to do the desired analysis for each segment (depending on what channels are needed), and retrieve the data. It would be desirable to provide a mechanism to interpret logical combinations of segment groups (e.g. "H1-locked and H2-locked and L1-locked") to produce a derived set of segments.

Note that segments can be used to represent elements of the "state vector" for an interferometer. This may, in fact, be the best way to store state-vector information, depending on how many elements are envisioned. The current design does not provide any other place to store "state vectors".

5.2.1. `segment_definer` Table Definition

```
CREATE TABLE segment_definer
(
-- List of processes which define segments.  Note that multiple processes
-- can define segments in the same segment_group.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROGRAM WHICH IS DEFINING SEGMENTS
-- Program name
      program            CHAR(16) NOT NULL,
```

```
-- Unique process ID
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- INFORMATION ABOUT THE SEGMENTS BEING DEFINED
-- Descriptive name for this group of segments (e.g. 'H2-locked')
      segment_group      CHAR(48) NOT NULL,
-- Version number for segment group (to allow re-evaluation)
      version            INTEGER NOT NULL,
-- Interferometer(s) for which these segments are meaningful
      ifos               CHAR(12),

-- Optional user comment about this segment_group
      comment            VARCHAR(240),

      CONSTRAINT segdef_pk
      PRIMARY KEY (segment_group, version, creator_db, process_id),

      CONSTRAINT segdef_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Also create a clustering index for quicker scanning
CREATE INDEX segdef_ind_seggrp
    ON segment_definer(segment_group, version) CLUSTER;
```

5.2.2. `segment` Table Definition

```
CREATE TABLE segment
(
-- A "segment" is a time interval which is meaningful for some reason.  For
-- example, it may indicate a period during which an interferometer is locked.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Unique process ID of the process which defined this segment
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- Segment group (e.g. 'H2-locked') and version to which this segment belongs
      segment_group      CHAR(48) NOT NULL,
      version            INTEGER NOT NULL,

-- INFORMATION ABOUT THIS SEGMENT
-- Segment start and end times, in GPS seconds and nanoseconds.
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,

-- Note that (segment_group,start_time) should be sufficient to uniquely
-- identify a segment, but we include end_time in the primary key to
-- facilitate faster queries.
      CONSTRAINT segment_pk
      PRIMARY KEY (segment_group, version, start_time, end_time),
```

```
      CONSTRAINT segment_fk_segdef
      FOREIGN KEY (segment_group, version, creator_db, process_id)
          REFERENCES
          segment_definer(segment_group, version, creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
```

6 SUMMARY INFORMATION

It will be useful to be able to store various summary information about intervals of data. At present, tables are provided to store arbitrary scalar values; standard channel statistics (max, min, mean, rms, etc.); spectra (with sizes up to 512 kilobytes); and text comments. The time interval to which the summary information applies may correspond to a frameset listed in the `frameset` table or to a segment listed in the `segment` table, but neither is required. For example, one may wish to store the rms seismic noise (perhaps after applying a low-pass filter) for each "H2-locked" segment, or simply for each 30-minute interval while data is being collected. The latter might be done on-line by a GDS data-monitoring process, for instance, which does not know how the DAQS is dividing up the data into framesets as it writes to disk.

6.1. `summ_value` Table Definition

```
CREATE TABLE summ_value
(
-- Table to record a value about a particular time interval

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH RECORDED THE VALUE
-- Program name
      program            CHAR(16) NOT NULL,
-- Unique process ID
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- TIME INTERVAL FROM WHICH THIS VALUE WAS CALCULATED
-- Group name for frameset which determined this time interval, if any
      frameset_group     VARCHAR(48),
-- Group and version of segment which determined this time interval, if any
      segment_group      VARCHAR(128),
      version            INTEGER,
-- Start and end times (in GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,

-- THE SUMMARY VALUE
-- Site or interferometer to which this applies (H0, H1, H2, L0, L1)
      ifo                CHAR(2) NOT NULL,
-- Descriptive name (does not have to indicate interferometer)
```

```
      name              CHAR(32) NOT NULL,
-- The value itself (must be a real number)
      value             REAL NOT NULL,
-- Optional comment
      comment           VARCHAR(80),

      CONSTRAINT summval_pk
      PRIMARY KEY (creator_db, process_id, ifo, name, start_time, end_time),

      CONSTRAINT summval_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Also create a clustering index for quicker queries
CREATE INDEX summval_ind_name
    ON summ_value(program, name, ifo, start_time) CLUSTER;
```

6.2. `summ_statistics` Table Definition

```
CREATE TABLE summ_statistics
(
-- Table to contain minimum, maximum, mean, rms, etc. for a single channel
-- (or pseudo-channel) for a specific time interval.

-- Database which created this entry
      creator_db        INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH PRODUCED THESE STATISTICS
-- Program name
      program           CHAR(16) NOT NULL,
-- Unique process ID
      process_id        CHAR(13) FOR BIT DATA NOT NULL,

-- TIME INTERVAL FOR WHICH THESE STATISTICS WERE CALCULATED
-- Group name for frameset which determined this time interval, if any
      frameset_group    VARCHAR(48),
-- Group and version of segment which determined this time interval, if any
      segment_group     VARCHAR(128),
      version           INTEGER,
-- Start and end times (in GPS seconds and nanoseconds)
      start_time        INTEGER NOT NULL,
      start_time_ns     INTEGER NOT NULL,
      end_time          INTEGER NOT NULL,
      end_time_ns       INTEGER NOT NULL,
-- Number of frames actually used to calculate statistics
      frames_used       INTEGER,
-- Number of samples from which sums were accumulated (needed so that one
-- can convert between raw sums and mean, rms, etc.)
      samples           INTEGER NOT NULL,

-- CHANNEL (OR PSEUDO-CHANNEL) NAME
-- The channel name should indicate the interferometer or site
      channel           CHAR(48) NOT NULL,
```

```
-- STATISTICS INFO
-- Minimum and maximum value of the channel during this time interval
      min_value          DOUBLE,
      max_value          DOUBLE,
-- Minimum and maximum CHANGE in the value of the channel
      min_delta          DOUBLE,
      max_delta          DOUBLE,
-- Minimum and maximum second-order finite difference
      min_deltadelta     DOUBLE,
      max_deltadelta     DOUBLE,
-- Mean, rms, 3rd and 4th moments
      mean               DOUBLE,
      rms                DOUBLE,
      moment3            DOUBLE,
      moment4            DOUBLE,

      CONSTRAINT summstat_pk
      PRIMARY KEY (creator_db, process_id, channel, start_time, end_time),

      CONSTRAINT summstat_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Also create a clustering index for quicker queries
CREATE INDEX summstat_ind_chan
    ON summ_statistics(channel, start_time) CLUSTER;
```

6.3. `summ_spectrum` Table Definition

```
CREATE TABLE summ_spectrum
(
-- Table to contain a summary spectrum derived from the data in a specific
-- time interval.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH PRODUCED THIS SPECTRUM
-- Program name
      program            CHAR(16) NOT NULL,
-- Unique process ID
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- TIME INTERVAL FROM WHICH THIS SPECTRUM WAS DERIVED
-- Group name for frameset which determined this time interval, if any
      frameset_group     VARCHAR(48),
-- Group and version of segment which determined this time interval, if any
      segment_group      VARCHAR(128),
      version            INTEGER,
-- Start and end times (in GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,
```

```
-- Number of frames actually used to create spectrum
      frames_used        INTEGER,

-- CHANNEL (OR PSEUDO-CHANNEL) NAME
-- The channel/pseudo-channel name should indicate the interferometer or site
      channel            CHAR(48) NOT NULL,

-- SPECTRUM DATA AND ASSOCIATED INFO
-- Spectrum type (descriptive name)
      spectrum_type      CHAR(32) NOT NULL,
-- The spectrum itself is stored in a Binary Large OBject (BLOB).
-- We specify COMPACT since we do not expect this ever to be updated.
      spectrum           BLOB(512K) COMPACT NOT NULL,
-- Length of the spectrum (in bytes)
      spectrum_length    INTEGER NOT NULL,

      CONSTRAINT summspect_pk
      PRIMARY KEY (creator_db, process_id, channel, spectrum_type,
          start_time, end_time),

      CONSTRAINT summspect_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Also create a clustering index for quicker queries
CREATE INDEX summspect_ind_chan
    ON summ_spectrum(channel, spectrum_type, start_time) CLUSTER;
```

6.4. `summ_comment` Table Definition

```
CREATE TABLE summ_comment
(
-- Table to attach a comment to a particular time interval

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH ADDED THE COMMENT
-- Program name
      program            CHAR(16) NOT NULL,
-- Unique process ID
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- TIME INTERVAL TO WHICH THIS COMMENT APPLIES
-- Group name for frameset which determined this time interval, if any
      frameset_group     VARCHAR(48),
-- Group and version of segment which determined this time interval, if any
      segment_group      VARCHAR(128),
      version            INTEGER,
-- Start and end times (in GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,
```

```
-- COMMENT AND ASSOCIATED INFO
-- Name of person who made comment
      submitter          VARCHAR(48) NOT NULL,
-- Timestamp at which comment was submitted (automatically set by DB2)
      submit_time        TIMESTAMP WITH DEFAULT CURRENT TIMESTAMP,
-- Interferometer or site to which the comment applies
      ifo                CHAR(2) NOT NULL,
-- The comment itself
      text               VARCHAR(1000) NOT NULL,
-- Unique ID for this comment (needed for primary key)
      summ_comment_id    CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT summcomm_pk
      PRIMARY KEY (creator_db, summ_comment_id),

      CONSTRAINT summcomm_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create an index based on time the comment refers to
CREATE INDEX summcomm_ind_time ON summ_comment(start_time, end_time);
```

7 GDS TRIGGERS AND ASTROPHYSICS EVENT CANDIDATES

There will be a number of on-line data monitoring programs running while data is being collected, as a part of the Global Diagnostics System (GDS) intended to detect transients in the apparatus which could cause false gravity-wave signals. A single monitoring program may look for transients in many different channels, or with multiple algorithms. Additional programs may be run off-line. (These programs may also generate summary information which will be stored in the database.) In general, a data monitoring program may filter signals in some way and look for a signature exceeding a pre-set threshold, generating a “trigger”. The bottom-line characteristics of a trigger may generally be expressed as a “size” and a “significance”, although the exact interpretation will depend on the trigger algorithm. Additional algorithm-specific results (e.g. from a multi-parameter fit) may be stored in binary form in a variable-length character array.

Astrophysics event searches (currently envisioned to include inspirals, bursts, ringdowns, “unmodeled-source” searches, and directed searches for periodic sources) will be done using two basic approaches. The first approach is to first identify candidate events (above some signal-to-noise threshold) in the gravity-wave signal from each individual interferometer, and then look for coincidences among the events found (i.e. among different interferometers and/or different event types). The first stage of this approach will nominally be done on-line in near-real-time, though there are likely to be off-line single-interferometer analyses as well. The second basic approach is to do a simultaneous analysis of the gravity-wave signals from multiple interferometers.

GDS triggers, and the various single- and multi-interferometer astrophysics searches, produce different information, so we define several database tables to store the results. However, they are

otherwise rather similar. (The use of the different terms “trigger” vs. “event” is entirely conventional.) Therefore, we let them all use the same tables to store filter descriptions, filter parameters, and additional information about the triggers/events found.

7.1. Filter Information

In general, a GDS data-monitor program or an astrophysics search program may filter the data in multiple ways, and each of these filters may have its own set of parameters. Therefore, we provide `filter` and `filter_params` tables to store this information in a structured way. These tables are filled at initialization time, and then each trigger/event can point to the filter it used (if any). Note that each invocation of a program adds rows to the `filter` table, even if they are the same as from the last invocation, because there is no straightforward way to check whether a given filter and set of parameter values is already in the database.

There is some freedom of choice about when one should enter filter descriptions and parameters into the `filter` and `filter_params` tables. For some monitoring programs, the filtering done may be implicit in the trigger algorithm, and any filter parameters could equally be considered to be process parameters. At the other extreme, a single inspiral-search process may apply hundreds or thousands of different filters, but each filter might be completely described by the pair of object masses which is stored with each event, so that an entry in the `filter` table would not provide any additional information. As a general rule, the `filter` table should be used whenever a process applies multiple distinct algorithms (or the same algorithm with multiple parameter sets), and the `filter_params` table should be used to store any filter-specific parameters. The `filter` table should also be used to store the name of the filter algorithm whenever it is not obvious, e.g. for all “unmodeled-source” search algorithms.

7.1.1. `filter` Table Definition

```
CREATE TABLE filter
(
-- Table of filter instances used by GDS and astrophysics-search programs.
-- Note that this table should contain an entry for each invocation of the
-- program (similar to the process table).  It is also possible for a single
-- process to use multiple filters with the same name but different
-- parameters.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROGRAM WHICH INITIALIZES THIS FILTER
-- Program name
      program            CHAR(16) NOT NULL,
-- Program start time (GPS seconds)
      start_time         INTEGER NOT NULL,
-- Process ID
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- INFORMATION ABOUT THIS FILTER
-- Filter name (e.g. “FFT”)
      filter             CHAR(32) NOT NULL,
-- Unique identifier for this invocation of the filter
      filter_id          CHAR(13) FOR BIT DATA NOT NULL,
```

```
-- Parameter set identifier.  Permits an association between multiple
-- invocations of a filter which use the same set of input parameters.
-- Probably not filled initially, but updated later.
      param_set          INTEGER,

      CONSTRAINT filter_pk
      PRIMARY KEY (creator_db, filter_id),

-- Foreign key relationship to process table.  The 'ON DELETE CASCADE'
-- modifier means that if a row in the process table is deleted, then
-- all its associated filters are deleted too.
      CONSTRAINT filter_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id)
          ON DELETE CASCADE
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on program name and start time
CREATE INDEX filter_ind_prog ON filter(program, start_time) CLUSTER;
```

7.1.2. `filter_params` Table Definition

```
CREATE TABLE filter_params
(
-- This table contains parameters for filters.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Filter name
      filter             CHAR(32) NOT NULL,
-- Unique process ID (not unix process ID)
      filter_id          CHAR(13) FOR BIT DATA NOT NULL,

-- The triplet will store a single parameter.
-- One example might be name = "pole1", type="REAL"
-- and value = "12345.6789"
      param              VARCHAR(32) NOT NULL,
      type               VARCHAR(16) NOT NULL,
      value              VARCHAR(64) NOT NULL,

      CONSTRAINT filtparam_pk
      PRIMARY KEY (creator_db, filter_id, param),

-- Foreign key relationship to filter table.  The 'ON DELETE CASCADE'
-- modifier means that if a row in the filter table is deleted, then
-- all its associated parameters are deleted too.
      CONSTRAINT filtparam_fk_pid
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
          ON DELETE CASCADE
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
```

7.2. GDS Triggers

As mentioned earlier, there are likely to be both on-line and off-line data monitoring programs. They can all use the same database table. (Information about whether a process was run on-line or off-line is stored in the `process` table.)

7.2.1. `gds_trigger` Table Definition

```
CREATE TABLE gds_trigger
(
-- Triggers generated by data monitoring programs (on-line or off-line).

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROGRAM AND FILTER WHICH GENERATED THIS TRIGGER
-- Process ID
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Unique descriptive trigger name.
      name               CHAR(32) NOT NULL,
-- Sub-type of the trigger (e.g. a particular channel name?)
      subtype            CHAR(32) NOT NULL,

-- INFORMATION ABOUT THIS PARTICULAR TRIGGER INSTANCE
-- Site or interferometer to which this trigger applies (H0, H1, H2, L0, L1)
      ifo                CHAR(2) NOT NULL,
-- Time when the trigger detects a transient (GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
-- Duration of the transient (seconds)
      duration           REAL NOT NULL,
-- Priority of the trigger (meaning has yet to be defined)
      priority           INTEGER NOT NULL,
-- Bitmask indicating what was done with this trigger (sent to operator
-- console, logged in database, etc.)
      disposition        INTEGER,
-- Bottom-line trigger results, useful for querying.  Exact interpretation
-- will depend on the particular trigger type.
      size               REAL,
      significance       REAL,
      frequency          REAL,
-- Full result, e.g. parameters from a fit.  User-defined binary format.
-- This will generally be very small (or NULL), so we prefer to store this
-- in a VARCHAR rather than in a BLOB.
      binarydata         VARCHAR(1024) FOR BIT DATA,
-- Length of full result, in bytes (0 if no full_result is recorded)
      binarydata_length  INTEGER NOT NULL,
-- A unique identifier for this trigger
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT gdstrig_pk
      PRIMARY KEY (creator_db, event_id),
```

```
      CONSTRAINT gdstrig_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT gdstrig_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time, etc.
CREATE INDEX gdstrig_ind_time ON gds_trigger(name, ifo, start_time) CLUSTER
;
-- Create an SQL trigger so that if a gds_trigger entry is deleted, any
-- associated sngl_datasource and/or sngl_transdata entries are deleted too.
-- Must be done this way because there is no foreign-key relationship.
CREATE TRIGGER gdstrig_tr_del
    AFTER DELETE ON gds_trigger
    REFERENCING OLD AS o
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
       DELETE FROM sngl_datasource
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
       DELETE FROM sngl_transdata
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
    END
;
```

7.3. Single-Interferometer Astrophysics Event Candidates

As mentioned earlier, these database tables are designed for recording the results of searches based on the gravity-wave channels from individual interferometers. Thus, interpretation which depends on post-processing or combining information from multiple interferometers (e.g. sky coordinates) is not included in these tables. The “properties of the event” columns are intended to contain the quantities reported by the event-finding algorithms, as they are currently understood. It is less clear what properties would be reported by “unmodeled-source” searches, so an additional table (`sngl_unmodeled_v`) is provided as a place to store an arbitrary set of result values, if needed.

“Directed” searches for a pre-selected set of possible periodic sources are expected to be performed on-line. In this case an “event” is most likely a long time interval (several days?) which is integrated over to look for a significant signal.

7.3.1. `sngl_inspiral` Table Definition

```
CREATE TABLE sngl_inspiral
(
-- Event table for single-interferometer binary-inspiral search.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,
```

```
-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS EVENT
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Interferometer
      ifo                CHAR(2) NOT NULL,

-- TIME OF THE EVENT
-- The coalescence time of this inspiral event (GPS seconds and nanoseconds)
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,
-- The time duration of this inspiral event (seconds)
      duration           REAL NOT NULL,
-- Time when filter output reaches maximum value (GPS seconds and nanoseconds)
      fout_peak_time     INTEGER NOT NULL,
      fout_peak_time_ns  INTEGER NOT NULL,
-- duration of filter used to produce event (seconds)
      filter_duration    REAL NOT NULL,

-- PROPERTIES OF THE EVENT
-- Absolute signal amplitude (fractional strain)
      amplitude          REAL NOT NULL,
-- Mass of the larger compact stellar object (in solar mass units)
      mass1              REAL NOT NULL,
-- Mass of the smaller (or equal) compact stellar object
      mass2              REAL NOT NULL,
-- Coalescence phase angle (radians)
      coalescence_phase  REAL,
-- Effective distance to the compact binary system (inferred from amplitude)
      eff_distance       REAL,
-- Signal to noise ratio
      snr                REAL,
-- Confidence variable
      confidence         REAL,

-- Unique identifier for this event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT s_inspiral_pk
      PRIMARY KEY (creator_db, event_id),

      CONSTRAINT s_inspiral_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT s_inspiral_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time
CREATE INDEX s_inspiral_ind_tim ON sngl_inspiral(ifo, end_time) CLUSTER
;
-- Create an SQL trigger so that if a sngl_inspiral entry is deleted, any
-- associated sngl_datasource and/or sngl_transdata entries are deleted too.
```

```
-- Must be done this way because there is no foreign-key relationship.
CREATE TRIGGER s_inspiral_tr_del
    AFTER DELETE ON sngl_inspiral
    REFERENCING OLD AS o
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
       DELETE FROM sngl_datasource
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
       DELETE FROM sngl_transdata
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
    END
;
```

7.3.2. `sngl_burst` Table Definition

```
CREATE TABLE sngl_burst
(
-- Event table for single-interferometer burst-event search.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS EVENT
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Interferometer
      ifo                CHAR(2) NOT NULL,

-- TIME OF THE EVENT
-- The start time of this burst event (in GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
-- The time duration of this burst event (seconds)
      duration           REAL NOT NULL,

-- PROPERTIES OF THE EVENT
-- Center of frequency band in which observation is made (Hz)
      central_freq       REAL,
-- Range of frequency observed (Hz)
      bandwidth          REAL,
-- Absolute signal amplitude (fractional strain)
      amplitude          REAL NOT NULL,
-- Signal to noise ratio
      snr                REAL,
-- Confidence variable
      confidence         REAL,

-- Unique identifier for this event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT s_burst_pk
      PRIMARY KEY (creator_db, event_id),

      CONSTRAINT s_burst_fk_pid
      FOREIGN KEY (creator_db, process_id)
```

```
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT s_burst_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time
CREATE INDEX s_burst_ind_time ON sngl_burst(ifo, start_time) CLUSTER
;
-- Create an SQL trigger so that if a sngl_burst entry is deleted, any
-- associated sngl_datasource and/or sngl_transdata entries are deleted too.
-- Must be done this way because there is no foreign-key relationship.
CREATE TRIGGER s_burst_tr_del
    AFTER DELETE ON sngl_burst
    REFERENCING OLD AS o
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
       DELETE FROM sngl_datasource
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
       DELETE FROM sngl_transdata
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
    END
;
```

7.3.3. `sngl_ringdown` Table Definition

```
CREATE TABLE sngl_ringdown
(
-- Event table for single-interferometer ringdown search.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS EVENT
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Interferometer
      ifo                CHAR(2) NOT NULL,

-- TIME OF THE EVENT
-- The start time of this ringdown event (in GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
-- The time duration of this ringdown event (seconds)
      duration           REAL NOT NULL,

-- PROPERTIES OF THE EVENT
-- Absolute signal amplitude (fractional strain)
      amplitude          REAL NOT NULL,
-- Fundamental ringdown frequency
      frequency          REAL NOT NULL,
```

```
-- Quality factor
      q                  REAL NOT NULL,
-- Black hole mass (in solar mass units)
      mass               REAL,
-- Signal to noise ratio
      snr                REAL,
-- Confidence variable
      confidence         REAL,

-- Unique identifier for this event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT s_ringdown_pk
      PRIMARY KEY (creator_db, event_id),

      CONSTRAINT s_ringdown_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT s_ringdown_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time
CREATE INDEX s_ringdown_ind_tim ON sngl_ringdown(ifo, start_time) CLUSTER
;
-- Create an SQL trigger so that if a sngl_ringdown entry is deleted, any
-- associated sngl_datasource and/or sngl_transdata entries are deleted too.
-- Must be done this way because there is no foreign-key relationship.
CREATE TRIGGER s_ringdown_tr_del
    AFTER DELETE ON sngl_ringdown
    REFERENCING OLD AS o
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
       DELETE FROM sngl_datasource
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
       DELETE FROM sngl_transdata
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
    END
;
```

7.3.4. `sngl_unmodeled` Table Definition

```
CREATE TABLE sngl_unmodeled
(
-- Event table for searches for “unmodeled” sources, i.e. sources for which
-- the waveform is unknown.  There will probably be several algorithms in
-- use, so this table includes the algorithm name.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS EVENT
```

```
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Descriptive name of the particular algorithm which flagged this event
      name               CHAR(32) NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Interferometer
      ifo                CHAR(2) NOT NULL,

-- TIME OF THE EVENT
-- The start time of this event (in GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
-- The time duration of this event (seconds)
      duration           REAL NOT NULL,

-- PROPERTIES OF THE EVENT
-- Absolute signal amplitude (fractional strain)
      amplitude          REAL NOT NULL,
-- Signal to noise ratio.
      snr                REAL,
-- Confidence variable
      confidence         REAL,
-- Note: additional properties may be recorded in the sngl_unmodeled_v table.

-- Unique identifier for this event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT s_unmod_pk
      PRIMARY KEY (creator_db, event_id),

      CONSTRAINT s_unmod_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT s_unmod_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time, etc.
CREATE INDEX s_unmod_ind_time ON sngl_unmodeled(name, ifo, start_time) CLUSTER
;
-- Create an SQL trigger so that if a sngl_unmodeled entry is deleted, any
-- associated sngl_datasource and/or sngl_transdata entries are deleted too.
-- Must be done this way because there is no foreign-key relationship.
CREATE TRIGGER s_unmod_tr_del
    AFTER DELETE ON sngl_unmodeled
    REFERENCING OLD AS o
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
       DELETE FROM sngl_datasource
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
       DELETE FROM sngl_transdata
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
    END
```

```
;
```

7.3.5. `sngl_unmodeled_v` Table Definition

```
CREATE TABLE sngl_unmodeled_v
(
-- Generic table to store additional values describing events found by a
-- search for “unmodeled” sources.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Unique identifier for the source-independent event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,
-- Site or interferometer to which this applies (H0, H1, H2, L0, L1)
      ifo                CHAR(2) NOT NULL,

-- Descriptive name of the result variable
      name               CHAR(32) NOT NULL,
-- The value of the result (must be a real number)
      value              REAL NOT NULL,

      CONSTRAINT s_unmodv_pk
      PRIMARY KEY (creator_db, event_id, name),

      CONSTRAINT s_unmodv_fk_unmod
      FOREIGN KEY (creator_db, event_id)
          REFERENCES sngl_unmodeled(creator_db, event_id)
          ON DELETE CASCADE
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on event id
CREATE INDEX s_unmodv_ind_eid ON sngl_unmodeled_v(event_id) CLUSTER;
```

7.3.6. `sngl_dperiodic` Table Definition

```
CREATE TABLE sngl_dperiodic
(
-- Event table for single-interferometer directed periodic-source search.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS EVENT
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Interferometer
      ifo                CHAR(2) NOT NULL,

-- TIME PERIOD FOR THIS “EVENT” (generally a long integration period)
-- Start and and time (GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
```

```
      start_time_ns      INTEGER NOT NULL,
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,
-- Duration (seconds)
      duration           REAL NOT NULL,

-- PROPERTIES OF THE EVENT
-- Name of target
      target_name        CHAR(32),
-- Sky position
      sky_ra             REAL NOT NULL,
      sky_dec            REAL NOT NULL,
-- Frequency
      frequency          REAL NOT NULL,
-- Absolute signal amplitude (fractional strain)
      amplitude          REAL NOT NULL,
-- Signal phase with respect to beginning of time interval
      phase              REAL NOT NULL,
-- Signal to noise ratio
      snr                REAL,
-- Confidence variable
      confidence         REAL,

-- Unique identifier for this event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT s_dperiod_pk
      PRIMARY KEY (creator_db, event_id),

      CONSTRAINT s_dperiod_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT s_dperiod_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time, etc.
CREATE INDEX s_dperiod_ind_time
    ON sngl_dperiodic(target_name, ifo, start_time, end_time) CLUSTER
;
-- Create an SQL trigger so that if a sngl_dperiodic entry is deleted, any
-- associated sngl_datasource and/or sngl_transdata entries are deleted too.
-- Must be done this way because there is no foreign-key relationship.
CREATE TRIGGER s_dperiod_tr_del
    AFTER DELETE ON sngl_dperiodic
    REFERENCING OLD AS o
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
       DELETE FROM sngl_datasource
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
       DELETE FROM sngl_transdata
           WHERE event_id = o.event_id AND creator_db = o.creator_db;
    END
;
```

7.4. Additional Information About Single-Interferometer Triggers/Events

Two additional tables are provided to store additional information about triggers/events, if desired. The `sngl_datasource` table can be used to indicate what input data (channel(s) and time interval) led to the trigger decision. In general it will specify a longer time interval than the duration of the candidate event itself, and may be useful to indicate what data is needed to verify or refine the trigger decision in off-line analysis. There can be at most one `sngl_datasource` entry per event (but it can list multiple channels).

The `sngl_transdata` table can be used to store the actual time series which exceeded the trigger threshold, e.g. the output of a digital filter or a pseudo-channel derived from multiple input channels. The size of the time series is limited to 1 megabyte. However, there can be multiple `sngl_transdata` entries per event, if desired. Heavy use of the `sngl_transdata` table could rapidly fill up the database, so it should be used only in cases where regenerating the time series would be computationally expensive.

7.4.1. `sngl_datasource` Table Definition

```
CREATE TABLE sngl_datasource
(
-- Pointer to specific data which prompted a single_interferometer trigger.
-- That is, this indicates what data the program was looking at when it
-- decided to generate the trigger.  In general, this datasource time
-- interval will be longer than the duration of the transient it contains.
-- Note that there can be only one sngl_datasource entry per trigger, but
-- it can list multiple channels.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Table with event to which this applies (gds_trigger, sngl_inspiral, etc.)
      event_table        CHAR(18) NOT NULL,
-- Trigger/event identifier
      event_id           CHAR(13) FOR BIT DATA NOT NULL,
-- Site or interferometer to which this applies (H0, H1, H2, L0, L1)
      ifo                CHAR(2) NOT NULL,

-- Source of this data.  Use 'DAQS' for original raw data, otherwise the
-- name of the frameset_group read.  If multiple frameset_groups were read,
-- list them all, separated by spaces.
      data_source        VARCHAR(240),
-- Channel(s) the data come from.  If more than one, list them all,
-- separated by spaces.
      channels           VARCHAR(512) NOT NULL,

-- The beginning of the time interval for the data (GPS seconds/nanoseconds)
      start_time         INTEGER NOT NULL,
      start_time_ns      INTEGER NOT NULL,
-- The end of the time interval for the data (GPS seconds/nanoseconds)
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,
```

```
      CONSTRAINT s_datasource_pk
      PRIMARY KEY (creator_db, event_id)

-- We cannot set up a foreign key based on event_id since the parent
-- table varies.  Could maybe set up a trigger to do the equivalent check.
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
```

7.4.2. `sngl_transdata` Table Definition

```
CREATE TABLE sngl_transdata
(
-- Record of transformed data upon which trigger decision was based.  There
-- may be multiple entries for a particular trigger instance, if desired.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- Table with event to which this applies (gds_trigger, sngl_inspiral, etc.)
      event_table        CHAR(18) NOT NULL,
-- Trigger/event identifier
      event_id           CHAR(13) FOR BIT DATA NOT NULL,
-- Site or interferometer to which this applies (H0, H1, H2, L0, L1)
      ifo                CHAR(2) NOT NULL,
-- One word or a few words to indicate what is stored in this table
      transdata_name     CHAR(16) NOT NULL,

-- X-axis parameters;
-- number of values in this tranformeddata
      num_bins           INTEGER NOT NULL,
-- starting and ending values of x-axis
      x_start            DOUBLE NOT NULL,
      x_end              DOUBLE NOT NULL,
-- Units of x axis (e.g. 'GPS seconds', 'Hz', etc.)
      x_units            CHAR(16) NOT NULL,
-- Y-axis parameters;
      data_type          CHAR(16) NOT NULL,
      y_units            CHAR(16) NOT NULL,
-- Transformed data itself is stored in a Binary Large OBject (BLOB).
-- We specify COMPACT since we do not expect this ever to be updated.
      transdata          BLOB(1M) COMPACT NOT NULL,
-- Length of data, in bytes
      transdata_length   INTEGER NOT NULL,

      CONSTRAINT s_transdata_pk
      PRIMARY KEY (creator_db, event_id, transdata_name)

-- We cannot set up a foreign key based on event_id since the parent
-- table varies.  Could maybe set up a trigger to do the equivalent check.
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
```

7.5. Coincidences of Single-Interferometer Events

After event candidates have been identified for individual interferometers, the next step will be to look for coincidences between interferometers and between events of different types. For example, we would like to check for an inspiral candidate found by more than one interferometer (with consistent signal amplitudes and mass combinations), possibly followed by a ringdown. (Note that the *end* time of the inspiral should be compared against the start time of the ringdown.) We would also like to note any instrumental/environmental trigger at nearly the same time, which might have caused a false gravity-wave signature. This correlation procedure will be done by a stand-alone program, and the results stored in the database. (SQL is not suited to doing a sophisticated correlation in any reasonable manner.) One will have to make choices about coincidence windows (probably wider for GDS triggers than for astrophysics events), the minimum number of event candidates to be considered a coincidence, how to handle multiple event candidates of a given type at the same time (e.g. different mass combinations for an inspiral search), etc. The philosophy should be to make a highly inclusive list of *possible* coincidences which will be correlated more precisely by off-line re-analysis of the data.

The following section presents a database table to point to the various individual event candidates which make up a coincidence. Only LIGO interferometers are included at present. Columns are defined to contain information about a coincident gamma-ray burst, if any; other external event types could be added as well. The composite time, "quality" of the coincidence, properties of the astrophysical system, and sky position information are included in the table, although it is not clear whether these are all evaluated during the coincidence-finding process, or only after post-processing. This table definition is likely to be modified as the coincidence analysis takes shape.

7.5.1. `coinc_sngl` Table Definition

```
CREATE TABLE coinc_sngl
(
-- List of approximate coincidences between single-interferometer events
-- from different interferometers and/or of different event types.
-- Currently, only includes LIGO interferometers plus information about
-- an associated gamma-ray burst, if any.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS ENTRY
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,

-- Unique identifier for this coincidence
      coinc_id           CHAR(13) FOR BIT DATA NOT NULL,

-- Single-interferometer events which make up this coincidence (NULL if no
-- match).  Note that we must specify the "creator_db" for each individual
-- event, since it may not be the same as the database on which this
-- coincidence record is being created.
      h1_inspiral_cdb    INTEGER,
      h1_inspiral_id     CHAR(13) FOR BIT DATA,
      h1_burst_cdb       INTEGER,
      h1_burst_id        CHAR(13) FOR BIT DATA,
      h1_ringdown_cdb    INTEGER,
```

```
      h1_ringdown_id     CHAR(13) FOR BIT DATA,
      h1_unmodeled_cdb   INTEGER,
      h1_unmodeled_id    CHAR(13) FOR BIT DATA,
      h1_gdstrig_cdb     INTEGER,
      h1_gdstrig_id      CHAR(13) FOR BIT DATA,

      h2_inspiral_cdb    INTEGER,
      h2_inspiral_id     CHAR(13) FOR BIT DATA,
      h2_burst_cdb       INTEGER,
      h2_burst_id        CHAR(13) FOR BIT DATA,
      h2_ringdown_cdb    INTEGER,
      h2_ringdown_id     CHAR(13) FOR BIT DATA,
      h2_unmodeled_cdb   INTEGER,
      h2_unmodeled_id    CHAR(13) FOR BIT DATA,
      h2_gdstrig_cdb     INTEGER,
      h2_gdstrig_id      CHAR(13) FOR BIT DATA,

      l1_inspiral_cdb    INTEGER,
      l1_inspiral_id     CHAR(13) FOR BIT DATA,
      l1_burst_cdb       INTEGER,
      l1_burst_id        CHAR(13) FOR BIT DATA,
      l1_ringdown_cdb    INTEGER,
      l1_ringdown_id     CHAR(13) FOR BIT DATA,
      l1_unmodeled_cdb   INTEGER,
      l1_unmodeled_id    CHAR(13) FOR BIT DATA,
      l1_gdstrig_cdb     INTEGER,
      l1_gdstrig_id      CHAR(13) FOR BIT DATA,

-- Interferometer coincidence time in GPS seconds/nanoseconds
      coinc_time         INTEGER NOT NULL,
      coinc_time_ns      INTEGER NOT NULL,

-- Variable describing the quality of the coincidence
      coinc_quality      REAL NOT NULL,

-- Direction of Hanford-to-Livingston ray at time of event
-- (i.e. the central axis of the cone on which the source lies)
      ligo_axis_ra       REAL,
      ligo_axis_dec      REAL,
-- Wave arrival angle with respect to Hanford-to-Livingston ray, and error
      ligo_angle         REAL,
      ligo_angle_sig     REAL,

-- Gamma-ray burst event, if any
      grb_id             VARCHAR(64),
      grb_time           INTEGER,
      grb_time_ns        INTEGER,
-- Location of gamma-ray burst in the sky, if applicable
      grb_sky_ra         REAL,
      grb_sky_dec        REAL,

-- Place to indicate any other non-LIGO event in coincidence, if any
      other_external     VARCHAR(80),

-- PHYSICAL PROPERTIES FOR EVENT
-- Masses of inspiral objects
      inspiral_mass1     REAL,
      inspiral_mass2     REAL,
-- Q value for ringdown
```

```
      ringdown_q         REAL,
-- Fundamental ringdown frequency
      ringdown_freq      REAL,
-- Black hole mass from ringdown
      ringdown_mass      REAL,

      CONSTRAINT coincs_pk
      PRIMARY KEY (creator_db, coinc_id),

      CONSTRAINT coincs_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Foreign-key constraints are checked only for non-NULL values
      CONSTRAINT coincs_fk_h1in
      FOREIGN KEY (h1_inspiral_cdb, h1_inspiral_id)
          REFERENCES sngl_inspiral(creator_db, event_id),
      CONSTRAINT coincs_fk_h1bu
      FOREIGN KEY (h1_burst_cdb, h1_burst_id)
          REFERENCES sngl_burst(creator_db, event_id),
      CONSTRAINT coincs_fk_h1ri
      FOREIGN KEY (h1_ringdown_cdb, h1_ringdown_id)
          REFERENCES sngl_ringdown(creator_db, event_id),
      CONSTRAINT coincs_fk_h1un
      FOREIGN KEY (h1_unmodeled_cdb, h1_unmodeled_id)
          REFERENCES sngl_unmodeled(creator_db, event_id),
      CONSTRAINT coincs_fk_h1gds
      FOREIGN KEY (h1_gdstrig_cdb, h1_gdstrig_id)
          REFERENCES gds_trigger(creator_db, event_id),

      CONSTRAINT coincs_fk_h2in
      FOREIGN KEY (h2_inspiral_cdb, h2_inspiral_id)
          REFERENCES sngl_inspiral(creator_db, event_id),
      CONSTRAINT coincs_fk_h2bu
      FOREIGN KEY (h2_burst_cdb, h2_burst_id)
          REFERENCES sngl_burst(creator_db, event_id),
      CONSTRAINT coincs_fk_h2ri
      FOREIGN KEY (h2_ringdown_cdb, h2_ringdown_id)
          REFERENCES sngl_ringdown(creator_db, event_id),
      CONSTRAINT coincs_fk_h2un
      FOREIGN KEY (h2_unmodeled_cdb, h2_unmodeled_id)
          REFERENCES sngl_unmodeled(creator_db, event_id),
      CONSTRAINT coincs_fk_h2gds
      FOREIGN KEY (h2_gdstrig_cdb, h2_gdstrig_id)
          REFERENCES gds_trigger(creator_db, event_id),

      CONSTRAINT coincs_fk_l1in
      FOREIGN KEY (l1_inspiral_cdb, l1_inspiral_id)
          REFERENCES sngl_inspiral(creator_db, event_id),
      CONSTRAINT coincs_fk_l1bu
      FOREIGN KEY (l1_burst_cdb, l1_burst_id)
          REFERENCES sngl_burst(creator_db, event_id),
      CONSTRAINT coincs_fk_l1ri
      FOREIGN KEY (l1_ringdown_cdb, l1_ringdown_id)
          REFERENCES sngl_ringdown(creator_db, event_id),
      CONSTRAINT coincs_fk_l1un
      FOREIGN KEY (l1_unmodeled_cdb, l1_unmodeled_id)
          REFERENCES sngl_unmodeled(creator_db, event_id),
      CONSTRAINT coincs_fk_l1gds
```

```
      FOREIGN KEY (l1_gdstrig_cdb, l1_gdstrig_id)
          REFERENCES gds_trigger(creator_db, event_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time
CREATE INDEX coincs_ind_time ON coinc_sngl(coinc_time) CLUSTER;
```

7.6. Multi-Interferometer Astrophysics Event Searches

Event tables for multi-interferometer searches are very similar to those for single-interferometer searches, but with a column containing a list of interferometers rather than a single interferometer, and with sky position information. Table definitions for inspiral and burst events are given below as examples; other event types can be added as needed. For instance, there will certainly be multi-interferometer stochastic-background searches, but it is not clear at present how (or whether) these should be represented in the database.

7.6.1. multi_inspiral Table Definition

```
CREATE TABLE multi_inspiral
(
-- Event table for multi-interferometer binary-inspiral search.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS EVENT
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Interferometers used for this search
      ifos               CHAR(12) NOT NULL,

-- TIME OF THE EVENT
-- The coalescence time of this inspiral event (GPS seconds and nanoseconds)
      end_time           INTEGER NOT NULL,
      end_time_ns        INTEGER NOT NULL,
-- The time duration of this inspiral event (seconds)
      duration           REAL NOT NULL,
-- Time when filter output reaches maximum value (GPS seconds and nanoseconds)
      fout_peak_time     INTEGER NOT NULL,
      fout_peak_time_ns  INTEGER NOT NULL,
-- duration of filter used to produce event (seconds)
      filter_duration    REAL NOT NULL,

-- PROPERTIES OF THE EVENT
-- Absolute signal amplitude (fractional strain)
      amplitude          REAL NOT NULL,
-- Mass of the larger compact stellar object (in solar mass units)
      mass1              REAL NOT NULL,
-- Mass of the smaller (or equal) compact stellar object
      mass2              REAL NOT NULL,
-- Coalescence phase angle (radians)
```

```
      coalescence_phase  REAL,
-- Effective distance to the compact binary system (inferred from amplitude)
      eff_distance       REAL,
-- Signal to noise ratio
      snr                REAL,
-- Confidence variable
      confidence         REAL,

-- Direction of Hanford-to-Livingston ray at time of event
-- (i.e. the central axis of the cone on which the source lies)
      ligo_axis_ra       REAL,
      ligo_axis_dec      REAL,
-- Wave arrival angle with respect to Hanford-to-Livingston ray, and error
      ligo_angle         REAL,
      ligo_angle_sig     REAL,

-- Unique identifier for this event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT m_inspiral_pk
      PRIMARY KEY (creator_db, event_id),

      CONSTRAINT m_inspiral_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT m_inspiral_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time
CREATE INDEX m_inspiral_ind_tim ON multi_inspiral(end_time) CLUSTER;
```

7.6.2. multi_burst Table Definition

```
CREATE TABLE multi_burst
(
-- Event table for multi-interferometer burst-event search.

-- Database which created this entry
      creator_db         INTEGER NOT NULL WITH DEFAULT 1,

-- INFORMATION ABOUT THE PROCESS WHICH GENERATED THIS EVENT
-- Process which generated this event
      process_id         CHAR(13) FOR BIT DATA NOT NULL,
-- Filter identifier (indicates type of filter, plus parameters).  May be null
      filter_id          CHAR(13) FOR BIT DATA,
-- Interferometers used for this search
      ifos               CHAR(12) NOT NULL,

-- TIME OF THE EVENT
-- The start time of this burst event (in GPS seconds and nanoseconds)
      start_time         INTEGER NOT NULL,
```

```
      start_time_ns      INTEGER NOT NULL,
-- The time duration of this burst event (seconds)
      duration           REAL NOT NULL,

-- PROPERTIES OF THE EVENT
-- Center of frequency band in which observation is made (Hz)
      central_freq       REAL,
-- Range of frequency observed (Hz)
      bandwidth          REAL,
-- Absolute signal amplitude (fractional strain)
      amplitude          REAL NOT NULL,
-- Signal to noise ratio
      snr                REAL,
-- Confidence variable
      confidence         REAL,

-- Direction of Hanford-to-Livingston ray at time of event
-- (i.e. the central axis of the cone on which the source lies)
      ligo_axis_ra       REAL,
      ligo_axis_dec      REAL,
-- Wave arrival angle with respect to Hanford-to-Livingston ray, and error
      ligo_angle         REAL,
      ligo_angle_sig     REAL,

-- Unique identifier for this event
      event_id           CHAR(13) FOR BIT DATA NOT NULL,

      CONSTRAINT m_burst_pk
      PRIMARY KEY (creator_db, event_id),

      CONSTRAINT m_burst_fk_pid
      FOREIGN KEY (creator_db, process_id)
          REFERENCES process(creator_db, process_id),

-- Note that filter_id is allowed to be null, in which case no check is made.
      CONSTRAINT m_burst_fk_filt
      FOREIGN KEY (creator_db, filter_id)
          REFERENCES filter(creator_db, filter_id)
)
-- The following line is needed for this table to be replicated to other sites
DATA CAPTURE CHANGES
;
-- Create a clustering index based on time
CREATE INDEX m_burst_ind_time ON multi_burst(start_time) CLUSTER;
```

8 SAMPLE QUERIES

Common queries (with some options) will be predefined as part of the user interface, so the user will not normally have to know the SQL query language. The sample queries below are provided as an “existence proof” that it is possible to retrieve useful information from the database.

- Get list of frameset groups:

```
SELECT DISTINCT frameset_group
FROM frameset_writer
ORDER BY frameset_group
```

```
FOR READ ONLY;
```

- Get list of framesets within a given time interval:

```
SELECT frameset_group, start_time, end_time, name
FROM frameset
WHERE (start_time >= 627332233 AND end_time <= 627418633)
ORDER BY start_time, frameset_group
FOR READ ONLY;
```

  Note that the example above only finds framesets which are wholly contained within the specified time interval. To include framesets which overlap the specified time interval, use:

```
WHERE (end_time > 627332233 OR start_time < 627418633)
```

- Get list of framesets within a given time interval which include the `H2:LSC-AS1_Q` channel:

```
SELECT fs.frameset_group, fs.start_time, fs.end_time, fs.name
FROM frameset fs JOIN frameset_chanlist ch
    ON (fs.creator_db=ch.creator_db AND fs.chanlist_id=ch.chanlist_id)
WHERE (fs.start_time >= 627332233 AND fs.end_time <= 627418633)
    AND (ch.chanlist LIKE 'H2:LSC-AS1_Q ')
ORDER BY fs.start_time, fs.frameset_group
FOR READ ONLY;
```

- Get list of all summary-statistics periods for which the rms of the `H0:PEM-LVEA_SEISX` channel is bigger than 20 counts:

```
SELECT start_time, end_time, mean, rms
FROM summ_statistics
WHERE (channel = 'H0:PEM-LVEA_SEISX' AND rms > 20.0)
ORDER BY start_time
FOR READ ONLY;
```

9 CONCLUSION

This document has presented a detailed design for the database tables underlying the LDAS metadata / event database. This design will be implemented soon on a test basis. At this point, suggestions for changes particularly welcome, especially from people planning to be involved in the astrophysics event searches. Experience with actual ingestion procedures and realistic queries may point out areas needing modification or optimization.

Open issues include the following:

- The interface with the data acquisition system is not currently understood. In particular, it is not clear who is responsible for filling the `frameset_chanlist` table. It would be highly desirable for the DAQS to do this, since each frame-builder process obviously already knows the channel list. A less attractive alternative is for an LDAS process to scan the frame files written by the DAQS and extract the channel list; presumably this process would only have to scan a frame whenever the run number changes, since the channel list must be constant throughout a given run (?). Also, it is not clear how detector configuration information (servo gains, etc.) is recorded in the database.
- The representation of a channel list as a CLOB limits one's ability to figure out what frameset group(s) should be retrieved to do a particular analysis, especially if we decide

on a data-archive model in which the data is split up by channel groups as it is ingested into the archive, which would allow quick retrieval of customized data streams. Therefore, we may find it necessary to represent the channel list differently (e.g. as a vector of sampling-rate codes for each of the few thousand possible channels) and add code to the metadataAPI to support sophisticated queries. There are some complications, having to do with enlarging the vector on-the-fly to accommodate new pseudo-channels, which would need to be worked out.

- The procedure by which a program assembles information and submits it to the database needs to be worked out in detail, and software written to facilitate it.
- The user interface needs to be developed. Some ideas are currently circulating for a web-based graphical user interface. A command-line user interface might also be useful. In particular, an analysis job might use a textual “data request descriptor” to specify data to be retrieved from the archive.
- The relationship among the multiple database installations needs to be defined. At present, the idea is to copy information from the interferometer sites to the central database on a daily basis and to perform essentially all operations at the central database, rather than setting up a truly distributed system, due to concerns about limited bandwidth and network reliability. DB2 provides tools to automate this sort of operation, although it is not yet clear whether they will do exactly what we want. Additional database installations can be supported as well, though administration issues need to be worked out.
- The LSC needs to establish guidelines for what information is to be included in the database and a procedure for considering proposed changes.