# Simple Data Storage and Manipulation For Scientists

Charles Noneman Leonard McMillan

**Abstract**

When choosing a data management system, scientists are forced to select from using either spreadsheets or a relational database. However, neither of these systems is both flexible and powerful enough to fulfill all of a scientist's needs. We have developed a system that combines the simplicity of table creation and modification of a classical spreadsheet with the querying power of a full relational database. This system allows users to simply and efficiently input, query, and transform their data.

## 1 Introduction

Scientists are faced with a difficult trade-off regarding data management. This trade-off is between the two most common approaches to data management: spreadsheets and databases. Spreadsheets are very flexible, but lack the querying capabilities of a database. A database has powerful querying mechanisms and consistency guarantees, but is bound by a rigid schema.

A spreadsheet is a group of cells organized on a grid. Each cell can contain data; typically in the form of strings, numbers, dates, and formulas. Cells are addressed by their column number, in practice represent by a letter, and their row number. Scientists, for the most part, are familiar with spreadsheets and are comfortable using them to store data and to do basic analysis.

Typically no data management system is in place at the start of a project, so members of a group will each create ad hoc spreadsheets to store the data for their portion of the experiment. This leads to many difficulties and complications as the project continues and as the team tries to analyse their data. One risk of using spreadsheets is the possibility of data loss or corruption. If one of the files is accidentally deleted or incorrect data is written, there is no way to recover the information. Another problem is access to the spreadsheets. Teams often email sheets between members, or put the sheets on shared drives. The time and coordination required for this form of communication, combined with the lack of atomic transactions, means that data is only shared at the end of the experiment and that the integrity of the data is likely compromised. A better system would allow researchers to query and reorganise data as it is generated so preliminary analysis can begin immediately and inconsistencies and missing data can be found and resolved before they are impossible to correct. Another problem with the many-spreadsheets approach is that someone must merge and factor the sheets by hand. This is a time consuming and error prone process. In addition to errors generated by lost columns or copying mistakes, the sheets themselves contain implicit information, such as batch numbers or the dates of experiments, which is destroyed in the combining process. Even if all of these issues are addressed by a data management process and discipline on the part of the users, such as the process described in [6], the lack of proper joining and querying mechanisms in spreadsheet systems means that the data will need to be imported into a database eventually.

A relational database is collection of relations, commonly called tables. A relation has a fixed set of attributes and domain descriptions called a schema. A relation also has a set of tuples which are used to store data and all share the same attributes. Attributes are commonly called columns and tuples are commonly called rows. When designing a relational database, one strives to represent the data in a normal form. Normal form for relational data is a group of schemas where no data is replicated. This is im-

portant for efficiency and crucial for maintaining data consistency. Most relational databases systems are accessed via a query language, of which Structured Query Language, or SQL, is the most common.

Like the spreadsheet approach to data storage, an approach based on a relational database system also does not fit the needs of scientists. Databases are more difficult to use than spreadsheets. Users lack the training in database theory to generate schemas in normal form and to write queries in SQL. This results in a need to hire someone to handle the database's design. This person will first have to design the database schemas and then provide an interface for the researchers to use. This is the normal approach to database design and is called *schema-first*[7], since the schema is completely or mostly designed before any data is entered. Schema-first design assumes that the nature of the data is known a priori, but this is not often the case for real world projects. In the event that the researchers decide to run a new experiment, or simply want to add a new measurement to an existing experiment, the database schema no longer meets their needs. Users will need to repurpose an existing attribute to fit their new needs or they will be forced to contact the database administrators to perform the appropriate change in the schema. Additionally, if the researchers want to look at the data in a new way, they must have the administrators write a new query before they can begin their new analysis. In practice there will be enough of these changes to the nature of the project that database administrators must be retained for the entirety of the project. Also, these changes take time, so users may revert to spreadsheets for everyday use, bringing all of the problems associated with spreadsheets into the system. Actively maintaining the schema in this way is both inefficient and expensive.

Since the schema often cannot be determined before data collection has started and updating the schema is challenging and fails to keep up with users' needs, it may be tempting to construct a database after the data is collected. This approach is called *schema-later*[3]. By using spreadsheets during data collection, researchers have the flexibility they want, and by then loading this data into a database, they will get the querying power of a relational database. The failing of this approach is the enormous task of converting the data into a form suitable for a database.

There are many issues with data stored in a spreadsheet and many of them are difficult to resolve. Spreadsheet users will often put data of incorrect type into columns. For example, "NA" into a column of real numbers. Dealing with these situations must cause either an over general typing strategy, such as making everything strings, or result in lost data. Users will include decorative information into a sheet such as a title or empty rows and columns. These must be removed before an import can occur. Users will include statistics in the bottom rows of a table, such as sums and averages, which must be removed. Since users lack a joining mechanism, they will create separate columns for a repeated measurement, such as "Weight 4/8/2010" and "Weight 4/22/2010", instead of putting these into a new table. This type of data must be transformed by hand or by writing code to parse the sheet, convert the data, and save the resulting sheets. Many of the design issues in spreadsheets come from one-to-many and many-to-many relationships, such as the repeated measurement issue. These relationships are difficult to create in a spreadsheet and, in the many-to-many case, are difficult for users to understand since a table must represent a relationship and not just an entity. The process of converting from spreadsheet to database can easily become more work than actively maintaining the schema of a database.

There are some partial solutions to these problems. Online spreadsheets facilitate sharing of information between users and provide access to old versions of the spreadsheet, but combining and querying sheets remains a challenge. Visual databases ease the creation of queries[1], but still require an understanding databases and do little to aid in database design. In particular a user who has no database training is unlikely to store data in a normal form, even with visual tools. Additionally, schema rigidity is not solved by visual tools. Database usability is an active area of research[3].

In this paper, we provide a description of a system, called *S3*, that is as simple to use as a spreadsheet, but also has the full power of a relational database.

This system allows each researcher to produce tables and run queries that are easy to create, use, and change. By giving researchers the power manage their own data, they can easily keep the schema and queries up-to-date and applicable to their current needs. Importantly, the system is always usable as a relational database with a changing schema, including the full expressive power of SQL. This form of technique is called *schema-during*[7].

The system is accessed using a web interface, Figures 1 and 2. Having the database online ensures that users are always able to access their and other's information. Additionally, data can be entered directly into the database, which avoids a typical–time consuming and error prone–approach of writing data by hand, entering it into a spreadsheet, merging the spreadsheet with other spreadsheets, transforming the data, and loading the final sheet into a database.

Despite the flexibility in S3, users are encouraged and helped to convert their data into a more database appropriate form. For example, if the type of a column is set to an integer, any cells that are not integers are highlighted in red. By having an efficient and easy to use querying mechanism, users are more open to storing data in separate tables, unlike their tendency to want to force data into one large summary sheet. By including statistics for columns and allowing aggregate functions in queries, users will not need to clutter the sheet by filling cells with that type of summary information.

The system is designed with a cell-centric approach, which allows for much of the flexibility seen in spreadsheets. This approach also enables the system to maintain cell history, which allows for users to recover historical information and find when it was entered. Since all of the data is stored in a relational database, the full power of the existing joining and querying system is always available.

S3 provides a "gentle slope" approach to database design. It enables the transitions from data collection, to organization, to reorganization and to the transparent creation of a full-fledged relational database.

## 2 Related Work

Researchers have been developing user-friendly querying mechanisms for nearly as long as there have been databases. Query by Example [12] was one of the first. It involved users filling in example values into columns to serve as placeholders for the values in the tuples that the database would return. Visual Query Engines, of which [1] is an example, show a digram of the schema of the relations and draw lines between the relations to represent foreign keys. These systems still require the user to create the query in SQL, or may support simple queries using a point-and-click interface. Liu et al. [4] describe an iterative querying mechanism which is displayed like a spreadsheet.

Combinations of spreadsheets and databases have been proposed before. Tyszkiewicz [9] describes a method for translating SQL statements into equations in a spreadsheet. Query by Excel [11] translates spreadsheets into a database. The data from cells is stored similarly to the way that S3 stores cells and the functions are translated using an extended SQL syntax described in [10].

While previous work has focused on well designed databases in normal form, S3 is designed to address data input, querying, *and* manipulation on schema that may be poorly designed. Although S3 includes an easy-to-use querying interface and combines the concepts of spreadsheets and databases, these are only pieces of the whole. It provides a dynamic, yet fully functional, schema during the entirety of the data collection process. Joins, by default, are done using a method that allows scientists to easily access all relevant entities without having to consider the order of the joins. This means that data is simple to query and that it is easy to find where data is missing or incomplete. Finally, database refactoring is simple and is often done transparently for the user.

## 3 Model

First we will describe S3 from the user's perspective. This will demonstrate the capabilities of the system and further clarify the problems being ad-

Figure 1: The hub page. Here, users can access and manage tables, enumerations, and reports.

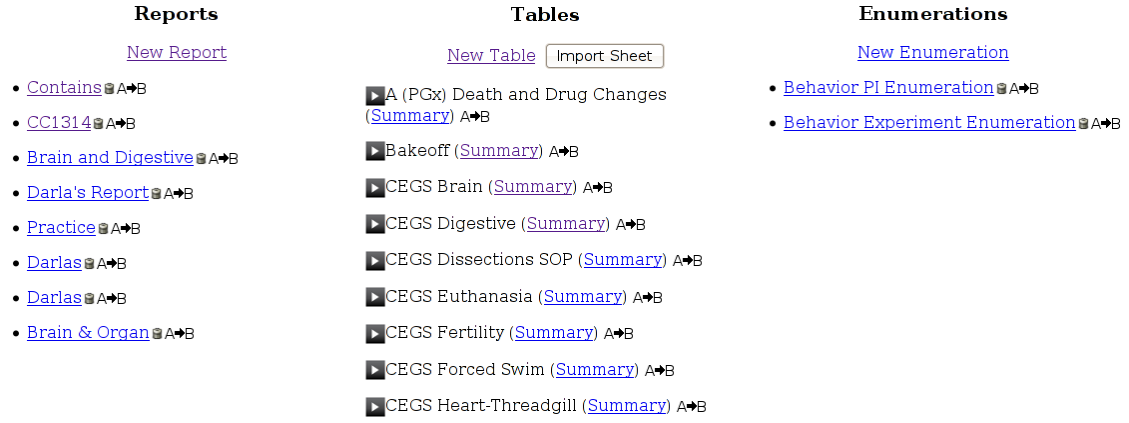
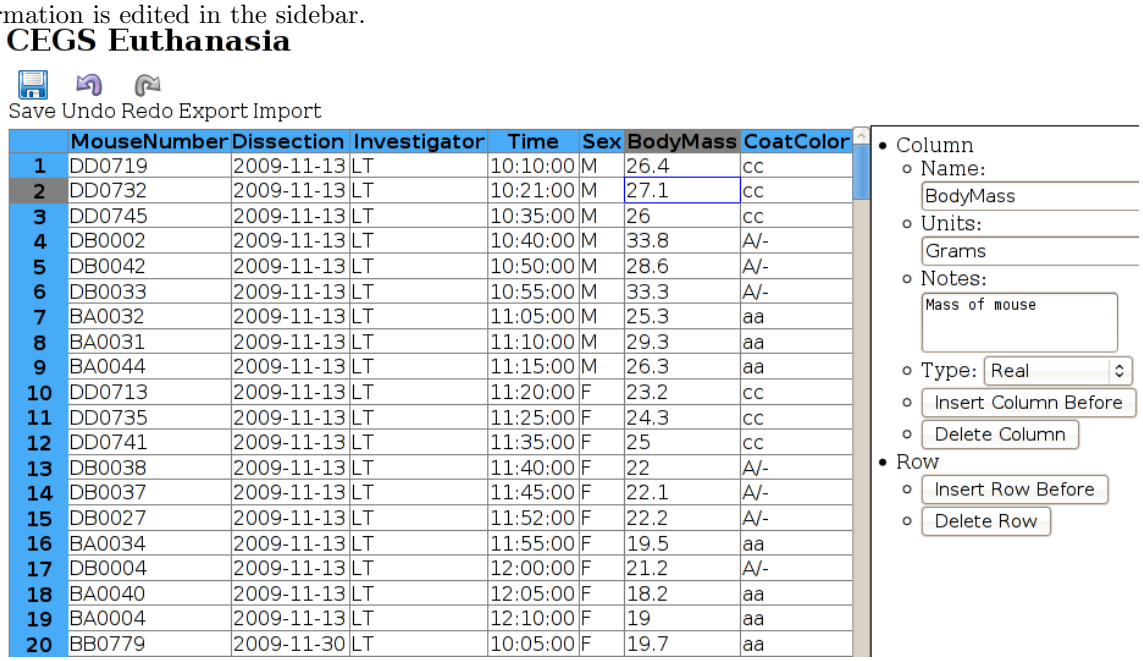



Figure 2: A typical table containing actual data. Users edit data as they would in a spreadsheet. Attribute information is edited in the sidebar.



dressed. The tasks performed by users are: adding data, querying that data, and manipulating the data to be closer to normal form.

### 3.1 Adding Data

A relational database requires the creation of a *schema* before data can be entered into a table. A schema is a list of the names of all of the columns and the data-type–number, text, date, etc.–allowed in each column. Additionally column names must be unique within the table. Although these requirements are good practice, in S3 this information is not required to create a table. Users may simply put data into the table and specify column name and type information later. The system is designed to handle duplicate or unnamed columns. Columns are assumed to store text by default, but this can be changed at any time. In a relational database, a column's type can only be changed if all of the data can immediately be converted to the new type. S3 always allows the change to happen and highlights problematic fields in red. Additionally, extra information can be attached to a column. This includes units and a full text description of the column's contents and intended use. A typical table is shown in Figure 2.

In addition to standard types, S3 supports enumeration and file uploads cells. An enumeration is a list of valid values for a column. For example, red, orange, and yellow could be listed as valid colors. Enumerations are beneficial since they enforce a standard nomenclature. Files can be placed in cells just like data, this allows for images, video, or any file type to be easily associated with applicable data. Users can then search for a sample and then simply click on the file-cell to open the file.

Sometimes files are not merely stored in cells, but are a source of derived fields. Data needs to be extracted from these, often machine-generated, files. These files are handled by uploading them into cells like normal files. A user with knowledge of the format writes a *filter* to process one of these files and associates the filter with the column. S3 will run the filter on all existing files, new files as they are uploaded, and new versions of files. Each file will result in data added to the current row or rows of data added to another table, as specified by the filter.

Since data often exists outside of S3, especially when converting from an existing project, importing directly from spreadsheets is supported by S3. Simple data can be copied and pasted into tables. Existing workbooks can be uploaded and split into any number of tables. As with typing into cells, these methods allow data of incorrect type to be entered into columns and highlights them in red, so no data is lost in the importing process.

### 3.2 Querying Data

The query interface is designed with simplicity in mind. Users first select which tables to include in the query. Then, for each table, users select which column contains data that is equivalent across the tables. A suggestion is given for a compatible column for each table based on heuristics, such as column name, matching data-types, and joins in other queries. Users can then limit the selected columns to a subset of the tables' columns. Additionally users may add restrictions based on the data. For example limiting the data to mice born after a certain date. These queries can be saved as *reports*. Tables and reports can be exported in Common Separated Value, CSV, format, so they can be trivially imported into other applications.

While the report generation interface is designed to handle common report cases. If more advanced queries are required, users can write arbitrary SQL queries based on the tables. All tables and reports result in actual views in the DBMS.

### 3.3 Manipulating Data

Most experiments involve a repeating measure of the same entity under different conditions. For example, a mouse's weight will be measured repeatedly at different points in its life. In a spreadsheet, a scientist will often create a column for each measurement. These columns might be: "Weight 10/4/10", "Weight 10/11/10", "Weight 10/18/10" or simply: "Weight 1", "Weight 2", "Weight 3". This is a bad data-model, for a few reasons. Most crucially, important information is stored in the column name,

where it cannot be accessed in a query. Additionally, the schema has to change whenever a new measurement is added. In the relational data storage model, the schema would look like: "Mouse ID", "Date", "Weight" and there would be a row for every time any mouse was measured. The many-columns method is commonly used in spreadsheets since users have never seen the correct way to store this data and even if they had, the lack of a sophisticated query mechanism would make data in normal form difficult to access. Additionally, users find a view with one line per mouse much easier to read, understand, and perform input on than one line per mouse-measurement.

To solve this problem, a combination of heuristics and user selection is used to identify these repeated measurement columns. Once the system identifies them, users have the option of viewing this data in three ways. The first is in the repeated columns that they entered. The second is a summary view that compresses the columns into a single column using an aggregate function such as mean, median, or sum. The final view has the data converted into the many-rows form of a database. This is achieved by taking every cell in the repeated columns and converting it into a row. This row contains a cell with the data part of the original column name, for example "10/4/10", a cell for the value in the original cell value, and finally an exact copy of the data in the non-repeated columns. Giving users access to this data in multiple forms means that they can interact with their data in the way that is easiest for the task at hand.

# 4 System Design

S3 is implemented in a relational DBMS, which gives S3 and its users access to the querying power of SQL. Despite storing information in a database, the data-model is more closely related to a spreadsheet. By representing the data in manner similar to that of a spreadsheet, S3 is able to avoid the limitations of a database. S3 makes cells the focus of the data-model, instead of tuples like in a database. This allows for the data in cells to manipulated in ways that are not possible with a tuple-centric model.

## 4.1 Schema

The basis of S3 is a representation of a (virtual) table. By representing a table instead of creating an actual database table, the system gains significant flexibility. This flexibility is apparent when considering the most important of the actual tables, the `Cells` table, Table 1.

Table 1: `Cells` Table

| Cells |
|---|
| value |
| attribute_id |
| agglomeration_id |
| created |
| replaced |

Each value in a virtual table has an entry in the `Cells` table. Each `Cell` has a reference to the `Agglomerations` table and the `Attributes` table. The `Agglomerations` table represents the virtual rows. The `Attributes` table represents the virtual columns and stores the name of the column, the data-type that the system will use in views, and optionally units or any user notes about the column, Table 2. These are used to place the cell within a virtual table.

An advantage of this approach to storing data is that `NULL` values take no space. They are stored implicitly by not having a `Cell` with a matching `agglomeration_id` and `attribute_id`. In a traditional database, `NULL`s are stored by setting a specific bit, or possibly even a byte, in the row header.

Table 2: `Attributes` and `Agglomerations` Tables

| Attributes |
|---|
| id |
| name |
| type_id |
| units |
| notes |

| Agglomerations |
|---|
| id |

By storing the user's input as a string in the `value` field, the user's original input, and thus intention, can always be recovered. This method allows for the typing of a column to change without any risk of permanent information loss. Additionally it allows users to input data that does not match the column type, and then correct this mismatch at a later time.

The enumeration and file types are handled slightly differently. For the file type, when the user uploads a file, it is stored and a record is added to the `Files` table which contains a unique id and some information about the file. The id number is stored in the `value` field of the appropriate cell. For the enumeration type, the source `attribute_id` is stored in the referring `attribute`'s record. The `Cell`s themselves have the `agglomeration_id` of the wanted cell stored in `value`, like a foreign key.

Unlike a traditional schema, entries in the `Cells` table are not updated when a user changes a value. A new entry is placed in the table, with `created` set to the current time, and the `replaced` field of the old `Cell` is changed from `NULL` to the current time. This maintains the history of the virtual table and allows for users to see the changes to the table over time.

A few auxiliary tables exist to manage the virtual tables, columns, and rows. The `Tables` table contains the name and creation date of a table. The `Table_Columns` table is a listing of all of the `Attribute`s in a given `Table` and `Table_Rows` is a list of all of the rows for any `Table`.

This data-model creates flexibility in not only the typing of the `Cell`s, but in the location of the `Attribute`s and `Agglomeration`s. `Attribute`s and `Agglomeration`s can not only be moved within a table, but can be trivially moved to new tables. This provides a mechanism for schema refactoring, which is needed keep the schema applicable to current needs and to move the schema towards normal form.

## 4.2 Views and Querying

Both the virtual tables and the reports are manifest as views in the DBMS. The view for a `Table` is created by first creating a view for each `Attribute`. This is created by taking the `Cells` table and selecting only those cells where `replaced` is `NULL`–to get the current versions of the `cells`–and the `cells` with the requisite `attribute_id`. Only the `agglomeration_id` and the `value`, cast as the appropriate type, are projected. For the `attribute` with `id` equal to $n$, this can be stated in relational algebra as Equation 1.

$$\rho A_n(\pi_{\text{agglomeration\_id, value}}( \sigma_{\text{replaced=NULL, attribute\_id=}n}(\text{Cells}))) \quad (1)$$

The `Table` view is then created by taking the `Table_Rows` table, selecting the correct `table_id`, $t$, and performing left joins against against each of the `attribute` views on `agglomeration_id`. This is given by the expression in Equation 2.

$$\pi_{\text{agglomeration\_id}}(\sigma_{\text{table\_id}=t}(\text{Table\_Rows})) \bowtie_{n \in N} A_n \quad (2)$$

To preform a query, users select which tables they want to join and an equivalent column for each of these tables. Optionally they can specify additional conditions. Users choose if they want the intersection of the values in the equivalent columns or the union. For an intersection an inner join is done between the views of the selected `Table`s.

For a union, an SQL statement is constructed that first takes the union of the equivalent columns and then does a left join with each of the selected tables. With $C$ as the set of views of compatible attributes and $T$ as the set of tables, this yields Equation 3.

$$\left( \bigcup_{A \in C} \pi_{\text{value}}(A) \right) \bowtie_{V \in T} V \quad (3)$$

This produces the report that users expect. In particular every identifier is included no matter which `Table` it was in and each identifier appears only once. If instead of using the union method a chain of left joins was used, identifiers that did not appear in the first `Table` in the join would not appear in the result. If a full outer join is used, identifiers that are the same, but are in `Table`s that are not directly joined, will produce two rows if the identifier does not exist in any one of the intermediary tables.

Queries can be saved using a table called `Reports`, which stores the name of the Report. `Report_Wheres`, `Report_Columns`, and `Report_Tables` store the selection criteria, chosen columns, and joined tables respectively. `Report_Tables` additionally stores if all of the tables column's were chosen, as is the default, or if specific columns were chosen. If all of the columns were selected and a column is added to the table at a later date, the report will be updated to include the new column.

### 4.3 Repeated Measurement Columns

S3 stores information about repeated-measurement columns in a table called `Attribute_Groups`. `Attributes` in an `Attribute_Group` have a foreign key to that group. `Attribute_Group` has a `name`, used for the name of the data column, and an `instance_name`, used for the name of the column that specifies to which measurement the tuple refers.

When generating the view of a table, the `Attribute_Groups` must be transformed from the many column form into the form with a measurement number column and a data column. To achieve this, each `Attribute` in the `Attribute_Group` is converted into a view with column for the `agglomeration_id`, a column containing the `Attribute`'s name, and a column with the values of the `Cells`. The union of these tables is generated and this union can be joined with other `Attributes`, to create the final `Table` view. If $G$ is set of `attribute_ids` specified in the `attribute_group` and $A_i$ is the relation constructed in Equation 1, this yields Equation 4.

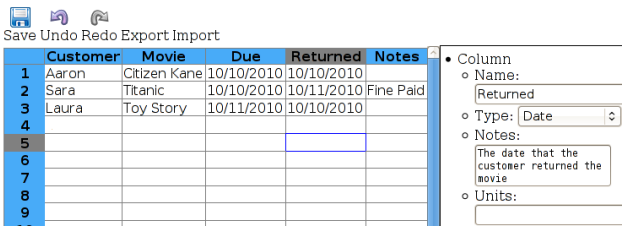
$$\bigcup_{i \in G} \left(A_i \times \pi_{\text{name}}(\sigma_{\text{id}=i}(\text{Attributes}))\right) \qquad (4)$$

## 5 Motivating Example

To illustrate S3's facilities for managing data in a non-normal form, transitioning data into a more normal form, and handling a changing schema, we present an example of data and tasks that benefit from S3.

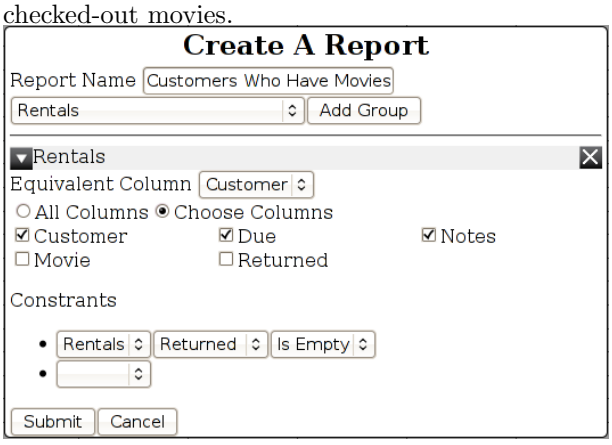
Suppose a mom-and-pop movie rental store decides to start using S3 to manage its checkouts. The owner creates the Rentals Table, Figure 3. "Customer" is a string with the customer's name. "Movie" is a string that holds the name of the movie. "Due" and "Returned" are dates. "Notes" stores any extra information.

Figure 3: Rentals table, as designed by a typical user.




He creates a report by choosing the "Rentals" table and filtering on "Returned" being empty, Figure 4. This produces a report of customers with checked-out movies.

Figure 4: Designing a query to list all users with checked-out movies.




One day a customer tries to rent more than one movie at a time. Two new columns, "Movie 2" and "Movie 3", are added to allow for up to three movies to be rented at once. If a customer returns some, but not all, of her rented movies, this is recorded in the "Notes" field. Only when all movies are returned is

"Returned" updated. The result of this is that the customers with checked-out movies report continues to work as expected.

Later, the owner decides that employees are using different names for for movies and miss-typing movie names, making search difficult. To solve this, he creates a table called "Titles" that simply lists all of the movie titles. He changes the type of the "Movie" columns in "Rentals" to an enumeration based on the "Title" column in "Titles". Internally, the strings that used to be in the cells are converted to the `agglomeration_id` of the corresponding row in "Titles". The result is that "Rentals" has a foreign key to "Titles", but this is transparent to the users. Any of the fields that are not listed in "Titles", due to typos or omission, are highlighted and can be corrected at any time.

The owner wants to list every time "Titanic" has been checked out. This is challenging in the current set-up since movies are listed across three columns. The owner uses the repeated measurement system to create a column group of the "Movie" columns. Now when he goes to make a report, "Movie 1", "Movie 2", and "Movie 3" have been replaced by "Movie" on the list of columns. Now he can easily filter on any movie.

When the owner started his database, he did not anticipate that he would need to do more than manage a few customers and movies. Since his needs seemed so basic, he made the table that was the simplest to design and use. The querying mechanism allowed him to quickly query the data without writing any equations or SQL. The enumeration and repeated measurement mechanisms begin normalisation of the schema, without the owner having to manually move any data.

Future features will expand the data manipulation abilities of S3. One of the most important for the movie-rental store owner is the ability to define a function, that is based on a query, that will generate values in a column. This will allow him to add a column to the "Rentals" table for customer identification numbers which will automatically fill out the customer name field by looking it up in another table.

# 6 Future Work

S3 is already being used by a group of biologists to manage their data, but there are several new features that are planned for the future.

First, enumerations will be expanded to allow for *synonyms*. Synonyms are alternate terms that can be used to set an enumeration. For example, "alb" could be set as a synonym for "albino". Cells containing either term would be set to the same key. These would primarily be used to expedite the process of converting a string column into an enumeration column, but could also be used to allow users to input legacy terminology or abbreviations. These would likely be implemented using the repeated measurement subsystem. Synonyms would appear as entries in columns, "Synonym 1", "Synonym 2", etc., and these would be transformed into a table suitable for mapping. Synonyms would be another feature help users evolve their data towards a normal form.

One of the biggest problems with SQL is that users must write an entire query without seeing any intermediate results. This violates the principle of direct manipulation [8], which dictates that users should have a visual representation of the object they are interacting with and should be able to make incremental changes to the object. Since S3's querying mechanism provides a guess for the joins, the query can be evaluated and presented to the user after each manipulation by the user. This would simply require submitting the partially completed form to the server and returning the current result.

Adding equations will make S3 more like a spreadsheet. Users should be able to define a column that is generated by some function applied to other columns. These functions could be based on another table. For example, a user could have a column of DVD identification numbers which would populate a column of movie titles by finding the title in another table. These join-based functions could also work in reverse: entering the movie title would result in a drop-down list of valid ids.

Records of the dependencies between columns and between cells, that result from functions, will be stored in order to update values after a change of arguments. Dar et al. [2] analyse the performance

of various partial transitive closure algorithms for databases. One such algorithm would be needed to efficiently find the set of cells that need updating.

Inferring functional dependencies is the next step in helping users normalize their schema. Some functional dependencies will be obvious from the equation system, but many will be implicit in the data. Most users will not have the experience or the willingness to even confirm potential functional dependencies, so this process must be handled entirely by the system. Dep-Miner [5] is one such algorithm.

Finally, the virtual tables which are currently views in the DBMS could be materialized. The view would simply be replaced with an actual table. The system would have to update the table as well as the `Cells`, but this is simple to manage since S3 is aware of all of the changes. This takes more space, but is much faster since no joins are needs to construct that table. Most importantly, indexes can then be built on the columns that are used in joins that generate reports. This would likely be used for tables that have many rows or have not changed schema in a long time.

# 7 Conclusions

Data-management is an enormous task facing any project today. The flexibility of a spreadsheet is wanted and needed by most users, but ultimately results in data that is not structured enough to perform real queries on. Databases have the capability to perform joins and queries, but require too much training for the average user. S3 exists to bridge the gap between these two methods by achieving two goals. The first goal is to allow users to input data in an irregular form and, over time, easily convert that data into normal form. The second is to allow a schema to evolve over long periods of time, since these changes are a fundamental part of the data being managed.

Users do not naturally create spreadsheets in normal form not only because they do not know how, but because spreadsheets in normal form are often too fine-grained and unnatural to interact with. By providing users with a querying mechanism that is always available, the need to repeat data across different tables is greatly diminished. Despite these gains, users will still generate tables that do not have good design, whether due to inexperience or due to changing needs. The repeated measurement subsystem allows for users to correct one of these common design mistakes and still input data as though the data had not been transformed.

By designing the system with a focus on cells, changing a schema involves no moving or deleting of data. Storing user input as a string and casting it when types are beneficial means that users are free to delay design decisions and make mistakes while the system can still sort and search correctly and efficiently.

S3 provides a gentle slope for users to enter their data in a way that is natural to them and, over time, to evolve that data into a well-structured database, while having full access to a dynamic schema at each step.

# References


[1] Steven S. Curl, Lorne Olfman, and John W. Satzinger. An investigation of the roles of individual differences and user interface on database usability. *The Database for Advances in Information Systems*, 29(1):50–65, 1998.

[2] Shaul Dar and Raghu Ramakrishnan. A performance study of transitive closure algorithms. *SIGMOD Rec.*, 23:454–465, May 1994.

[3] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 13–24, New York, NY, USA, 2007. ACM.

[4] Bin Liu and H.V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 417–428, April 2009.

[5] Stphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional de-

pendencies and armstrong relations. In *Advances in Database Technology – EDBT 2000*, volume 1777 of *Lecture Notes in Computer Science*, pages 350–364. Springer Berlin / Heidelberg, 2000.

[6] Boaz Ronen, Michael A Palley, and Henry C. Lucas, Jr. Spreadsheet analysis and design. *Commun. ACM*, 32:84–93, January 1989.

[7] Nick Roussopoulos and Dimitris Karagiannis. Conceptual modeling: Past, present and the continuum of the future. In *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 139–152. Springer Berlin / Heidelberg, 2009.

[8] Ben Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology*, 1:237–256, 1982.

[9] Jerzy Tyszkiewicz. Spreadsheet as a relational database engine. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 195–206, New York, NY, USA, 2010. ACM.

[10] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Shen, and Sankar Subramanian. Spreadsheets in rdbms for olap. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 52–63, New York, NY, USA, 2003. ACM.

[11] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Aman Naimat, Lei Sheng, Sankar Subramanian, and Allison Waingold. Query by excel. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1204–1215. VLDB Endowment, 2005.

[12] Moshé M. Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, AFIPS '75, pages 431–438, New York, NY, USA, 1975. ACM.