1. Introduction

In the 2003 paper, "Updated Guidelines for Atmospheric Trace Gas Data Management" [WMO, 2003], recommendations and guidelines for conceiving and utilizing a database management system (DBMS) were presented and developed, followed by a description of the DBMS developed for the National Oceanic and Atmospheric Administration (NOAA) Earth Systems Research Laboratory (ESRL) Global Monitoring Division (GMD) Carbon Cycle and Greenhouse Gas (CCGG) Group.

This paper will describe updates to the database design recommendations with references to how these design goals have been applied in the current model used by the CCGG Group today. The original introduction, reprinted below, is followed by a revised guideline reflecting the evolution of data management strategies in the intervening years.

Making the necessary measurements of atmospheric CO_2, its stable isotopes, and other trace gas species to better our understanding of the global carbon cycle consumes increasing amounts of time and resources. Changing scientific objectives impose demands for greater measurement precision and better temporal and spatial coverage. Advances in technology have made these demands attainable. Instruments have been designed to be more compact, robust, and efficient making it easier to produce reliable in situ measurements from ships, aircraft, towers, and from permanent and temporary remote locations. The quality of sample storage containers used in discrete sampling programs has improved, increasing the long-term stability of sampled air for a greater number of trace gas species. Semi-automatic analytical systems have been designed to use smaller volumes of air enabling multiple-species analyses from a single ambient sample. Finally, the affordability of computers and electronic storage media has greatly reduced a once formidable component of the operational budget. This partial list of advancements combined with an increasing scientific demand for more observations has resulted in the generation and accumulation of greater volumes of data. These data must be organized and maintained so that the effort required to make the measurements results in advances in our understanding of the scientific issues.

Measurement programs maintain records of sample collection details (e.g., collection location, date), raw analysis data (e.g., chromatograms, voltages), processed analysis data (e.g., mixing ratios, isotopic ratios), instrument diagnostic data (e.g., temperature, pressure), and standard gas calibration histories. In many instances, these data are maintained for each trace gas constituent measured. Further, individual laboratories may operate both continuous and discrete (flask and cylinder) measurement strategies from both fixed (land surface, towers, and ice cores) and moving (ship and aircraft) platforms. All of this information must be managed so that measurement data can be readily viewed, re-processed, selected, analyzed, and disseminated. A poor data management strategy can render even the very best measurements almost useless. Thus, it is imperative that measurement programs employ a well-designed data management strategy.

The carbon cycle measurement community, and in particular, the CO_2 community, has spent considerable effort developing instrument and technical guidelines for scientists entering into the atmospheric trace gas measurement field [WMO, 1998]. Guidelines which emphasize measurement techniques, calibration methods, and common pitfalls, provide fledgling measurement programs with a "recipe" for obtaining high-precision measurement results more quickly. A similar effort is required for

the management and maintenance of data produced by these programs. A first attempt to establish guidelines for atmospheric trace gas database management is presented. The data management strategy described is a compendium of concepts in use among many of the laboratories making trace gas measurements. The National Oceanic and Atmospheric Administration (NOAA) Climate Monitoring and Diagnostic Laboratory (CMDL) Carbon Cycle Greenhouse Gases (CCGG) Group has committed considerable resources towards the management of discrete and continuous data from the cooperative air sampling network, the CMDL baseline observatories, and the tall towers and light aircraft programs. Many of the examples presented are from CCGG programs that address many of the same data management considerations as other carbon cycle measurement laboratories. The majority of guidelines set forth here have been in use by CCGG for many years; however, the CCGG strategy has considerable room for improvement and continues to evolve as the measurement programs expand and new technologies become available. This discussion represents the direction in which the CMDL Carbon Cycle Greenhouse Gases Group is moving towards a robust database management system.
[Masarie et al., 2003]

In this paper, section 2 contains discussion of five basic requirements for designing a data management strategy along with commentary on how they can be achieved. Section 3 delves deeper into the design strategies needed to create an effective relational database, with practical examples of a 'normalized' database structure to minimize data inconsistencies and improve data integrity. Section 4 provides a brief overview of basic data query and manipulation techniques using Structured Query Language (SQL).

Sprinkled throughout are facts and examples from the CCGG Group's data management system, which currently houses almost 7 million trace gas measurements from discrete air samples and over 90 million measurements from quasi-continuous in situ analyzers.

2. Requirements

Few things are more important when designing a complex data management strategy than fully assessing and understanding the basic requirements of the project. The number of concurrent users, accountability needs, projected data growth and intended outputs are just a few of the questions that should be considered. The answers to these questions help inform important decisions on the type and structure of the data management system to be used.

Many organizations, when faced with quickly implementing a data management strategy, will choose a file based system. Comma Separated Value (CSV) text files or proprietary spreadsheets like Microsoft Excel are simple, portable and easy to use. Network file servers and cloud based file shares make it easy to share arbitrary files among users, allowing a basic data management system to be implemented quickly. However, the ease of implementation should be carefully weighed against the long term goals and requirements of the strategy. File based storage does not scale well when multiple users and processes need to access a large dataset. Concurrent modifications by different users can lead to

inconsistencies and corrupted data if not carefully managed and because the file system has no knowledge about the contents of the dataset, it is not able to help maintain the integrity of the data being stored.

In contrast, a Relational Database Management System (RDBMS) is expressly purposed to share large datasets with multiple users. Running on a server in your network (Figure 1), it abstracts away the details of how data is stored and presents a single logical interface for querying and manipulating records using Structured Query Language (SQL). A RDBMS provides a centralized data repository that manages the many requirements of allowing multiple users and computer programs to access and modify data in a controlled manner. By utilizing a 'relational' data model (described in section 3.2), a RDBMS allows for efficient storage that minimizes redundancy and maintains data consistency.

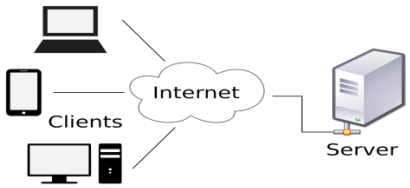

Figure 1 – Client/server architecture

When looking at the needs of a successful measurement program, it is clear that a RDBMS should be an essential part of the data management strategy. Fundamental requirements include data integrity, scalable performance, accessibility, recoverability and traceability. Each requirement is reviewed and followed with a brief discussion on how the requirement can be met as well as some commentary on how it is handled in the NOAA CCGG Group.

2.1 Data Integrity

Critical to the long term success of a measurement program is ensuring the accuracy and consistency of recorded measurement data. Any particular measurement has a host of related data that must be maintained. Sampling information like location, date and time along with analysis details about the instrument and its raw output must be recorded. Information about sampling equipment, ambient conditions and quality assurance checks need to be documented. These data points need to all be related together accurately so that, for example, measurements of different trace gases from the same sample can be easily reviewed. Multiple users, programs and processes will interact with, add to, and update attributes of the data at various stages of its life cycle. Maintaining data integrity throughout these stages is a difficult but vitally important requirement.

Data are more valuable when they are related to other data. As will be discussed in later sections, a RDBMS enforces and maintains relationships among data items by design. By managing how users are allowed to interact with and manipulate data, a RDBMS also provides an important buffer that can help enforce integrity rules. Stored data can be separated into different types, minimizing entry errors such as an invalid date. Trigger procedures can be automatically invoked after a modification to verify entries and log user edits. Multi-user access can be managed to prevent issues with concurrent editing and security rules can ensure only authorized users are allowed to modify data.

A thoughtfully designed database can help measurement programs protect one of their most valuable assets, but policy and procedures to manage how data are entered are equally important.

In the CCGG Group, the ability to enter, modify or delete data is restricted to a few designated users and funneled through a select group of computer programs. When possible, data are entered through automated processes like bar code scanners to check-in flask samples from the field and computer programs to process raw analysis instrument output and insert measurement results into the database. Manual data entry is minimized, but when necessary, it is carefully reviewed and validated by a second user to ensure accuracy. Periodic reviews of measurement results and comparisons to measurements from different sources and sampling strategies (e.g., flask vs in situ) are done to find discrepancies that need to be investigated. Automated consistency checks along with other error checking routines help monitor and look for potential issues. Database triggers are used on select tables for data validation and to record a history log of data modifications, providing accountability and traceability.

A soundly designed data repository, computer programs to control access and well thought out policy and procedures are essential components of a successful data management strategy.

2.2 Performance and scalability

Measurement programs may start with a limited staff and amount of recorded data, but can quickly grow as new sites, strategies and measured trace gases are added. Continuous observations alone can quickly add tens of millions of measurements. With more measurements, more staff are required for quality assurance, data analysis and operational logistics. As the value of the repository grows, increasing numbers of end users may request access to the data for comparisons and research. Planning for projected scalability is a key element of any data management strategy so that quick, but controlled, access is ensured even as the data and concurrent users grow.

There are many commercial and open source RDBMSs available (Oracle, Sybase, SQL Server, MySQL/MariaDB, PostgreSQL...) that, paired with sufficiently powerful server hardware, can scale to meet the needs of the largest programs. Instrumental to that ability however, is a thoughtfully architected data model that allows for efficient storage and access. This will be discussed further in later sections.

The CCGG Group operates various measurement programs including both discrete and continuous sample measurements from stationary and mobile sites. Discrete air sample collection in flasks has been ongoing since 1967 with quasi-continuous measurements starting in the early 1970s. The original text file based data system eventually became overwhelmed and too cumbersome to use so a DBMS and then a RDBMS was instituted in various stages through the 1990s and into the early 2000s. Currently hosting close to 100 million discrete and continuous measurements along with data from other GMD programs, the open source MariaDB database running on CentOS Linux uses approximately 400GB of storage space. It is able to handle queries from dozens of interactive users in addition to web queries.

2.3 Accessibility

Convenient, quick access to observational data is essential for internal users performing quality assurance, operational logistics and analysis. Users should be working from a central repository, not copies that quickly become out of date. These tasks, done using in-house and third party software, need to directly and indirectly interact with the data management system in a secure and controlled way. External users will also need access to exports of the data in a variety of formats, tailored to use and experience. A data management system needs to be accessible to all of these use cases by providing a consistent, flexible interface that can be readily incorporated into new and existing analysis programs while restricting access for editing purposes to authorized personnel.

RDBMSs are generally accessed through the use of Structured Query Language (SQL), a language designed by IBM Corporation [Halpin et al., 2001] for accessing and manipulating database information. SQL queries are very flexible, allowing datasets to be quickly combined and filtered in numerous ways. Queries are submitted to the database through a client program that must first connect to and then authenticate with the RDBMS server. Most programming environments (Python, PHP, .NET, Perl…) used to create web applications, graphical user interfaces (GUI) and command line programs include, or have available, libraries for connecting, authenticating and querying to the various commercial and open source RDBMS products available today. This allows authorized computer programs to become 'clients' to the RDBMS server and use a consistent interface (SQL) to access and modify its data. The general availability of SQL access in commercial software and client programming environments means that in-house computer program developers have great flexibility in how to program applications for different classes of users.

The CCGG Group has developed many tools over the years to analyze and work with our measurement data including command line programs written in Perl, graphical and plotting programs written in Python, IDL, R and MATLAB and web applications written in PHP. These programs support automated and interactive processes to manage operational logistics, quality assurance, data analysis and exports (FTP, ObsPacks…) to the community and wider public.

The ability to view, plot and analyze data is widely available to internal users. Custom computer programs have been developed to enable users to quickly access and review any of the trace gas measurements done by our different measurement programs. This allows results to be quickly disseminated and incorporated into current modelling and other research projects. It also plays a vital role in quality control. For example, in the CCGG aircraft sampling program, elevated measurements of certain trace gases like H1211, a halon compound used in fire extinguishers but not naturally occurring, are used to flag suspected sampling system leaks of cabin air. This gas is measured by the Halocarbons & other Atmospheric Trace Species (HATS) Group, but is integrated into the central database and available for all users. Being able to quickly and accurately link results from multiple trace gases allows problems to be detected promptly and potential issues to be tracked down.

Custom software developed in-house can be difficult to program and requires dedicated staff, but has the advantage that functionality is tailor made for the intended use and can be readily adapted as

needs change. It also allows a measurement program to standardize methods and calculations among different internal groups which helps to preserve the long term consistency of the data.

2.4 Recoverability

A sound and well thought-out backup strategy is an essential part of any data management system. There are many points of failure for any computerized system. A backup strategy must address everything from how to correct hard drive corruption to recovering from a catastrophic building fire. The main decision an organization must make is what the level of recoverability should be or, put another way, what is acceptable data loss?

For instance, a backup strategy of taking a weekly snapshot implies that the loss of a week's worth of data is acceptable in the case of a catastrophic failure. On the other end of the spectrum, High Availability (HA) systems can be implemented for instant failover to replicated servers for near zero downtime and data loss. The costs and complexity to implement the backup strategy increase proportionally as the acceptable data loss is reduced. At a minimum, database backups should be run on a scheduled basis with offsite storage of periodic snapshots to be able to restore operational data.

The CCGG RDBMS is backed up daily to a network storage device, with 31 days of history. The storage device itself is copied to tapes with six months of recoverability and periodic off site storage, allowing disaster recovery of operational data to various points in time with minimal data loss. In addition, select processed data sets are archived annually at the World Data Centre for Greenhouse Gases (WDCGG) and at the National Centers for Environmental Information (NCEI).

2.5 Traceability

Traceability, as it applies to the trace gas measurement community, requires that every measurement must be traceable to primary standards through an unbroken chain of direct comparisons to successively higher order standards. By maintaining traceability, measurements done at different times, locations or even programs can be compared. It also allows changes to the primary scale or to the assigned value of standard tanks in the scale hierarchy to be propagated down to applicable measurements.

Changes to the assigned values of standards used by a measurement program should be anticipated and planned for so that affected raw instrument output can be reprocessed using the most current values. Instrument output data, whether stored in text files or a database, must be preserved and in a form that allows processing logic to recalculate mole fractions using the updated standards information. This should be as automated as possible so that changes are quickly and accurately propagated.

In the CCGG Group, raw instrument output for discrete measurements is stored in text files whose format has changed many times over the years as new instruments are added and processes change. Computer programs written to process files have versioning logic that allows any of the historical formats

to be read. This ensures that historical raw instrument response files can be reprocessed at any future date. All files store standard tank identification numbers so that reprocessing logic can retrieve current assigned values from our centralized calibration database. While not fully automated due to the nature and complexity of calibration changes, the procedures and methodologies are well documented so that reprocessed mole fractions can quickly be updated in the database and reflected in all analysis and plotting computer programs.

3 Designing a database

Creating a relational database is a complex project that requires planning, development and ongoing maintenance. Measurement programs with limited staff may want to consider engaging consultants or custom software developers to assist in designing the database as well as computer programs to interact with it.

For those with the staff and resources to pursue their own development, the following sections provide some insights and guidelines for designing a database with a goal of optimizing storage, speed of access and data integrity. Practical examples of a hypothetical measurement program are used to illustrate how to apply these goals to a relevant data set.

These guidelines loosely describe the database currently in place in the CCGG RDBMS which has evolved over many decades to meet changing requirements. The experiences and pitfalls encountered by CCGG developers in designing and adapting the data management system to available technology and the particular nuances of a trace gas measurement program are reflected in the recommendations below.

3.1 Data flow

A first step to designing a database system is to analyze the flow of data to determine what the fundamental data entities or data objects are and how they relate to each other. For example, in a CO_2 measurement program based on discrete air samples, the flow of data may start at the moment of collection. Information about the time and location must be recorded as well as equipment details such as the flask ID and collection method. Recorded meteorology and any anomalies that may affect the quality of the measurement need to be related to the raw data. After the sample is sent back to the measurement lab, operational logistic systems will need to know when it's checked-in and where it resides. In the NOAA GMD program, the flask will then begin a journey through one or more of 5 different labs for analysis of over 60 trace gas species. Raw instrument output during analysis will be processed and converted into mole fractions and isotopic ratios. Analysis details like species measured, instrument used and analysis time are recorded with the processed data. Sampling and analysis tables are related to each other via the primary key (discussed in section 3.2.2). Quality assurance will likely occur in several steps, first by automated processes, and then by one or more people using plotting and statistical programs to identify analytical problems and statistical outliers that should be designated as not likely to be representative of specified conditions. During this process, measurement results of several related species, possibly from different labs, will need to be available for review. Once the measurement details have been finalized, both internal and external users will analyze and use the results for comparisons, in models, and for basic research.

Other collection strategies will have slightly different data flow and therefore different design requirements. For example in a continuous in situ measurement system, the collection of air isn't a discrete event whose attributes are shared with many trace gas measurements. The collection and measurement are done at the same time, which creates a similar but significantly different set of recordable attributes. Aircraft campaigns and other mobile measurement systems may generate both continuous and discrete measurements with the added complication of needing to record profile and changing location information.

Only after the flow of data is thoroughly analyzed and the relationships between data entities identified can the process of architecting the database structure begin.

3.2 Database Tables

Data in an RDBMS are stored in a series of data tables that can be related or 'joined' together to create full data sets. Actual disk storage details differ, but tables can logically be thought of as a matrix of rows and columns, similar to a spreadsheet or a CSV file. There are important differences however.

A CSV file can be opened and edited by many programs and users, allowing great flexibility but little control over how data are modified. A column in the file can be labeled as a date, but there is no guarantee that all values actually are dates. And while the rows may all represent different records, there is no mechanism in place to enforce that they are uniquely selectable.

A database table, in contrast, is not directly accessible. Viewing and manipulating data in a table is strictly managed by the RDBMS, which follows rules laid out by the database designer. Each column, or field, is named and rigidly classified (typed) by the kind of data it will contain (date, time, integer, string…). Invalid values are disallowed by the RDBMS. Rows in the table are comprised of records identified by a 'primary key', discussed below, which ensures they are all uniquely selectable. By managing access to tables, a RDBMS is able to control the contents to ensure consistency.

There is also a significant conceptual difference in how relational database tables are defined. CSV files are generally stand alone data sets, often combining many different domains of data for users to manipulate and analyze interactively. They are structured for ease of use and portability. They can be used in complicated statistical and plotting programs like JMP and IDL or in simple text readers like VI or NOTEPAD. Relational database tables however, are rarely interacted with individually. Their primary purpose is to serve as a logical structure for storage and efficient access. As part of an architected system, a relational database table doesn't need to be independently portable, or even human readable. They are structured and organized for storage efficiency, access efficiency and data integrity.

In general, relational database tables are architected to represent a single logical entity or data object. The columns in the table describe the fundamental attributes of that individual data object. Additional tables can hold information that relates to the data object in some way. Individual records in the related tables can be linked together to construct a complete set. How those data objects and

subsequent tables are defined and related are very important for both performance and reducing data inconsistencies. There is much literature and theoretical analysis of relational database design that is outside the scope of this paper, but a general understanding of the concept of database normalization is very helpful to understanding how to create a sound database architecture.

3.2.1 Database normalization

First described by E.F. Codd in the early 1970s and later refined in subsequent papers and books by Codd and others, database normalization is a method of organizing data to reduce redundancy and potential inconsistencies by structuring data tables in a logical and efficient manner.

For an example, assume a person was tasked with recording the site code, date, flask ID and collection method for each of a program's discrete samples. They could create a simple table with all the columns (Figure 2).

Site_code	Datetime	Flask_ID	Method
MLO	2015-12-29 19:46:00	208-99	P
MLO	2015-12-22 19:41:00	2516-99	P
BRW	2016-01-04 19:12:00	83-99	S
BRW	2016-01-11 19:05:00	4007-99	P
...			

Figure 2 - Sample events

What if in addition to the three letter site code, they also had to record the full site name? That could easily be added as another column (Figure 3).

Site_code	Site_name	Datetime	Flask_ID	Method
MLO	**Mauna Loa, Hawaii**	2015-12-29 19:46:00	208-99	P
MLO	**Mauna Loa, Hawaii**	2015-12-22 19:41:00	2516-99	P
BRW	**Barrow, Alaska**	2016-01-04 19:12:00	83-99	S
BRW	**Barrow, Alaska**	2016-01-11 19:05:00	4007-99	P
...				

Figure 3 - Events table with both site_code and site_name columns

If a sample location was entered incorrectly, both the site_code and site_name columns would need to be edited together and kept in sync. A small possibility for an inconsistency is introduced because the site_code and site_name may not get updated together, leaving the possibility that a row could have a site_code with incorrect name. There is also a small amount of redundancy as the combination of site code and site name are repeated but never change.

Now consider if in addition to site_name, the person was asked to record the site's offset from utc, country, state/province and a short description that would change periodically with staffing changes.

Site_code	Site_name	utc_offset	Country	State_prov	Description	Datetime	Flask_ID	Method

MLO	**Mauna Loa, Hawaii**	**-10**	**USA**	**HI**	**Mauna Loa Observatory is located on the Island of Hawaii at an elevation of 3397 m ...**	2015-12-29 19:46:00	208-99	P
MLO	**Mauna Loa, Hawaii**	**-10**	**USA**	**HI**	**Mauna Loa Observatory is located on the Island of Hawaii at an elevation of 3397 m ...**	2015-12-22 19:41:00	2516-99	P
BRW	**Barrow, Alaska**	**-9**	**USA**	**AK**	**Barrow Observatory, established in 1973, is located near sea level 8 km east of Barrow, Alaska...**	2016-01-04 19:12:00	83-99	S
BRW	**Barrow, Alaska**	**-9**	**USA**	**AK**	**Barrow Observatory, established in 1973, is located near sea level 8 km east of Barrow, Alaska ...**	2016-01-11 19:05:00	4007-99	P
...								

Figure 4 - An un-normalized events table with extra site attributes data bolded

If these columns are added to the same table (Figure 4), several problems become apparent. A much larger chance for inconsistency is introduced as there would be several columns (site_code, site_name, utc_offset, state_prov and country) to update and keep synchronized if the site has to be changed. Changes to the site description would need to be replicated to every applicable row. Storage requirements would be significantly larger due to the repeated information.

Instead, the person would likely avoid the extra work and repetition by creating a second table that just contained the site information, one row for each site. When someone wanted to know the utc offset or the current site description for an event, they could just look up the site code in this new master site list and retrieve the data. Updates to the site description would only need to occur once to apply to all applicable rows.

This, essentially, is the concept of normalization. Logical data objects are separated into separate data structures that can be linked together to recreate a full data set. The problems of redundancy and data inconsistencies are solved through design.

While a person may intuitively understand the relationship between a sampling event's site code and how to look up further details in a master site list, a relational database needs this relationship explicitly defined thru the use of primary and foreign keys, described below.

3.2.2 Primary keys and unique constraints

To continue the example as applied to database table architecture, the logical data objects are separated into separate tables, one for sampling events (Figure 5) and the other for site information

(Figure 6). Each row in the tables represents a single record and must be unique among all the other records in the table.

Site_code	Datetime	Flask_ID	Method
MLO	2015-12-29 19:46:00	208-99	P
MLO	2015-12-22 19:41:00	2516-99	P
BRW	2016-01-04 19:12:00	83-99	S
BRW	2016-01-11 19:05:00	4007-99	P
...			

Figure 5 - Events table

Site_code	Site_name	utc_offset	Country	State_prov	Description
MLO	Mauna Loa, Hawaii	-10	USA	HI	Mauna Loa Observatory is located on the Island of Hawaii at an elevation of 3397 m ...
BRW	Barrow, Alaska	-9	USA	AK	Barrow Observatory, established in 1973, is located near sea level 8 km east of Barrow, Alaska...
...					

Figure 6 - Sites table

The set of attributes (columns) that uniquely identifies a record is referred to as the 'primary key' and is designated at the time the table is defined. The database enforces a unique constraint on the primary key, such that it disallows insertions or modifications of records that would create a duplicate primary key. Because of its uniqueness, it can be used to reference specific records in the table or to join related records when querying the database.

For the site table, the site_code is an obvious primary key as it will be unique among all site records. In the events table, each record can be uniquely defined by the combinations of columns date and flask_id since our data flow analysis dictates that there can not be 2 samples in the same flask from the same date and time. Note that it doesn't actually matter if there technically could be duplicates, the unique constraint being defined here is to enforce business rules (operational rules) about what is allowed to be stored into the database. There can be records with either the same date or with the same flask_id - it is the designated combination of the two fields that creates a unique record. This is known as a compound key or multi-column primary key.

Because the primary key is often used to identify a record, either in direct queries or when referenced in related tables, many architects will designate a 'surrogate' primary key for convenience, performance and other practical reasons. For tables with multi-column primary keys, a surrogate key makes queries clearer and more concise. The surrogate key is generally an integer, but can be any supported type. Most RDBMSs can implement an 'auto-incrementing' function for integer keys so that new rows are automatically assigned the next value without having to be specified in the insert command. Be sure to verify that the range of the chosen data type is large enough to hold the expected number of rows. For instance, a MySQL SMALLINT can only hold 65,535 values while an unsigned INTEGER in the same database has a range of 4,294,967,295 values.

Event_num	Date	flask_ID	Site	Method
123	**2015-12-29 19:46:00**	**21-69**	MLO	P

Figure 7 - Events table with surrogate key event_num and unique 'primary' key date flask_id

In this configuration, the event_num field is designated primary key and a separate unique constraint is configured for Datetime and Flask_ID so that the business logic of a unique date and flask is still enforced. The record can be unambiguously identified by either the event_num or the combination of date and flask_id equally. The ability to enforce business rules and data integrity in the design and implementation of the data model allows the database designer to ensure that stored data are always in a consistent state no matter which program or user entered it.

Similarly, many database architects will designate a surrogate key for tables with character based primary keys for several practical reasons. An integer key is generally more efficient to store than a character key with similar range. The extra storage requirements can be significant on large tables. With an integer key, there is no question of whether case is significant and there is no ambiguity between similar characters like capital I (i) versus lower case l (L) or with different character sets. If a need arises to change the user facing text (change a three letter site code or use a different character set), an integer surrogate key and all its references won't need to be updated. Lastly, disassociating the data content (site_code, method...) from the actual mechanism used to store and reference it gives future developers flexibility when having to adapt the data model to new requirements.

That said, using a text based primary key like site_code can save some complexity in queries (section 4) and is easier to read, so the decision on whether to use a surrogate key depends on the preference of the database architect.

Site_num	Site_code	Site_name	utc_offset	Country	State_prov	Description
75	MLO	Mauna Loa, Hawaii	-10	USA	HI	Mauna Loa Observatory is located on the Island of Hawaii at an elevation of 3397 m ...
15	BRW	Barrow, Alaska	-9	USA	AK	Barrow Observatory, established in 1973, is located near sea level 8 km east of Barrow, Alaska...
...						

Figure 8 - Sites table with surrogate primary key site_num and unique constraint on site_code

3.2.3 Foreign keys and related tables

The 'non-primary' columns in the table record secondary attributes of this specific record. These include the site and collection method for each sample in our events example. Each of these non-primary columns needs to be analyzed to see where they fit in a relational model. As a rule of thumb, any datum not specific to an individual record should be in a related table. Or put another way, if a piece of data is an attribute of something other than the primary key, it should be in a related table.

We have already decided that the site's extended data should be in a related table, but what about the method? For this question, a database architect needs to consider future potential requirements to decide if there will ever be a time when someone wants to record something more about the collection method than a single letter. It seems likely that a description might be useful to record. Since, like the

site description, a method description is an attribute of the method not a particular event, these data should be in a related table.

Method_num	Method_code	Method_description
8	P	Sample collected using a portable, battery powered pumping unit.

Figure 9 - Methods table with a surrogate primary key method_num and unique constraint on method_code

To link the sites and methods tables to the events table, the related tables primary keys can be used as a 'foreign key' in the events table, which provides an explicit link between the tables and a means to join an event with it's extended site data (utc offset, site description…) and method data (method_code, description) (Figure 10). Most RDBMSs provide foreign key constraints to enforce relational integrity between referenced tables. For example to ensure that a foreign key (site_num) always points to an actual row in the database and wasn't mis-entered, a constraint, a rule that is imposed in the design of the database, would prevent adding an event row without referencing an existing site_code and conversely prevent deleting a site row if any linked event records exist.

Site_num	Site_code	Site_name	utc_offset	Country	State_prov	Description
75	MLO	Mauna Loa, Hawaii	-10	USA	HI	Mauna Loa...

Method_num	Method_code	Method_description
8	P	Sample collected using a portable, battery powered pumping unit.

Event_num	Date	flask_ID	Site_num	Method_num
123	**2015-12-29 19:46:00**	**21-69**	75	8

Figure 10 - Sites and Methods tables linked to Events table thru foreign keys

3.2.4 Different relation types

To continue the example, once a sample is returned to the lab and analyzed, one or more trace gas measurement results will need to be recorded for each sample event. This requires a similar but conceptually different relation than the sites and methods relations used above. Those tables are referred to as 'lookup' or 'dictionary' tables, meaning that they contain a list of available choices for a particular field, e.g. - each sample event can only be attached to one of the defined sites and use only one of the defined collection methods.

To record the one or more measurement results that belong to a given sample event, we instead create a 'master-detail' or 'parent-child' relation. Functionally, they can be structured in the same way as

a lookup table, using a foreign key to join records, but in this case the child rows are considered to 'belong' to the parent row. A measurements table might look like this:

event_num	species	value	unc	flag	inst	meas_date
123	co_2	401.650	-999.99	...	L8	2016-01-14 16:13:00
123	ch_4	1848.660	1.1000	...	H6	2016-01-14 15:21:00
...						

Figure 11 - Measurements table with event_num foreign key

In this arrangement, the events table is referred to as the 'parent' table and the measurements table is referred to as the 'child' table because one or more measurements belong to each sample. The measurements event_num column is the 'foreign key' that links this measurement to its sample information. Queries retrieving details of this event and measurements can join the tables using the foreign key (as described in section 4) to view or work with full details of the event and measurements.

The distinction between parent-child and lookup table relations, while logically very similar, is mentioned because some RDBMS implementations allow a designer to treat them differently, which may have implications on how data modifications are handled. For instance, in some implementations deleting a member of a lookup table will be prevented if it is referred to by another table, but deleting a parent row can be configured to cause all of it's dependent child rows to be deleted (cascading delete). The chosen RDBMS documentation should be consulted so that any implementation details are thoroughly understood.

3.2.5 Completing the example

To complete the normalization of this example, each of the 'non-primary' columns in the measurements table need to be analyzed to see where they fit in a relational model. Species and instrument are candidates for relational tables because there will likely be attributes of each that we wish to record (Figure 12 and Figure 13).

species_num	abbr	name	unit
1	co_2	Carbon Dioxide	umol mol-1
2	ch_4	Methane	nmol mol-1
...			

Figure 12 - Species table

inst_num	name	manufacturer	model
17	L8	Licor	7000
10	H6	HP	6890
...			

Figure 13 - Instrument table

And finally, we can use these related tables in the measurements table (Figure 14).

data_num	event_num	species_num	value	unc	flag	inst_num	meas_date
456	123	1	401.650	-999.99	...	17	2016-01-14 16:13:00
457	123	2	1848.660	1.1000	...	8	2016-01-14 15:21:00
...							

Figure 14 - Normalized measurements table with foreign keys to the events, species and instruments tables

Conversely, flag, uncertainty and value are attributes of this unique measurement and therefore are recorded directly. The decision on how far to take this process, i.e. how many relation tables to create, is not always clear. Convenience in construction of queries is a factor as is unknown future requirements and the anticipated importance of the data attribute. For example, a site's country could be recorded as a text string or in a lookup table. Systems that will need to use the country for shipping information would likely want to record it in a lookup table to prevent entry errors. Systems recording it for informational display purposes may decide a text string is more flexible and can better handle sites without an obvious country like Antarctica or ship based measurements.

3.2.5 Indexing

Indexes are internal structures that provide metadata about the content and storage of a table to the database so that it can quickly find requested information. Proper indexing of data tables is very important to how efficient and fast a database can respond to a query. While the details of how they are implemented and used by a RDBMS are beyond the scope of this paper, the general concepts will be extremely helpful to understand when designing and optimizing a database.

Consider as an example, a paper phone book that contains an alphabetized listing of phone customers in a city. For each person, the address and phone number are listed. To find the number for John Mund, a user would open the book at the middle, thumb back or forward to the Ms, then to MUs and finally to the page with the listing. In just a few steps, any name in the book can be looked up and the address and phone number retrieved.

Now consider if instead of finding a number for a name, a user wants to know which person belongs to a particular number. The physical layout of the book is no longer helpful and the user would have no choice but to scan through each page until the number is found.

These two scenarios represent the difference between what's called an indexed search and an un-indexed search, the latter taking many orders of magnitude more work to resolve. The way a database would optimize the second case is to create an ordered list of every phone number in the book along with the page it occurs on. A user could quickly scan through the ordered list to the desired phone number and use the accompanying page number to find the right page in the phone book. They would still need to scan the page to find the number, but the total number of steps is now similar to doing a lookup by name.

In our discrete measurement example above, it might be common for users to retrieve measurements from a specific site and date range. An index created on the events site_num and date columns would likely dramatically speed up this type of query. A database designer should consult the RDBMS manual for information on how to use built in tools to analyze query performance to see where indexes might be appropriate. Because indexes impose overhead and additional storage requirements, they should only be thoughtfully added after careful analysis.

3.3 Triggers and constraints

Database triggers and constraints are implemented differently depending on the specific RDBMS being used, but have a similar logical purpose of allowing the database designer to ensure that all data modifications (inserts, updates and deletes) are done in a controlled manner that enforces data integrity and consistency. For example, foreign key constraints, mentioned in previous sections, prevent occurrences of 'orphaned' child rows that have invalid references to the parent table. Other constraints can enforce a minimum set of required fields for a row. Triggers that are automatically run on "insert" and "update" operations can automate the creation of ancillary data or be used to create a data modification log, recording before and after snapshots of all user edits. Because this logic can be embedded at such a low level in the data structure, meaning that it is run no matter which user or program is modifying data, it is a powerful tool for controlling how the data are managed.

3.4 Views

A normalized relational table layout is very clean, efficient and robust, but is not always very intuitive or accessible for users accessing the data without a detailed knowledge of the design. It's generally advisable to funnel user access through client programs that can provide a friendlier interface and abstract away the details of the RDBMS. For users or developers that want to access the database directly through SQL queries, most RDBMSs provide a construct called a view, which provides the ability to hide some of the implementation details and present a consistent interface to relevant data. A view is a dynamic query that can, for example, join all the tables in the previous flask measurement example and provide an output similar to a flat file. Users can query the view as if it were the real table and the RDBMS does the work of combining tables behind the scenes to provide the results.

Generally created by the database manager, they can be used as a user convenience, performing common joins and output formatting in the background, or as an abstraction layer allowing changes to the underlying data structure without having to modify the way clients access the data.

4 Data query and manipulation

There are many courses, books and papers describing in detail how to access and manipulate a database using Structured Query Language (SQL). For the purposes of this paper, we will only give a short overview with some practical examples.

4.1 Access restrictions

While advanced users may have the ability and need to directly query the database using SQL to retrieve data, data modifications (inserts, updates and deletes) should be restricted to database administrators and developers as the commands can be very powerful and potentially misused. This is easily accomplished in most RDBMS implementations through the use of user permissions. General users can be granted 'select' (read) permissions and disallowed from issuing modification commands. Administrators, developers and specific computer programs can be granted additional permissions as appropriate, so that the ability to modify data is controlled and restricted. Access permissions to vital data should play an important part in any organization's overall security strategy. See your vendor documentation for details.

4.2 Transactions

Most RDBMS software provides support for transactional control of data modification commands. Transactions allow the caller, whoever is modifying the data, to wrap one or more SQL statements into a single logical group such that either all succeed and are saved to the database or if one fails for some sort of error condition, all fail and none are saved to the database. This atomicity enforces a consistent state among the group of statements and allows the caller to gracefully fail in an error condition without having to worry about whether some operations succeeded or not.

Transactional control is very difficult to add to an existing database and so if desired, should be implemented at a very early stage of the design process. The CCGG database, having grown and evolved organically over many years, does not make use of transactions. Instead, a set of automated consistency checks are run on the database to look for potential issues and notify database administrators.

4.3 Inserting

Automated scripts processing raw analysis data as well as programs allowing users to enter data will need to insert into various tables. There are several variations and exact syntax differs among database implementations, but in general an insert statement specifies the table and columns that will be inserted and then the values to use for those columns.

```
insert events(event_num,date,flask_id,site_num,method_num)
        values (123,'2015-12-29 19:46:00',26,75,8)
```

Other variations allow inserting multiple rows at once or inserting data selected from another table.

4.4 Updating

Corrections, completions and general edits to the data are inevitable. The update command is similar to the insert command in that it specifies the target table and columns to modify, but the structure is a little different.

```
update events set date='2015-12-29 19:45:00' where event_num=123
```

Here the table is specified and then one or more columns along with their new value. The 'where' clause limits the update to the row with primary key of 123.

A critical function of a measurement program is being able to quickly and accurately propagate changes in calibration scales and assigned values for standards to the measurement data table. The update statement can be used to update particular data rows through reprocessing of raw data or to apply a correction en mass to whole sets of rows by using a different 'where' filter that selects rows based on species and date, for instance. These types of statements can be dangerous however as it is easy to inadvertently update unintended rows. The RDBMS documentation should be carefully consulted to ensure a thorough understanding of the command's abilities.

4.5 Selecting

Retrieving data from a database is done through a select command. There are many variations, some implementation specific, but the general command is to select a list of columns from one or more joined tables meeting a set of conditions. For instance, a simple query to select all sample dates for site_num 75 looks like this:

```
select date from events where site_num=75
```

To display and filter by the site's three letter code requires joining to the sites table. The 'join' clause combines related results from two tables based on the condition in the 'on' clause.

```
select sites.site_code, events.date
from events join sites on events.site_num=sites.site_num
where sites.site_code='MLO'
```

Here, the events table is joined to corresponding sites rows using the 'on' clause to specify which columns relate them. During the query processing, a virtual table is created by combining each event row with it's corresponding sites row using the site_num key in each. The 'where' clause is used to filter the results to the target rows (site.site_code is MLO). Only the site_code and event date are returned. We specify the full table names to distinguish between ambiguous column names (site_num is a column in both tables).

Both ‘on’ and ‘where’ use boolean logic to combine multiple clauses. To limit the query to just MLO events using collection method_num 8, we can use the ‘and’ operator.

```
select sites.code, events.date
from events join sites on events.site_num=sites.site_num
where sites.code=’MLO’ and events.method_num=8
```

An ‘or’ operator can be included using parentheses to designate the order of operations.

```
select sites.code, events.date
from events join sites on events.site_num=sites.site_num
where sites.code=’MLO’ and (method_num=8 or method_num=9)
```

This limits the result rows to MLO events that are either method_num 8 or 9.

Multiple tables can be joined to select any subset of available columns. Aliases or abbreviations for column output headers and query tables (created using ‘as’ command) can be used to keep the code clear when many tables are joined and need to be referred to. Here we select the site name, date, species and measurement value for all samples taken in 2015.

```
select s.name,
        e.date as 'collection_date',
        sp.abbr as 'species',
        m.value
from events as e
        join measurements as m on e.event_num=m.event_num
        join sites as s on e.site_num=s.site_num
        join species as sp on m.species_num=sp.species_num
where e.date>='2015-01-01'
        and e.date<'2016-01-01'
```

Most RDBMS implementations have datetime data type which allows for the full date and time to be stored in a single field for use in comparisons and arbitrarily complex date arithmetic. For example to find the samples taken at MLO in 2015 between 0900 and 1400 local time we could use something like this (MySQL specific) syntax:

```
select s.code,
        e.date
from events e join sites s on e.site_num=s.site_num
where s.code=’MLO’
        and e.date>='2015-01-01'
        and e.date<'2016-01-01'
        and time(date_add(e.date, interval s.utc_offset hour))
                between '09:00:00' and'14:00:00'
```

The ‘group by’ operator allows for aggregate functions like count, average, min & max to be quickly and easily calculated. Adding month and the ‘count(*)’ keyword to the selected columns with a ‘group by’ directive specifying how to aggregate gives monthly counts by site of the above.

```
select s.code,
        month(e.date) as month_num,
        count(*)
from events e join sites s on e.site_num=s.site_num
where s.code=’MLO’
        and e.date>='2015-01-01' and e.date<'2016-01-01'
        and time(date_add(e.date, interval s.utc_offset hour))
                between '09:00:00' and'14:00:00'
group by s.code,
        month(e.date)
```

These examples are by no means an exhaustive instruction in how to query a relational database using SQL. They are meant to show a range of possibilities of what type of queries are possible. Interested users should consult the RDBMS documentation for specific details of available syntax and capabilities.

5. Conclusion

A measurement program’s data are one of its most important assets. A comprehensive, well planned data management strategy is essential to a program’s long term success. A strategy must help ensure the integrity of the data while providing quick access. A well designed relational database management system can serve as the core data repository, abstracting away the storage details to provide a managed interface for programs and users to access data. Policies, procedures and user programs must also be developed to ensure that data entry is validated and done in a controlled manner. Measurements must be traceable to the scale and appropriate working standards so that calibration changes can be applied quickly and accurately. Finally, all aspects of the system from architectural design decisions to detailed process instructions should be well documented to ensure that data remains useable and accountable over the long term.

The data management system in use by the CCGG Group has evolved over several decades to meet needs of a growing program. We continue to adapt and redesign it to meet changing requirements.

Acknowledgments

We thank P. Lang, E. Dlugokencky, A. Crotwell, M. Crotwell, K. Thoning. A. Andrews, B. Hall and M. Meyerson for their helpful discussions while preparing this paper.

References

Baxendale, P., & Codd, E. (1970). A relational model of data for large shared data banks. Communications of the Acm, 13(6).

Codd, E. (1991). The relational model for database management : Version 2 (Reprinted with corrections. ed.). Reading, Mass.: Addison-Wesley.

Halpin, T. (2001). Information modeling and relational databases : From conceptual analysis to logical design (Morgan kaufmann series in data management systems). San Francisco: Morgan Kaufman.

Hernandez, M. (1997). Database design for mere mortals : A hands-on guide to relational database design. Reading, Mass.: Addison-Wesley Developers Press.

Sumathi, S., & Esakkirajan, S. (2010). Fundamentals of relational database management systems (Studies in computational intelligence, v. 47). Berlin: Springer.

World Meteorological Organization (WMO) (2003). Updated Guidelines for Atmospheric Trace Gas Data Management, WMO/TD- No. 1149; GAW Report- No. 150, Geneva: Global Atmos. Watch.