Project Number: **MYE005**

FusionDB: Conflict Management System for Small-Science Databases

A Major Qualifying Project submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science Degree of Bachelor of Science

Submitted By:

Nathaniel Selvo

Mohamad El-Rifai

April 25, 2013

APPROVED:

Advisor: Prof. Mohamed Y. Eltabakh

1.Abstract

Scientific databases usually involve collaboration among many scientific groups and labs within which scientists share their data at different stages of the exploration phases. In many cases, parts of the shared data are conflicting and values do not match with each other. Despite that, database management systems do not provide efficient support for handling and managing conflicting data. In this report, we present the ***FusionDB*** conflict management system, an extension to the open source PostgreSQL DBMS, and we discuss FusionDB new features and implementation details. The FusionDB system allows scientists to manage conflicts that arise in scientific data coming from multiple sources. Specific features of the system allow users to define what constitutes a conflict when comparing tables, querying conflicts discovered in the data, querying confidence levels of the data and prioritizing conflicts based on a number of criteria.

Table of Contents

List of Figures

2. Introduction

2.1 Overview

One of the most important parts of any scientific field such as biology, chemistry, physics, etc., is the collection and analysis of data. Every new experiment leads to new data, and this data is stored in a local database belonging to the scientists performing the experiment. However, science is also about collaboration, and so the data contained in all of these small-localized databases may be shared frequently, resulting in a relational database whose data are coming from multiple sources. Unfortunately, the fusion of data from more than one source--- also known as data integration across many sources--- introduces several critical problems because of the lack of standardization among the different sources. Thus, the data coming from one group may conflict with the data coming from another group, which may lead to confusing results and conclusions. Figure 2.1 shows examples of integration problems that arise because of the conflicting data at different levels.

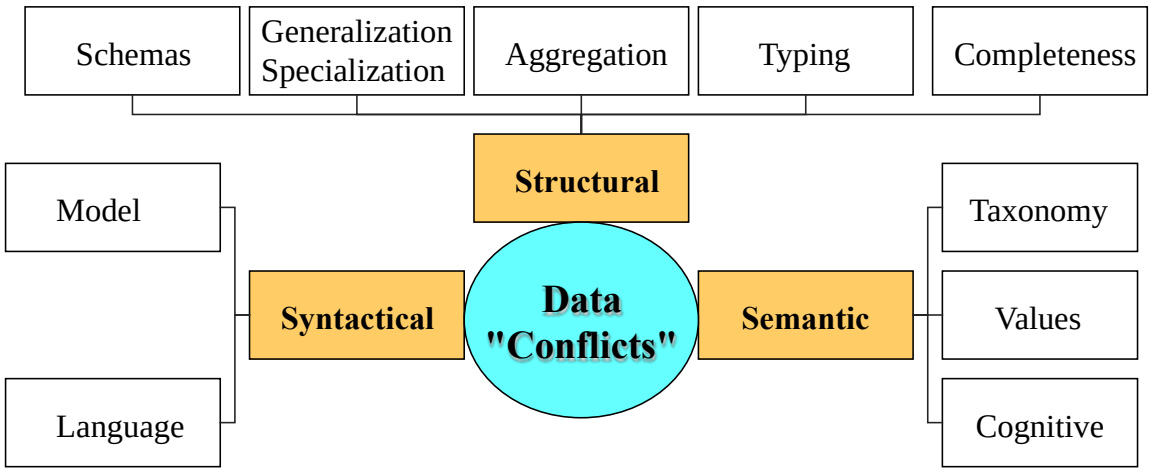


Figure 2.1: Reasons of Data Conflicts.

Existing relational databases lack various features that would be invaluable to scientists that wish to view and resolve the conflicts between the various sources in their database. Because the information in the database can take on many forms, it is not possible to fully automate the mechanism of determining what constitutes a conflict given a set of values. Moreover, data integration systems that aim for building a single consistent view of the data may not be applicable for many small-science databases. This is because in the later case, scientists may share their data at different stages of the discovery process with the aim for comparing and verifying results with each other, detecting and assessing conflicts as early as possible, and possibly refining experimental setups or adjusting parameters. Therefore, an important extension to existing database systems is needed to semi-automate the conflict management process in an easy and abstract way. It should allow users to define how a pair of relations should be compared and what "conflicting" means for those tables. Once the definitions of the conflicts have been created, the database should then not only keep track of these conflicts as they are discovered, but also calculate the correctness of all the values in the database by analyzing the conflicts. The database should then be extended with new querying capabilities that allow users to view the data's conflicts and correctness values. Querying could also be enhanced if the system manages how often individual values are queried, assigning values a priority based on this and allowing users to sort the data using this criterion. Lastly, the system should provide interfaces that allow scientists to review the conflicts in the system and use the queried information to resolve these conflicts in the manner of their choosing.

2.2 Focus and Goals of the Project

The goals of this project are as follows:

- Enhance the capabilities of the database management systems to semi-automate the conflict detection, assessment, and resolution.
- Enable users and scientists---especially in small sciences--- to define rules that determine conflicts in their data coming from multiple sources, and to query the conflicts found within a system and order them based on a number of criteria, and
- Allow users to query the data in the system while also considering the conflicts in the system, i.e., conflict-aware query processing.
- Providing high-level SQL interfaces for abstracting the new functionalities and enabling users to define them in an easy way.

2.3 Target Audience

The target audience for this project is: (1) Researchers interested in data management systems in general, and (2) users and scientists of small science databases who borrow data from one or more external sources. Currently, there is no simple way for these scientists to ensure that the data they have acquired from multiple sources is in agreement. Solving this issue should lessen the various costs associated with dealing with these data inconsistencies and allow scientists to have more confidence in their accumulated data.

2.4 Roadmap

The remainder of the report outlines and describes the *FusionDB* system. Chapter 3 provides an overview of the system, which covers the system's general features and interfaces. Chapter 4 goes into greater detail about the interfaces implemented in the system and describes the catalog tables used by the various interfaces. Chapter 5 details the execution engine, the SQL triggers that are created and executed as the *FusionDB* interfaces are put to use. Chapter 6 covers the querying and reporting capabilities of the system. Lastly, Chapter 7 covers the conclusion of *FusionDB* system project.

3. FusionDB: System Overview

3.1 Sharing Model in Small-Sciences

In most scientific applications, most discoveries and innovations are not driven by centralized processing, instead, they are done through many small-scale groups of scientists conducting their own experiments, collecting measurements, and storing their own data in local databases. Then, related groups working on the same (or similar) objects/subjects may collaborate by exchanging and sharing parts of their own data with each other. Such collaboration and sharing of data can be done at early stages of the discovery process with the aim for comparing and verifying results with each other, detecting and assessing conflicts as early as possible, and possibly refining experimental setups or adjusting parameters. These goals are different from those of traditional data integration systems that aim for building a unified and consistent warehouse. The sharing model in scientific data involves the following challenges:

(1) Scientific databases inherently contain conflicts that cannot be avoided or eliminated.

This is because there no single scientist or lab can curate the data, different interpretations and observations may lead to different values, and different scientists may have different beliefs about their data.

(2)Availability of conflicting data for analysis and querying. Although conflicting data may sometimes lead to confusion scientists prefer to have all the data available, even if conflicting, for analysis, querying, and comparison with other values.

(3) Ability to automatically assess conflicts and provide recommendations. Scientists are of great need for automatic mechanisms for detecting possible conflicts, assessing them,

and probably providing evidence-based recommendations on which values are more likely to be correct (or wrong).

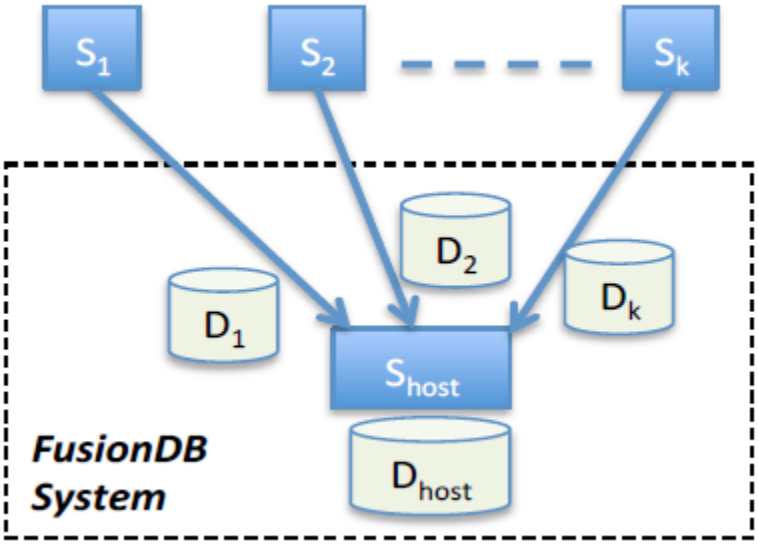


Figure 3.1:*FusionDB* System

Figure 3.1 shows an overview of the *FusionDB* system. The system has a central site and database, S_{host} and D_{host} respectively, and a number of external sites with their own data sources, S_1through $S_k$and D_1through D_k. The system allows the host to set a number of rules, which compare the host database to the external databases and report conflicts to the host.

3.2Glance on FusionDB Novel Features

The proposed FusionDB system is an extension to the open source PostgreSQL database management system. FusionDB provides the functionalities of identifying and querying conflicts between data from multiple sources. To implement these functionalities, we implemented several extensions that include SQL interfaces and commands, catalog tables, database triggers and functions, and also extend the query functionalities of PostgreSQL DBMS. The interfaces are the scientists' avenue for defining, querying and managing conflicts, while the tables, triggers

and functions support these interfaces and are not controlled directly by the user. Each interface in the FusionDB system realizes one of the previously mentioned functionalities and allows scientists to better manage data from multiple sources. In the following, we summarize the new interfaces and their basic definitions, and then in Chapter 4, we describe each interface in detail.

- ***Create Matching Rule*** **Interface**: The "Create Matching Rule" interface allows users to define a set of rules where each rule compares the tuples of two tables and finds the conflicts between them. For each rule that is defined, the interface creates a set of triggers which monitors the two tables in the rule and tracks any conflicts that are inserted or deleted. The "Create Matching Rule" interface requires a number of catalog tables in order to manage all of the system's conflicts. There is a table that stores the name of each matching rule, the tables it compares and the predicate that determines if two tuples are comparable. There are two additional tables that store rule information; one stores the names of the user-defined functions that calculate if a conflict exists, and the other stores the names of the arguments that should be provided to each function. The triggers of the "Create Matching Rule" interface are responsible for populating and updating two final tables that are used by other interfaces in the system. One table contains a list of conflicting tuples as well as the rule and function that reported each conflict. The final table contains a list of all the tuples in the system and counts the number of other values this tuple compares and conflicts with.
- ***Report Conflicts*** **Interface**: The "Report Conflicts" interface extends the querying capabilities of the PostgreSQL DBMS by allowing scientists to retrieve the conflicts associated with a table and its columns. The interface takes a query from the user, compares the results of the query to the catalog tables populated by the "Create Matching Rule"

interface, and then displays the information of any conflicts that involve the tuples of the given query.

- ***Keep Tracking*** **Interface**: The "Keep Tracking" interface allows the FusionDB system to manage the priority of all the tuples in the database. When the user appends "Keep Tracking" to the end of an SQL statement, any tuples involved in the query will have their priority increased. The priorities of all the tuples in the system are stored in a catalog table and the table is used to enhance the capabilities of other interfaces in the system.
- ***Prioritize Conflicts*** **Interface**: The "Prioritize Conflicts" interface further extends the querying capabilities of the PostgreSQL DBMS. It allows scientists to select all or (by using the "Report Conflicts" interface) a portion of the conflicts in the database and order them based on a criterion. By using catalog tables populated by the "Create Matching Rule" interface, the conflicts can be prioritized based on either the degree of each conflict or the number of conflicts each conflicting tuple participates in. Additionally, the conflicts can be sorted by the tuples popularity, which comes from the catalog table populated by the "Keep Tracking" interface. The conflicts can also be listed in ascending or descending order based on one of the three criteria.
- ***With Confidence*** **Interface**: The "With Confidence", allow scientists to view tuples in the system and how those tuples compare and conflict with others values. When "With Confidence" is appended to certain select statements, the system provides not only the data of the queried tuples, but can also provide the number of values each tuple compares and conflicts with as well as the confidence of the tuple, a value between zero and one calculated using the conflicts.

4. Rule-Based Conflict Detection

4.1Object Correspondence

The Create Matching Rule command allows users to define a matching rule, which compares the tuples of two tables using a set of given functions.

```
Create Matching Rule<id>As(
        Table<R> [As <R alias>]
        Table<S> [As <S alias>]
        WherePred(r, s)
        UsingF1(r.Ai, … r.Aj, s.Ak …, s.Am)
        [AndF2(r.A'i, … r.A'j, s.A'k, …, s.A'm)
        And …]);
```

<*id*>: The *id* value is a text which serves as both the name and unique identifier for the matching rule inside in the system.

<*R*>,<*S*>*:*** *R* and *S* (and their aliases) are the names of the relations whose tuples the matching rule will attempt to compare.

Pred(r, s): The *Pred(r, s)* function is a user-defined function or SQL statement, which returns true if the tuples of *R* and *S* are comparable and false otherwise. In the catalog table, this function can be stored as a string of text.

F1,…,Fi*:* The functions F_1through F_i are user defined functions which compare tuples from *R* and *S*, returning 0 if there is no conflict and a value more than 0 if there is a conflict. In the catalog table, these functions can be stored as string of text.

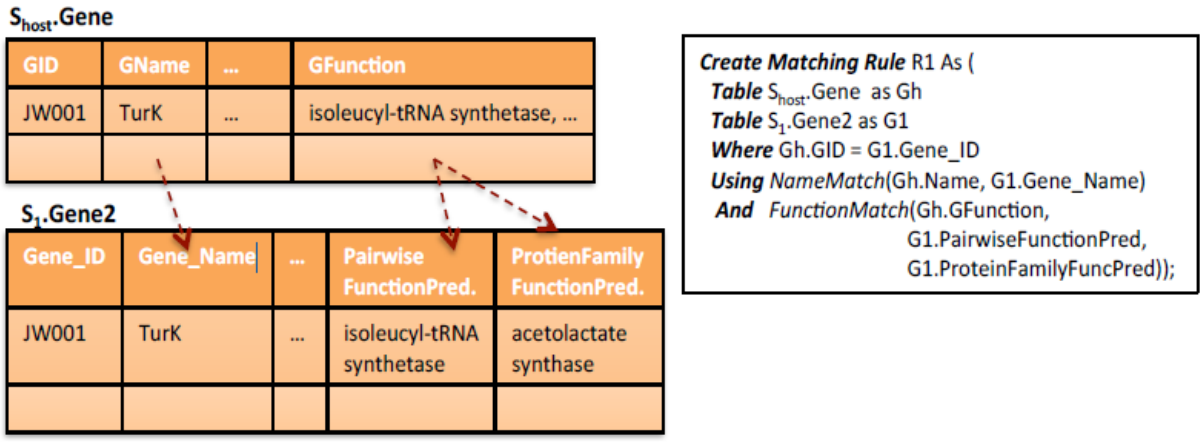


Figure 4.1: *Create Matching Rule* Gene Example.

Figure 4.1 provides an example of implementing a *Create Matching Rule* which compares genes stored in a number of databases. In this rule, the host database compares the genes in its host database to genes with the same ids in an external database. The rule uses two functions, one to ensure that the names of the genes match, and another to compare the other details of the genes.

4.2Populating Catalog Tables

After the command is entered, the rule's information is parsed from the command, and then a series of insert commands is executed to fill the necessary tables with the parsed information. First, the rule's name, tables, predicate and number of functions are stored in matching_rule_table:

```
INSERT INTO matching_rule_table VALUES(
            <id>,
            <R>,
            <R alias>,
            <S>,
            <S alias>,
            Pred(r, s),
            <number of functions>);
```

Next, the information for each function of the rule is stored in Matching_rule_functions:

```
INSERT INTO matching_rule_functions VALUES(
            <function key>,
            <id>,
            Fi,
            <number of arguments in Fi>);
```

Finally, the information for each argument in each function is stored in Matching_rule_arguments:

```
INSERT INTO matching_rule_arguments VALUES(
            <function key>,
            <argument number>,
            <argument table>,
            <argument column>);
```

Once all of the parsed information is stored in the appropriate tables, the tables are then used to generate triggers that will monitor and update conflict information in tables R and S.

Since conflicts may change with any change in the rows of either table, six triggers need to be generated. Three of the triggers are after insertion, deletion and update on table R, and the other three are after insertion, deletion and update on table S.

4.3Matching Rule Triggers

As the information in various user tables in the system are updated and changed, the information in the new catalog tables will need to change as well. To perform these changes, various types of triggers are created as matching rules are created, and an additional trigger manages the priority of tuples. The logic and format of these triggers are defined below.

4.3.1. After Insert

Once a *Create Matching Rule* has been defined, any tuples added to the tables that the rule compares must be checked for conflicts. The purpose of this trigger is to identify

```
After Insert r IntoR
//statements executed to update matching_rule_conflict_info
comparable_stmt text;
conflict_stmt text;
For s in Select * From S Where Pred(r, s) Loop
      //update comparable_stmt here
      For each function F_i(r, s)in the matching rule Loop
            If F_i(r, s) reports a conflict Then
                  //update conflict_stmt here
                  Insert conflict into matching_rule_conflicts
      End If
End Loops
If comparable tuples were found Then Execute comparable_stmt
If conflicting tuples were found Then Execute conflict_stmt
```

comparable and conflicting tuples and to update the necessary tables.

4.3.2. After Delete

Once a *Create Matching Rule* has been defined, whenever any tuples are deleted from the tables that the rule compares, the tables that store conflict information must be updated. The purpose of this trigger is to remove conflict information concerning tuples that no longer exist.

```
After Delete r From R
//statements executed to update matching_rule_conflict_info
comparable_stmt text;
conflict_stmt text;
For s in Select * From S Where Pred(r, s) Loop
      //update comparable_stmt here
      If matching_rule_conflicts has conflicts between r & s Then
            //update conflict_stmt here
            Delete conflicts From matching_rule_conflicts
      End If
End Loop
If comparable tuples were found Then
      Execute comparable_stmt End If
If conflicting tuples were found Then
      Execute conflict_stmt End If
```

4.3.3. After Update

Once a create matching rule has been defined, whenever any tuples are updated in the tables that the rule compares, the tables that store conflict information must be updated. With regards to the *Create Matching Rule*, an update is considered to be a delete followed by an insert, so no new trigger logic is implemented.

We note that, although in the scope of this project, we implemented updates as a sequence of deletion followed by insertion; there is a room for optimization if we handle the updates with their own logic in the *After Update* triggers. For example, if the user in updating a column that is not part of the existing matching rules, then this update will have no effect on the comparable tuples or the conflicting data. This kind of optimization is left for future work.

5. FusionDB Querying and Reporting

5.1. Report Conflicts

The Report Conflicts statement takes a select statement on one table as an argument, and reports all the conflicts that are related to this table. Report Conflicts doesn't support Where clause in the select statement. This is a current limitation that is left to be addressed in future work.

The syntax for the ReportConflict command is as follows:

```
ReportConflicts In (Select R.Ai,… From Table <R>);
```

Report Conflicts Example:

In this example, we assume we have two tables T1 and T2 and a rule that compare column X in both tables and if they match, then this means that the two records refer to the same object and hence the Y column values must also match, otherwise there is a conflict.

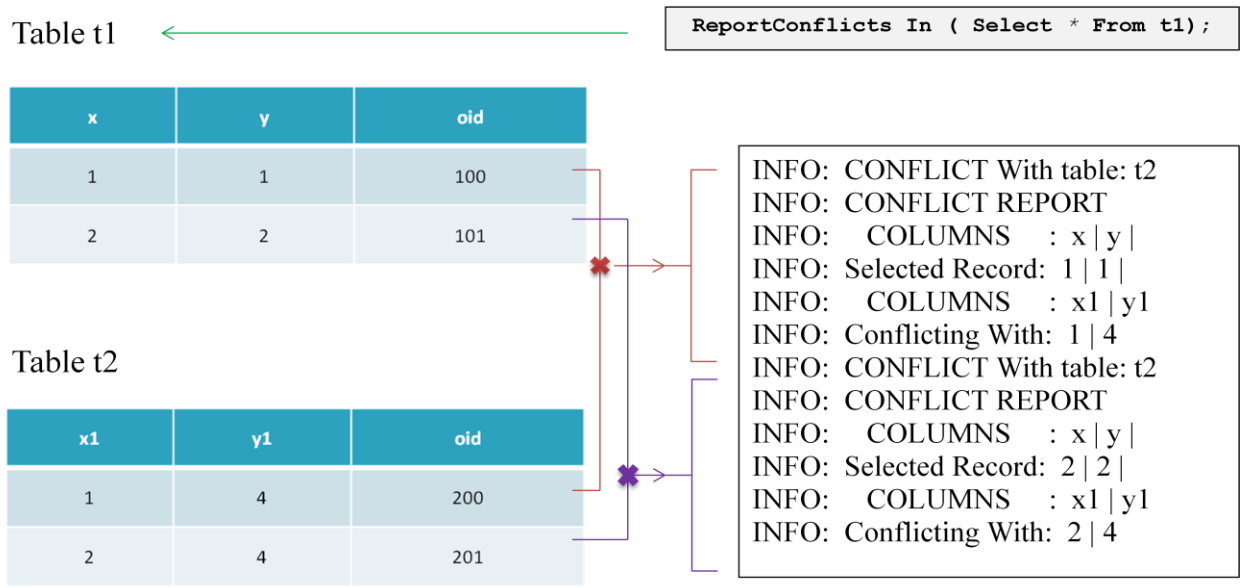
The matching rule defined on the tables is:

```
Create Matching Rule test As (
Table t1 Table t2
Where t1.x=t2.x1
Using check_match (t1.y, t2.y1));
```

Now, to report the conflicts on table T1, we use the following command:

```
ReportConflicts In (Select * From T1);
```

The output looks like the following:



5.2. Keep Tracking

The Keep Tracking clause can be added to the end of a select statement on a single or multiple tables which will allow the PostgreSQL system to detect and increment the popularity of the selected records or the records involved in built in aggregates. This clause enables the system to keep track of the popularity of records in the database, i.e., records that are queried more frequently are more popular than other records. Such popularity can be used to give popular records higher priority in resolving their conflicts than other records (Refer to Section 5.3).

The command with the Keep Tracking clause is as follows:

```
Select R.Ai,… From Table<R>Keep Tracking;
```

Keep Tracking Example:

Table t1

id	value	oid
x	1	100
y	2	101

Table t2

id	value	oid
x	4	102
y	4	103

TuplesPriorititites

tablename	tupleoid	priority
t1	100	1
t1	101	1
t2	102	1
t2	103	1

```
Select*Fromt1 Where value > 1Keep Tracking;
```

Query Output

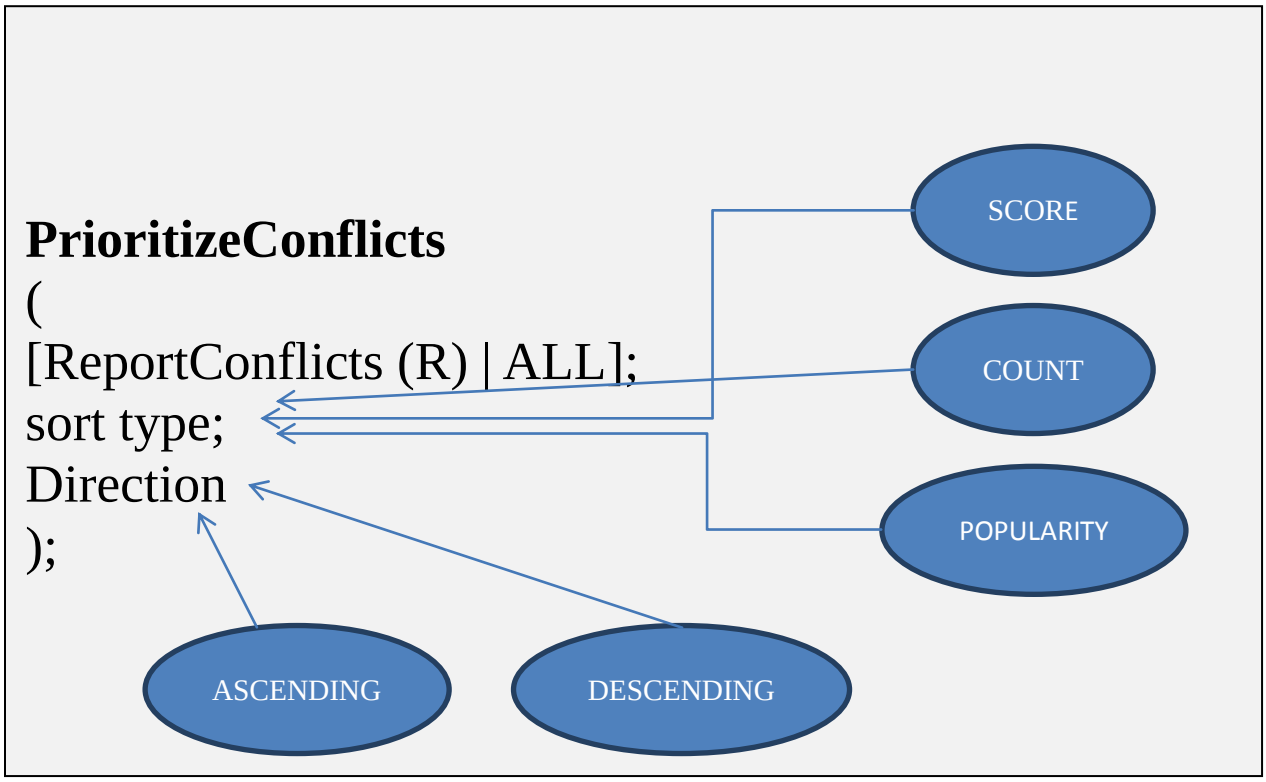
id	value	oid
y	2	101

TuplesPriorititites (Updated)

tablename	tupleoid	priority
t1	100	1
t1	101	2
t2	102	1
t2	103	1

5.3. Prioritize Conflicts

Prioritize Conflicts allows scientists to select all records or (by using the "Report Conflicts") a portion of the conflicts in the database and order them based on a criterion.



Prioritize Conflicts Example:
In this example we assume we have a rule that compare column x in t1 if it equal column x1 in table t2 and check if column y in table t1 is not equal to column y1 in table t2.

Table t1

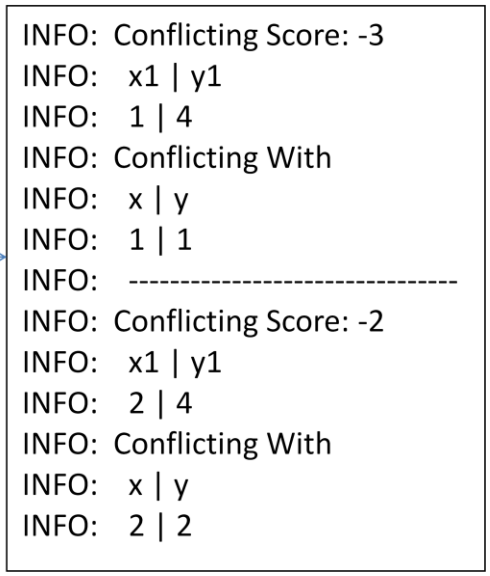
x	y	oid
1	1	100
2	2	101

Table t2

x1	y1	oid
1	4	200
2	4	201

Prioritize Conflict with respect to conflict Score in matching_rule_conflicts table:

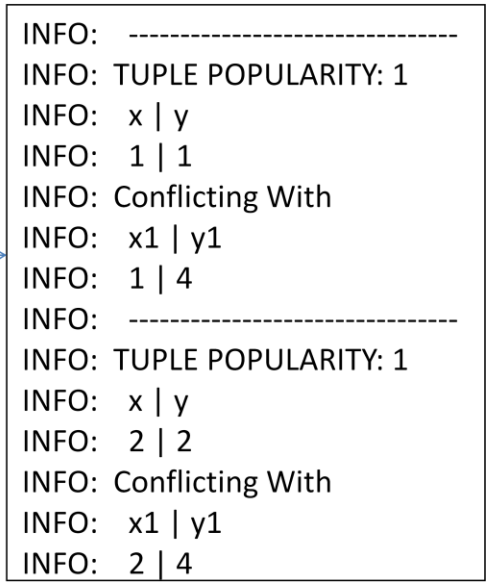
```
PrioritizeConflicts (
ReportConflicts in (select * from t1),
score,
asc);
```



Prioritize Conflict with respect to popularity column in Tuplespriorities table. In order to see the effect of prioritize conflict with popularity we should have been used select statements on the target table with using keep tracking capability.

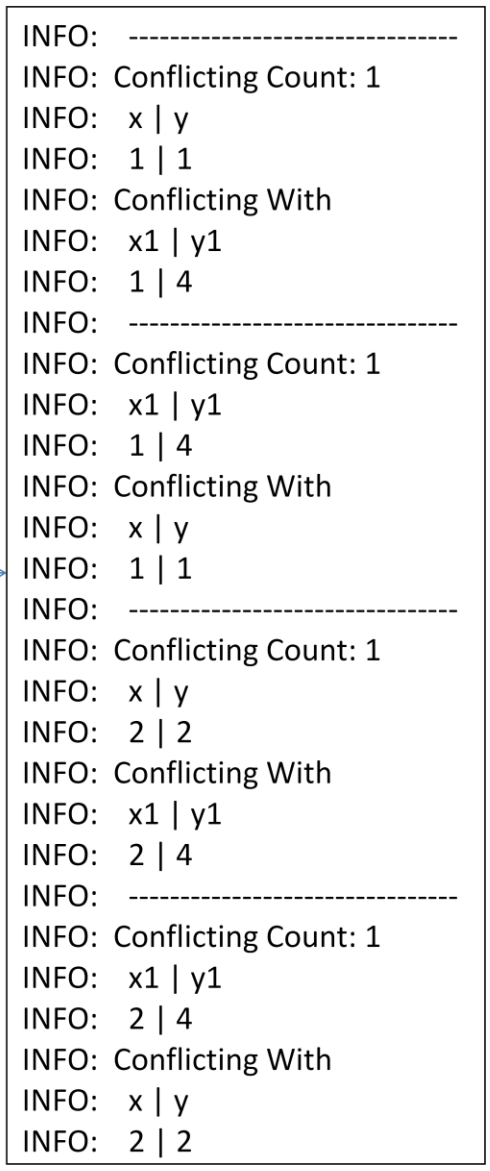
```
Select * from t1 keep tracking;
```

```
PrioritizeConflicts (
ReportConflicts in (select * from t1),
popularity,
asc);
```



Prioritize Conflict with respect to conflict_count column in Matching_rule_conflict_info table

```
PrioritizeConflicts (
ReportConflicts in (select * from t1),
cnt,
asc);
```



5.4. Insert If Not Conflicting

The If Not Conflicting clause can be added to the end of an Insert statement, which will cause the statement to be rejected to do insert in the table if the record we are trying to insert is conflicting with any other record.

```
Insert into Table<R> Values (...) If Not Conflicting;
```

If the insert statement has If Not Conflicting at the end of it, we will update a Boolean flag in if_not_conflicting table called conflicted and set it to be true, so later on if this record is conflicting then the Insert trigger will go to update FusionDB catalog tables to report the conflict, so at this point and before the trigger update the tables we will query if_not_conflicting table and if conflicted flag is true then we exit from the trigger and we will print warning message to the user.

5.5. Query Answers with Confidence

The With Confidence statement is an addition to the end of a select statement on a single table which will cause the statement to return not only the requested information, but also the conflict info for any selected rows.

```
Select <columns> From <table> [Where <predicate>] With Confidence
```

Once the command is entered, additional information is added to the various clauses in the statement. The new statement has the following format:

```
Select <columns>, calculate_confidence (comparable_count, conflict_count)
From<table>, Matching_rule_conflict_info
Where [<predicate>And] Matching_rule_conflict_info.table_name Like '<table>'
And <table>.oid = Matching_rule_conflict_info.tupleoid;
```

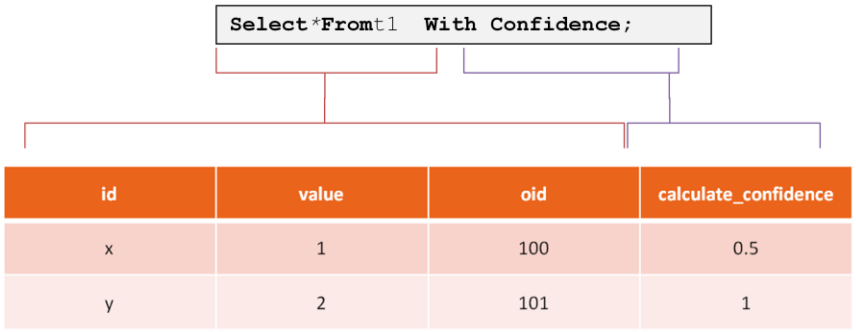
This new select is then executed, and both the rows and their confidence scores are returned to the user. For example, observe the following two tables:

Table t1

id	value	oid
x	1	100
y	2	101

Table t2

id	value	oid
x	1	200
x	4	201
z	0	202



6. FusionDB Catalog Tables

In order to keep track of the matching rules between tables, the conflicts reported based on these rules and the priority of different tuples, catalog tables must be added to the database. Detailed below are the catalog tables that are needed, the command needed to create them, and what the columns of each table represent.

6.1. Matching_rule_table

ruleid	table1_name	table1_alias	table2_name	table2_alias	predicate	function_count
rule1	R	NULL	S	NULL	R.r1 = S.s1	1
rule2	S	NULL	T	NULL	S.s2 = T.t3	2
…	…	…	…	…	…	…

ruleid (text): The unique name for the matching rule.

table1_name (text): The name of the first table being compared by the rule.

table1_alias (text): An optional alias for the first table in the rule.

table2_name (text): The name of the second table being compared by the rule.

table2_alias (text): An optional alias for the second table in the rule.

Predicate (text): The predicate by which the tuples in the tables will be compared. If the predicate returns true for two tuples, they will be checked for conflicts between them.

function_count (integer): The number of functions used in this rule to compare tuples. Detailed information about these functions is stored in Matching_rule_functions.

6.2. Matching_rule_functions

pk_id	ruleid	function_name	argument_count
rule1compare1	rule1	compare	2
rule2compare1	rule2	compare	2
rule2compare2	rule2	compare	2
…	…	…	…

pk_id (text): A unique key for each function in the table. The key is a concatenated text made from create matching rule the function applies to, the name of the function and the function's order in the rule. For example, if the function "check_conflict" was part of the rule "test" and was the third function in the rule, its pk_id would be "testcheck_conflict3".

ruleid (text): The rule to which this function applies.

function_name (text): The name of the function.

argument_count (integer): The number of arguments taken by this function. Detailed information about the arguments is stored in Matching_rule_arguments.

6.3. Matching_rule_arguments

function_pk_id	argument_order	argument_table_name	argument_column_name
rule1compare1	1	R	r2
rule1compare1	2	S	s2
rule2compare1	1	S	s1
rule2compare1	2	T	t1
rule2compare2	1	S	s4
rule2compare2	2	T	t4
…	…	…	…

function_pk_id (text): The id of this argument's function found in Matching_rule_functions.

argument_order (integer): The position of this argument in its function's signature.

When the functions used in a matching rule are defined, the arguments passed to each function must be given in the format X.Y, where X is the name of the table the argument belongs to, and Y is the name of the column.

argument_table_name (text): The table of the argument.

argument_column_name (text): The column of the argument.

6.4. matching_rule_conflicts

table1_name	tuple1oid	table2_name	tuple2oid	ruleid	function_pk_id	score
R	10000	S	20000	rule1	rule1compare1	-3
S	20004	T	30006	rule2	rule2compare1	2.2
S	20004	T	30003	rule2	rule2compare2	4
…	…	…	…	…	…	…

table1_name (text): The name of the table where the first conflicting tuple is found.

tuple1oid (oid): The oid of the first conflicting tuple.

table2_name (text): The name of the table where the second conflicting tuple is found.

tuple2oid (oid): The oid of the second conflicting tuple.

ruleid (text): The name of the rule under which the two tuples conflict.

function_pk_id (text): The id of the function that is reporting a conflict between the two tuples.

score (numeric): The conflict score returned by the conflict-reporting function.

6.5. Matching_rule_conflict_info

table_name	tupleoid	comparable_count	conflict_count
R	10000	2	1
S	20000	1	1
S	20004	2	2
T	30000	0	0
T	30003	4	1
…	…	…	…

table_name (text): The name of the table to which the tuple belongs.

tupleoid (oid): The oid of the tuple.

comparable_count (integer): The number of tuples that compare with this tuple.

conflict_count (integer): The number of tuples that conflict with this tuple.

6.6. TuplesPriorities

tableName	tupleOid	priority
R	10000	4
S	20000	7
S	20004	11
T	30000	6
T	30003	17
…	…	…

tableName (text): The name of the table to which the tuple belongs.

tupleOid (integer): The oid of the tuple.

priority (integer): The priority of the tuple.

7. Conclusion

In this project, we studied several data integration issues that arise from collecting and sharing scientific data from different sources and datasets. The data conflict, which is a result of data integration problem, is a big dilemma in databases systems because it is very hard and time-consuming process to detect conflicts manually, especially in large-scale databases. We focused on small-science databases, where collaborating scientists may share parts of their own data with each other especially at early stages of the discovery process with the aim for comparing and verifying results with each other, detecting and assessing conflicts as early as possible, and possibly re-running experimental setups or adjusting parameters. These goals are different from those of traditional data integration systems that aim for building a unified and consistent warehouse. Hence, our approach in this project with different from existing systems

Our mission in this project was to extend PostgreSQL system to allow scientists to manage conflicts that arise in scientific data coming from multiple sources in an automatic and systematic way. We proposed to enhance PostgreSQL with several features that give the user the ability to define rules on top of specific tables that can compare records and detect the degree of conflict along them. We also introduced several mechanisms to (1) Assess the conflicts and provide a confidence value for each data item in the database based on its conflicts, and (2) Allow users to query and prioritize conflicts in different ways that match different application requirements.

We believe the proposed FusionDB system with its new features and capabilities would help scientists speeding up their discovery process and increases the efficiency of sharing data among multiple collaborators in a scalable way.

Appendix A

Catalog Table CREATE TABLE Functions

Matching_rule_table

```
Create Table Matching_rule_table(
          ruleid text PRIMARY KEY,
          table1_name text,
          table1_alias text,
          table2_name text,
          table2_alias text,
          predicate text,
          function_count integer);
```

Matching_rule_functions

```
Create Table Matching_rule_functions(
          pk_id text PRIMARY KEY,
          ruleid text
               REFRENCES matching_rule_table(ruleid),
          function_name text,
          argument_count integer);
```

Matching_rule_arguments

```
Create Table Matching_rule_arguments(
          function_pk_id text
               REFRENCES matching_rule_functions(pk_id),
          argument_order integer,
          argument_table_name text,
          argument_column_name text);
```

Matching_rule_conflicts

```
Create Table Matching_rule_conflicts(
          table1_name text,
          tuple1oid oid,
          table2_name text,
          tuple2oid oid,
          ruleid text
               REFERENCES Matching_rule_table(ruleid),
          function_pk_id text
               REFRENCES Matching_rule_functions(pk_id),
          score numeric);
```

Matching_rule_conflict_info

```
Create Table Matching_rule_conflict_info(
          table_name text,
          tupleoid oid,
          comparable_count integer,
          conflict_count integer);
```

TuplesPriorities

```
Create Table tuplesPriorities(
          table_name text,
          tupleOid integer,
          priority integer);
```

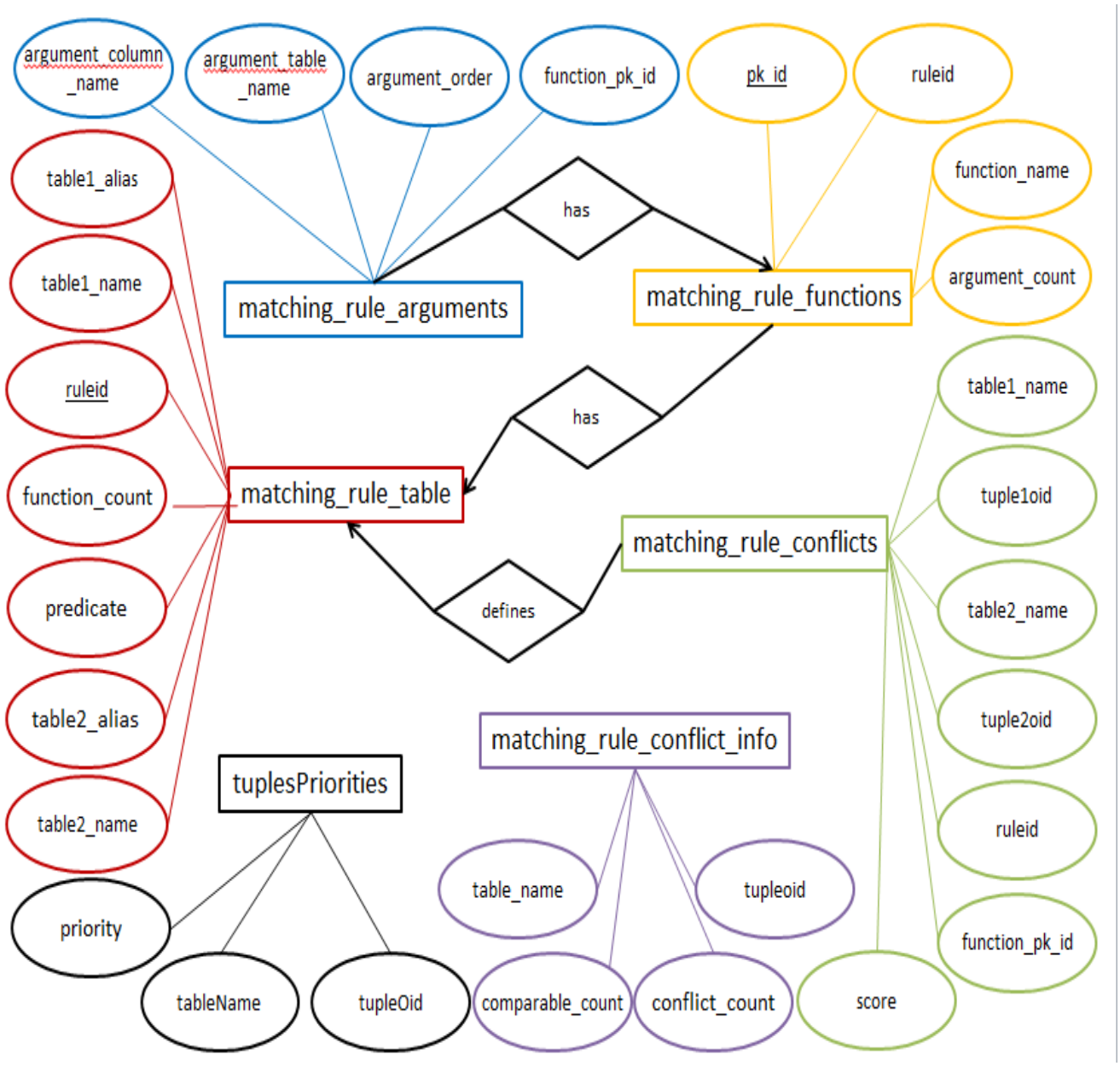
Supplementary Functions

calculate_confidence()

```
Create Or Replace Function calculate_confidence
     (comparable integer, conflict integer)
     Returns Real As
     $BODY$
          Begin
               If conflict = 0 Then
                    Return 1;
               Else
                    Return
                    1-((0.0 + conflict/(0.0 + comparable));
               End If;
          End;
     $BODY$
     Language plpgsql;
```

The calculate_confidence function is used by the “with confidence” command to find the confidence value of each selected tuple. If the given tuple does not compare to any other tuple, the confidence of the given tuple is one.

Catalog Table Entity Relationship Diagram



Appendix B

Install and Configure FusionDB on Ubuntu

1- Download the source code from: http://www.postgresql.org/ftp/source/

2- Unzip the downloaded folder

3- Move to the downloaded folder, for example:

- cd ~/Downloads/postgresql-9.0.3

4- Now you can configure and install PostgreSQL by executing the below commands:

- ./configure
- sudo make
- sudo make install

5- Create the data directory to store all data files and do some configuration:

- sudomkdir /usr/local/pgsql/data
- sudoadduserpostgres
- sudochownpostgres /usr/local/pgsql/data
- su - postgres

6- Initialize the database using the following command:

- /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data/

7-Start the database server using the following command:

- /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
  OR
- /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start

8- Create log directory and a sample database:

- cd /usr/local/pgsql/data/
- mkdir log
- /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >/usr/local/pgsql/data/log/logfile 2>&1 &
- cd ..
- bin/createdbmydb
- bin/psqlmydb