DataJoint Tutorial

Dimitri Yatsenko, Alexander Ecker, Andreas S. Tolias
Baylor College of Medicine

March 27, 2011

Contents

1 Overview **4**
1.1 What is DataJoint . . . 4
1.2 Conditions of use . . . 4
1.3 Resources . . . 4
1.4 History . . . 4
1.5 Prerequisites . . . 4
1.6 Foundations . . . 4
1.7 Components . . . 4

2 The relational data model: basic concepts **5**
2.1 Background . . . 5
2.2 Attributes and tuples . . . 5
2.3 Relations . . . 5
2.4 Matching tuples . . . 5
2.5 Primary key . . . 5

3 Data management in a science lab **6**
3.1 Project schema . . . 6
3.2 Data tiers . . . 6
3.3 Schema example . . . 6

4 Defining tables **7**
4.1 Create a MySQL schema . . . 7
4.2 Define the schema class . . . 7
4.3 Declare tables . . . 7
4.3.1 Instantiate a `DeclareDJ` object . . . 7
4.3.2 Specify parent tables . . . 7

4.3.3 Add primary key fields . . . 7
4.3.4 Add non-key table fields . . . 8
4.3.5 Field datatypes . . . 8
4.3.6 Add reference fields . . . 9
4.3.7 Omit the timestamp field . . . 9
4.3.8 Execute the declaration . . . 9
4.4 Set the table comment . . . 10
4.5 Viewing tables . . . 10
4.6 Reverse engineering existing tables . . . 10
4.7 Referential integrity . . . 10
4.8 Keep tables normalized . . . 10

5 Entering data into manual tables **12**
5.1 Entering manual data from MATLAB . . . 12
5.2 Upload from .MAT files and download to .MAT files . . . 12
5.3 Entering data through third-party database interfaces . . . 12
5.4 Delete . . . 12

6 Data queries **13**
6.1 Retrieving data: `fetch`, `fetch1`, and `fetchn` . . . 13
6.2 Relational operators . . . 13
6.2.1 Restrict . . . 13
6.2.2 Projection . . . 14
6.2.3 Semijoin and antijoin . . . 15
6.2.4 Join . . . 15
6.2.5 Union and difference (presently excluded) . . . 16

7 Populating tables automatically **17**
7.1 Setting the populate relation . . . 17
7.2 Filling out computed fields . . . 17
7.3 Subtables . . . 18
7.4 Transaction processing . . . 18

8 Distributed Jobs **19**
8.1 Defining the job callback . . . 19
8.2 Parallel execution with `runJob` . . . 19

9 Appendix A: The Vis2p schema example **20**

9.1 Scenario . . . 20

9.2 Entity relationship diagram . . . 20

9.3 Declarations for Mice, Sessions, and Scans . . . 20

9.4 Table declarations for visual stimuli . . . 21

10 Appendix B: Helpful SQL expression syntax **23**

10.1 comparison operators . . . 23

10.2 logical operators . . . 23

10.3 detecting missing values . . . 23

10.4 arithmetic operators and functions . . . 23

10.5 membership operators . . . 23

10.6 date and time functions . . . 23

10.7 aggregate functions . . . 23

11 Appendix C: Internal conventions **24**

11.1 Table names . . . 24

11.2 Foreign keys . . . 24

1 Overview

1.1 What is DataJoint

DataJoint is a MATLAB tool for the distributed processing and management of large volumes of data in a science lab. DataJoint is built on the foundation of the relational data model and prescribes a consistent method for organizing, populating, and querying the data with minimal chance of loss of data integrity.

1.2 Conditions of use

DataJoint is available for free use and distribution under the GNU General Public License version 3 (http://www.gnu.org/licenses/gpl.html)

We also request that any lab using DataJoint for processing data leading to a publication acknowledge this fact by including a variant of the following statement in their publication:

> "The data processing chain for this publication was implemented using the DataJoint library from Andreas S. Tolias' lab at Baylor College of Medicine, Houston, TX"

1.3 Resources

DataJoin code and documentation may be downloaded from http://code.google.com/p/datajoint/. Although we cannot provide technical support, common questions may be address in the wiki and user forum sections of the website.

1.4 History

DataJoint was developed by Dimitri Yatsenko in Andreas Tolias' lab at Baylor College of Medicine beginning in October 2009 with the first alpha release scheduled for March 2011. DataJoint was inspired in part by the earlier data management tool in the lab called Steinbruch developed by Alex Ecker and Philipp Berens.

1.5 Prerequisites

We assume that the reader is proficient with MATLAB and its object-oriented features in particular. Readers do not need to be a priori familiar with relational databases but will need to master the basic relational concepts presented in this tutorial. We also assume the availability of a MySQL server in the lab with accounts and schemas for each user. The installation and administration of the MySQL server is not covered in this tutorial.

1.6 Foundations

DataJoint adheres faithfully to the theoretical rigor and simplicity of the relational model[1]. This discipline then allows avoiding common pitfalls of data organization and affords simpler ways of querying data. DataJoint implements an intuitive relational algebra which allow formulating succinct and precise queries. As the result, even skilled SQL programmers may prefer DataJoint queries to SQL while novice users will quickly become proficient and efficient with a minimal set of correct relational concepts bypassing SQL's arcane syntax.

DataJoint relies on MySQL's native constructs for data organization. All data organized in DataJoint is transparently accessible by other database interfaces.

1.7 Components

The current version of DataJoint is developed in MATLAB 2007. For its database interface, DataJoint uses a modified version of the `mym` library from http://mym.sourceforge.net/.

DataJoint comprises two MATLAB classes: `DeclareDJ` and `DJ`.

Class `DeclareDJ` helps declare and organize tables with data. Each table in a DataJoint schema has a homonymous class that inherits from class `DJ`.

Class `DJ` implements all data query and manipulation functionality. `DJ` supports a relationally complete algebra that affords flexible, expressive, and succinct data queries.

[1] See C. Date, *SQL and Relational Theory*, 1st Edition, O'Reilly, 2009.

2 The relational data model: basic concepts

2.1 Background

Invented by IBM researcher Edgar F. Codd in 1969,[2] the relational model for databases steadily replaced the earlier hierarchical and network models and has become the de facto standard for mainstream databases today, supporting banking transactions, airfare bookings, and data-intensive websites such as Facebook, Google, Wikipedia, and YouTube, to pick but a few examples. Modern relational database management systems execute fast, precise, and flexible data queries and preclude inconsistencies arising from simultaneous or interrupted manipulations by multiple users. Interactions with a relational database are performed in a query language such as SQL.

Below we provide a brief semiformal overview of basic concepts of relational concepts as implemented by DataJoint.

2.2 Attributes and tuples

A *tuple* is a set of attribute name/value pairs. For example, the tuple

```
mouse_id   measure_date   weight
  1001      2010-10-10     21.5
```

in a given relation may represent a real-world fact such as "On Oct. 10, 2010, mouse #1001 weighed 21.5 grams." An attribute name is more than just a name: it implies a particular *datatype* and a unique *role* in the tuple and in the external world. Thus attribute names must be unique in a tuple and their order in the tuple is not significant.

The closest equivalent of a tuple in Matlab is a structure. We will use the terms *attribute* and *field* interchangeably.

2.3 Relations

A relation is a set of tuples that share the same set of attribute names. No duplicate tuples can exist in a relation. The ordering of tuples in a relation is not significant.

Starting with *base relations* corresponding to *tables* in the database, we can transform them into *derived relations* by applying *relational operators* until they contain only all the necessary information, and then retrieve their values into the MATLAB workspace. We will use the term *table* and *base relation* interchangeably.

2.4 Matching tuples

The key concept at the foundation of data manipulations in the relational model is *tuple matching*. Two tuples match if their identically named attributes contain equal values.

Thus one tuple may be used to address a group of other matching tuples in a relation (see subsubsection 6.2.1). Two tuples may be merged into one if they match, but not otherwise. The *join* (see subsubsection 6.2.4) of two relations is the set of all possible merged pairs from the two original relations.

2.5 Primary key

In DataJoint, each relation must have a *primary key*, i.e. a subset of its attributes that are designated to uniquely identify any tuple in the relation. No two tuples in the same relation can have the same combination of values in their primary key fields. To uniquely identify a tuple in the relation, one must provide the values of the primary key fields as a matching tuple.

[2] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):387, 1970

3 Data management in a science lab

3.1 Project schema

In DataJoint, every study comprising a set of experiments and related data is organized as a dedicated schema: a collection of logically related tables (Figure 1). DataJoint creates tables with properly configured referential integrity constraints and a dedicated MATLAB class that supports all manipulations on the table.

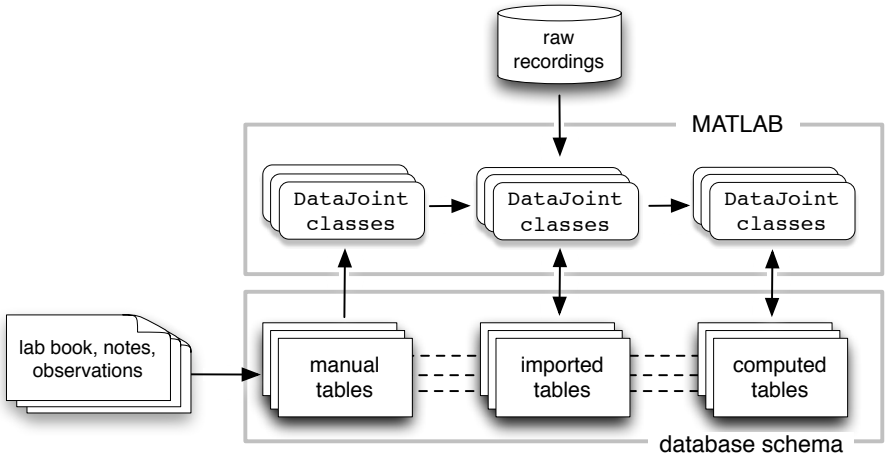

Figure 1: Data organization in a DataJoint schema.

3.2 Data tiers

Every table in a DataJoint schema is assigned one of the four tiers: *lookup tables*, *manual*, *imported*, and *computed* (Figure 1):

1. **Lookup tables** contain information that is involved in setting up the processing chain itself: filter settings, algorithm parameters, etc. Lookup tables are populated manually.

2. **Manual tables** contain information about experiments that is entered manually, e.g. subject information, independent variables, acquisition settings, visual observations, and paths to recorded data. Due to their high value and small size, manual tables are backed up most frequently (e.g. at every update).

3. **Imported tables** contain information extracted from files outside the database (e.g. raw recordings). Raw recordings are typically in large files that cannot be kept online indefinitely. Once the extraction algorithms have stabilized and the extracted data have been imported into the database, the raw recordings may be taken offline. Imported tables are populated automatically and need to be backed up regularly, especially after the raw data files have been taken offline.

4. **Computed tables** are populated automatically with data computed from other data already present in the database. Computed tables do not typically need to be backed up at all.

3.3 Schema example

To illustrate DataJoint usage in practice, section 9. shows a snippet of a schema named `Vis2p` used in our lab.

4 Defining tables

This section explains how to create a set of interdependent tables with referential constraints using the class `DeclareDJ`.

4.1 Create a MySQL schema

First, ask your database administrator to create a dedicated schema for your project. You will receive the host name, username, password, and the schema name.

4.2 Define the schema class

For every schema, there must exist an abstract class which assigns the connection parameters and inherits from `DJ` all the relational methods. All other classes will inherit from this schema class and will know how to connect to the database.

The `Vis2p` constructor below sets the schema connection parameters:

```
function obj = Vis2p( tableName, varargin )
conn.host = '<host-address>';
conn.schema = '<your_project_schema>';
conn.user = '<user_name>';
conn.pass = '<password>';
conn.schemaClass = mfilename;
conn.table = '';
if nargin>0
    conn.table = tableName;
end
obj = class(struct,'Vis2p',DJ(conn,varargin{:}));
```

In practice the connection parameters may be stored in a global variable that is set at startup to allow multiple user credentials while using the same code. The schema name does not need to match the schema class name.

4.3 Declare tables

4.3.1 Instantiate a `DeclareDJ` object

Various types of discrete facts about your study must be assigned their own tables in the database. Let's begin with manually populated tables `Mice`, `Sessions`, and `Scans` (subsection 9.3). These tables contain information about mice used in experiments, two-photon imaging sessions performed on these mice, and individual two-photon recordings (scans) performed within each session.

Each declaration begins by instantiating the object `ddj` of class `DeclareDJ` and specifies the schema class (`Vis2p` in this example), the table name, and the table tier (*manual*, *lookup*, *imported*, or *computed* as described subsection 3.2).

4.3.2 Specify parent tables

Tables may be independent (e.g. `Mice`) or subordinate (e.g. `Sessions` and `Scans`). A subordinate table can only contain tuples that match exactly one tuple from each of their parent tables.

Use the `ddj=addParent(ddj,P1,..,Pn)` method to set the parent/child dependency in a declaration, where `P1`,...,`Pn` are existing tables.

4.3.3 Add primary key fields

Each table must have a primary key: one field or a combination of fields that uniquely identify a row in the table. The database uses the primary key to prevent duplicate entries, to relate data across tables, and to accelerate data queries. The choice of the primary key will determine how you identify data elements. Therefore, make the primary key **short**, **expressive**, and **persistent**. For example, mice in our lab are assigned unique IDs. The mouse ID can serve as the primary key for the table `Mice`.

Add another primary key field as

```
ddj=addKeyField(ddj,'datatype','comment');
```

See subsubsection 4.3.5 for supported datatypes. Integers, dates, and timestamps make best primary key fields. To protect users from choosing poor primary key fields, `DeclareDJ` prohibits character strings, enums, and blobsin primary keys.

Subordinate tables automatically inherit the union of the primary key fields of their parent tables. In the case of one-to-one dependencies, no additional key fields are needed (e.g. table `Stims` in subsection 9.4 inherits its entire primary key from table table `Scans`). If multiple tuples are possible for each tuple in the parent table, then an additional key field is necessary to distinguish tuples that belong to a single tuple in the parent table.

A primary key comprising real-world attributes is a natural primary key. If no convenient natural key already exists, you may choose to introduce a surrogate primary key, i.e. an artificial attribute whose purpose is to uniquely identify tuples in a table. An institutional process must ensure the uniqueness and permanence of a surrogate key. For example, the U.S. government assigns every worker a surrogate attribute, the social security number, but the government must go to great lengths to ensure that this primary key is assigned exactly once by checking against another less convenient candidate key (the combination of name, parents' names, date and place of birth, etc.) Just like the SSN, well managed surrogate keys tend to get institutionalized and become natural.

4.3.4 Add non-key table fields

Non-key (or dependent) fields carry the useful information that is identified by the primary key. To add another non-key field to a table, use

```
ddj=addField(ddj,'datatype','comment'[,defaultValue]);
```

Field names must be sufficiently descriptive to communicate their role in the relation. Avoid using the same field name in different tables: For example, if both the `Sessions` and `Scans` tables have a comment field, give them different names, e.g. `sess_comment` and `scan_comment`.

Although not recommended, an optional default value can be specified for each non-key field. If the default value is `[]`, the field value is altogether optional (i.e. can be null). Only numeric and blob fields can be null. Missing field values will retrieved as empty cells (for blob fields) or NaNs (for numeric fields). Null values in optional fields may yield nonintuitive results in logical expressions and are best avoided: if a field does not apply to all the tuples in a table, it probably belongs in a separate table. For optional character string fields and enums, use the default value of `''` (empty string), which will result in the empty field instead of null for missing values.

4.3.5 Field datatypes

The following datatypes are supported by DataJoint:

enum one of several explicitly enumerated values specified as strings. Use this datatype instead of text strings to avoid spelling variations and to save storage space. For example, for anesthesia, the datatype could be `enum` with values `"urethane"` and `"isoflurane"`. Do not use enums in primary keys if you plan to change the enum values in the future.

date date as `'YYYY-MM-DD'`.

timestamp Date and time to the second. Also can be configured to timestamp data entries automatically.

char(N) a character string that takes exactly `N` characters of storage.

varchar(N) a text string of arbitrary length up to `N` characters.

decimal(N,F) a fixed-point number with `N` total decimal digits and `F` fractional digits. This datatype is well suited to represent numbers whose magnitude is well defined and does not warrant the use of floating-point representation..

decimal(N,F) unsigned same, but limited to nonnegative values.

float a single precision floating-point number. Takes 4 bytes. Single precision is sufficient for many measurements.

double a double precision floating-point number. Takes 8 bytes.

tinyint an 8-bit integer number, ranging from -128 to 127.

tinyint unsigned an 8-bit integer number, ranging from 0 to 255.

smallint a 16-bit integer number, ranging from -32,768 to 32,767.

smallint unsigned a 16-bit positive integer, ranging from 0 to 65,535.

mediumint a 23-bit integer number, ranging from -8,388,608 to 8,388,607.

mediumint unsigned a 24-bit positive integer, ranging from 0 to 16,777,216.

int a 32-bit integer number, ranging from -2,147,483,648 to 2,147,483,647.

int unsigned a 32-bit positive integer, ranging from 0 to 4,294,967,295.

longblob for MATLAB matrices, images, structures, and objects, up to 4 GiB in size. DataJoint also allows smaller blobs **mediumblob**, **blob**, and **tinyblob** to store up to 16 MiB, 64 KiB, and 256 B, respectively.

Use the smallest most restrictive datatype sufficient for your data.

4.3.6 Add reference fields

Reference fields are rarely necessary in DataJoint but may be helpful for some types of constraints. Reference fields are the extra primary key fields of another table that are added as non-key fields to the current declaration. Values in reference fields may be entered only if they exist in the referenced table.

Use the `ddj = addReference(ddj,TableName)` to add reference fields.

For example, the table `StimsOriPresentations` (subsection 9.4) is already uniquely identified for each tuple in its parent `Stims` by its extra primary key field `presnum`. To refer to tuples in a related table `StimsOriConditions`, the proper reference is added, which adds the field `cond_idx` to `StimsOriPresentations`. The same aim could have been achieved by making `StimsOriConditions` the parent of `StimsOriPresentations`, in which case the uniqueness of values in field `presnum` would not be enforced.

4.3.7 Omit the timestamp field

By default, `DeclareDJ` will add an automatic timestamp field, which will contain the date and time of the last update of each tuple. To omit the timestamp field, add

```
ddj=omitTimstamp(ddj);
```

4.3.8 Execute the declaration

Finally, to create the table in the database and to generate its MATLAB class in the current directory, execute

```
execute(ddj);
```

Declarations may be executed multiple times without the risk of overwriting existing tables or existing code. To modify an existing table, drop it first:

```
>> drop(Mice);  % drop table Mice from the database
```

4.4 Set the table comment

To add a brief description to the table, use the `DJ/setTableComment` method

```
setTableComment(Mice, 'Mouse information');
```

4.5 Viewing tables

You may start interacting with the new table immediately. To view the table attributes, use the `DJ/display` method (or simply omit the trailing semicolon after an object of the class):

```
>> display(Mice);
or
>> Mice
```

To view the current contents of the table, use the `DJ/disp` method or apply the apostrophe operator to an object of the class:

```
>> disp(Mice);
or
>> Mice'
```

Finally, to see the diagram of all or some of the tables in the your schema, use the `erd` command (subsection 9.2).

4.6 Reverse engineering existing tables

Use the `DJ/getDeclaration` method to get the `DeclareDJ` declaration of an existing table,

Use `DJ/exportSchema` to generate an m-file with declarations for all or some of the tables in the current schema.

4.7 Referential integrity

DataJoint supports only two types of data manipulations that preserve data dependencies: inserting tuples and deleting tuples from base relations. As long as only these operations are performed, DataJoint ensures that all data are matched correctly, all computations are up-to-date, and all inseparable groups of tuples are complete.

It may be useful to go around these constraints, which can be done through any other database interface (e.g. Navicat or the MySQL Workbench), in which case the user takes responsibility for the consistency of introduced changes.

The database will enforce referential integrity of the data across multiple tables: An attempt to insert a tuple into a dependent table before its matching tuples are present in all the parent tables will result in error. Conversely, a group of tuples cannot be deleted from a table if any one of them has matching tuples in any of the dependent tables.

4.8 Keep tables normalized

Under the relational model, each column in the table should contain the simplest datum as opposed to a complex structure to be parsed later. It may be tempting to place complex structures into a single blob field while the more appropriate course of action is to separate the structure into separate fields and even separate tables. However, some large objects such as images or traces are always manipulated as an atomic datum and should be kept together in a single field. In summary, if you plan to search by the value of an attribute or retrieve it separately from other attributes, it needs its own field.

Before learning the relational model, most programmers tend to lump too many things in one table similar to working with spreadsheets. Relations are not spreadsheets. Database normalization is the systematic process of dividing tables into separate simpler tables until they meet formal sets of criteria known as Normal Forms. Properly normalized tables store all information exactly once and relate it precisely to other information, precluding many anomalies arising in data manipulations. For extra credit, review the first three normal forms applicable

to DataJoint: http://dev.mysql.com/tech-resources/articles/intro-to-normalization.html, http://en.wikipedia.org/wiki/First_normal_form, http://en.wikipedia.org/wiki/Second_normal_form, and http://en.wikipedia.org/wiki/Third_normal_form.

The normal forms have been humorously summed up by this solemn oath: “Every attribute in a table must relate a fact about the primary key, the whole key, and nothing but the key — so help me (Edgar F.) Codd." If some attributes are not directly associated with the entire primary key, they must reside in a different table with a different primary key. As you design your first tables, you will quickly gain intuition as to which fields belong together in the same table. Table design, including the choice of the primary keys and table dependencies, will effectively communicate and enforce the rules and the structure of your data. As rules change, you can change the structure of your schema or design a new schema altogether and migrate your old data there.

Here are a few common signs indicating that a table may need to be split:

1. The same data needs to be entered or updated in more than one place, possibly in different forms. This indicates that the duplicated data must be moved to their separate table and referred to by other tables.

2. Some attributes do not apply to all tuples in a table. Your table appears to contain different kinds of facts, which is not allowed in a normalized design. You will need to split this table into two with different sets of attributes.

5 Entering data into manual tables

5.1 Entering manual data from MATLAB

In MATLAB, tuples may be represented as structures while relations may be represented as structure arrays. The `DJ` method `insert` inserts a structure into the table, e.g.:

```
insert(Mice,struct(...
    'mouse_id',122,...
    'mouse_dob','2011-02-11',...
    'mouse_strain','C57BL/6',...
    'mouse_sex','M'));
```

The order of fields in a tuple is not significant. Fields that have default values may be omitted. Multiple tuples may be inserted by providing a structure array with values. If a tuple with the same primary key already exists, `insert` will throw an error. Use the `inserti` method to quietly ignore duplicates. To change an existing tuple, you must first `delete` it and then re-`insert` it.

You may also enter data interactively using `DJ`'s `enter` method, e.g.:

```
>> enter(Mice);
```

This method opens a new form for each tuple, carries values over from a previously entered tuples, and warns the user if she attempts to enter a duplicate tuple. Furthermore, `enter` opens forms for dependent manual tables carrying over the primary keys from the root table.

5.2 Upload from .MAT files and download to .MAT files

`DJ` methods `download` and `downloadAll` save data from tables locally in .MAT files. The data may then be uploaded using methods `upload` and `uploadAll`. See these methods' help pages for details. These methods may also be used in a backup procedure.

5.3 Entering data through third-party database interfaces

Free versions of database interfaces (e.g. Navicat Lite, MySQL Workbench) allow editing tables as spreadsheets. This has the advantage of a more visual and interactive experience but lacks data protection mechanisms that are enforced by DataJoint's MATLAB-side utilities.

5.4 Delete

If you wish to delete all tuples from a table, use the `delete` command on the entire table.

```
>> delete(Sessions);
```

To delete a specific tuple, specify its primary key in a structure and use it to restrict the relation before deleting:

```
>> key = struct('mouse_id',2,'sessnum',3);
>> delete(Sessions(key));
```

The `delete` method allows restricting the deleted set of tuples using *restrict*, *semijoin*, and *antijoin* operators, (see subsection 6.2). For example, to delete all `Scans` associated with `Sessions` performed by operator Patrick that do not have `Stims` associated with them, the command would be

```
delete(Scans.*Sessions('operator="Patrick"')./Stims);
```

6 Data queries

6.1 Retrieving data: `fetch`, `fetch1`, and `fetchn`

To retrieve all tuples from relation `R` in the form of a structure array, use method `fetch`:

```
>> s = fetch(R);
```

By default, only the primary key attributes will be retrieved. List additional attributes to retrieve from the database or use the wild card `'*'` to retrieve all attributes of `R`:

```
>> s = fetch( Mice, 'mouse_dob' );  % retrieve the mouse_id and mouse_dob
>> s = fetch( Mice, '*' ); % retrieve all values from Mice
```

To extract individual attributes as separate arrays instead of a structure array as above, use methods `fetch1` and `fetchn`. Use `fetch1` to retrieve individual attributes from exactly one tuple. Use `fetchn` to retrieve attributes from any number of tuples, in which case string attributes and blobs will be returned as cell arrays.

```
>> dob=fetchn(Mice,'mouse_dob');
>> [id,dob]=fetchn( Mice,'mouse_id','mouse_dob');
```

Although most likely the retrieved values will be ordered by primary key values, do not assume any particular order or that two similar queries should return tuples ordered the same way. To correctly match values from different queries use the relational *join operator* (subsubsection 6.2.4).

Before fetching data from a relation, you may restrict the relation to the desired subset of tuples as explained in detail in subsubsection 6.2.1. For example, when the entire primary key is known, you can retrieve a field value from a specific tuple using `fetch1`:

```
>> key=struct('mouse_id',3);
>> dob=fetch1(Mice(key),'mouse_dob'); % get mouse date of birth for the key
```

Retrieving data by the primary key is the fastest and most efficient way to identify and retrieve data. Since primary key fields are propagated down the hierarchy of tables, the key of a given tuple in one table can be used to retrieve all and nothing but the related data from any other table in the hierarchy.

6.2 Relational operators

When data are organized relationally, all tuples in one relation represent discrete facts of the same form. Relations that share commonly named attributes may be combined to produce other, derived, facts. Relational algebra is a branch of mathematics that provides formal operators to derive new useful relations from base relations.

Here we will transition from database terminology to purely relational terminology: a *base relation* is the set of all tuples in its associated table; *relational operators* transform base relations into *derived relations*. Variables of class `DJ` are *relation variables* that may contain base relations or derived relations and support relational operators: *restrict, project, union, difference, semijoin, antijoin*, and *join*.

Relational operators are *algebraically closed*: their outputs are also relations that can be assigned to other relation variables and/or become operands in the next operator. None of the relational operators actually retrieve any data: they simply construct a query to the database. After the query is formed, the data may be retrieved into the MATLAB workspace using the `fetch` method (subsection 6.1).

6.2.1 Restrict

The *restrict* operator excludes some tuples from a relation based on a condition. DataJoint's `restrict` operator is rarely invoked directly because for base relations, restriction can be done more succinctly by the base relation's

constructor: `Scans(condition)` is equivalent to `restrict(Scans,condition)`. However, instantiated `DJ` objects can only be restricted using the `restrict` mehtod.

The first way to restrict a relation is to provide a tuple (in the form of a structure) to be matched:

```
key=struct('mouse_id',1001,'sessnum',3);
s=Sessions(key);  % all tuples of Sessions that match the key.
```

The same condition may be supplied as a string in the form of an SQL condition:

```
s = Sessions('mouse_id=1001 and sessnum=3');
```

The former syntax is more common because key attributes are often available in the form of a structure, but the latter syntax allows more flexible conditions by the use of comparison operators >,>=, <, <= and <>, algebraic operators, and various functions (see section 10). The two types of restriction conditions may be combined:

```
sessions = Sessions('sess_date>"2010-11-01"');  % all sessions after Nov 1, 2010
scans = Scans('z-surfz>300')  % all scans at depth=z-surfz greater than 300
cells = Cells(key,'green_contast>1.5');  % all high-contrast cells for the given key
```

Since tables within a given hierarchy share primary key fields, one can use the key obtained from one table to retrieve related information directly from any other table:

```
% iterate through all mice
for key = fetch(Mice)'  % iterate through Mice
   fprintf('Working on mouse %d\n', key.mouse_id);
   for key = fetch(Scans(key))'  %  iterate through Scans for current mouse
      fprintf('Working on session %d, scan %d\n', key.sessnum, key.scannum );
      % get all attributes of image masks that are classified as neurons
      s = fetch(Masks(key).*MaskSorts('mask_type="neuron"'),'*');
      % do other things here
      ...
   end
end
```

See the help page for `DJ/restrict` for further details.

6.2.2 Projection

Given relation `A`, the *projection* operator `B=pro(A,'attr1','attr2',..)` derives the new relation `B` containing the specified attributes from `A`. Primary key attributes are automatically included and cannot be excluded; this is an intentional constraint that prevents many logical errors. Thus the derivation `A=pro(A)` simply strips all non-key attributes from `A`.

The wildcard `'*'` matches all attributes.

An attribute may be renamed by listing it as `'old_name->new_name'`:

```
cells = pro(Cells,'*','cellnum->c1');  % rename field cellnum to c1
```

Computed attributes may be added by listing them as `'expression->new_name'`:

```
scans = pro(Scans,'*','z-surfz -> depth');  % add field 'depth' computed as z-surfz
```

Another form of the projection operator takes two relations as input and allows summary computations on groups of tuples in the second relation

```
P=pro(R,Q,'summary_expression->attr1',..).
```

The resulting relation `P` will have the same primary key as `R` and a new attribute `attr1` which contains a summary computation on matching groups of tuples in `Q`.

Aggregate functions are `count()`, `avg()`, `max()`, `min()`, `sum()`, `variance()`, and `std()`.

For example, `P=pro(P,Q,'*','count(*)->n')` keeps all the old attributes of `P` and adds the new computed attribute `n` containing the number of matching tuples in `Q`.

Another example, `P=pro(R,Q,'avg(pvalue<0.05)->frac_significant')` creates the relation `P` containing the fraction of tuples in `Q` whose attribute `pvalue` is less than 0.05 for every matching tuple in `R`.

All `fetch` operators perform a projection before retrieving the data and support all the functions provided by `DJ/pro`, including computed fields and aggregate computations. For example,

```
% retrieve orientations of tuned cells, convert from radians to degrees
prefOri=fetchn(OriTuningCells(key,'ori_pvalue<0.05'),'pref_ori/pi()*180->ori_degrees');

% plot the histogram of the fraction of tuned cells per scan
hist(fetchn( Scans, OriTuningCells, 'avg(ori_pvalue<0.05)->frac_tuned'));
```

For further information, see the help for `DJ/pro`, `DJ/fetch`, `DJ/fetch1`, and `DJ/fetchn`.

6.2.3 Semijoin and antijoin

As already discussed, tables in the schema are designed to have identically named fields (primary keys) in related tables, which allow matching tuples across different relations.

DataJoint's *semijoin* operator, sometimes called the *matching* operator, is implemented by the `times` operator: expression `A.*B` means "all tuples in `A` that have matching tuples in `B`." For example:

```
% all mice that have been used in imaging sessions performed by Cathryn
mice = Mice.*Sessions('operator="Cathryn"');

% all sessions since Nov 1, 2010 that have scans performed with the 20x lens
sessions = Sessions('sess_date>="2010-11-01"').*Scans('lens=20');
```

DataJoint's *antijoin* operator, sometimes called *not matching*, is implemented by the `rdivide` operator: expression `A./B` means "all tuples in `A` that do not have matching tuples in `B`." For example:

```
% all mice that have not been used in any imaging sessions
mice = Mice./Sessions;

% all mice that have not had any imaging sessions under urethane anesthesia
mice = Mice./Sessions('anesthesia="urethane"');

% all mice that have been imaged but not under urethane anesthesia
mice = Mice.*Sessions./Sessions('anesthesia="urethane"');
```

Semijoin and antijoin are non-commutative and non-associative.

For further information see the help pages for `DJ/mtimes` and `DJ/rdivide`.

6.2.4 Join

Also known as the *natural join*, the join operator is implemented by `DJ`'s `mtimes` operator: relation `A*B` has all attributes from both `A` and `B` from all possible matching tuples from `A` and `B`. The join operator may be used

to combine related information from two relations into a single relation and to generate all possible matching combinations of tuples from two relations. For example:

```
% add mouse information to the Sessions relation
sessions = Session*Mice;
```

The following example retrieves arrays of the image coordinates (`segx`,`segy`) and the matching preferred directions `pref` of the significantly tuned cells processed with `trace_opt=1` for a given `key`. The image coordinates are in `Masks` whereas the preferred directions are in `OriTuningCells`:

```
Neurons=Masks(key).*MaskSorts('mask_type="neuron"');
TunedNeurons=Neurons*OriTuningCells('trace_opt=1 and pvalue<0.05');
[x,y,pref]=fetchn(TunedNeurons,'segx','segy','pref');
```

The following uses SQL function `datediff` to compute the mouse postnatal day to select scans performed on young mice (pday≤ 24). Although learning full SQL syntax and concepts is not required to use DataJoint, some SQL functions may greatly simplify a query. See section 10 for several useful SQL functions.

```
YoungMice = restrict(Mice*Sessions,'datediff(sess_date,mouse_dob)<=24');
```

In some cases it may become necessary to rename some attributes in one or both of the joined relations to get the desired result. The following example plots the scatter plot of the values of `pref` from `OriTuningCells` with attribute `trace_opt=1` against those with `trace_opt=2`. To prevent attributes from being matched, they must first be renamed using the projection operator:

```
Tuned1=pro(OriTuningCells('trace_opt=1 and pvalue<0.05'),'trace_opt->t1','pref->p1');
Tuned2=pro(OriTuningCells('trace_opt=2 and pvalue<0.05'),'trace_opt->t2','pref->p2');
Pairs=Tuned1*Tuned2.*MasksSorts(key,'mask_type="neuron"'); %restrict to neurons match key
[p1,p2] = fetchn(Pairs,'p1','p1');
scatter(p1,p2);
```

For further information see the help page for `DJ/join`

6.2.5 Union and difference (presently excluded)

The union and difference operators are implemented by `DJ`'s operators + and −, respectively. Since relations are sets of tuples, the set union and the difference operators have their regular meanings. However, we chose not to include them in the present distribution of DataJoint for the following reasons:

The difference operator `A-B` is a special case of the antijoin operator `A./B` where both operands have the same attributes.

The need for the union operator is rare since the same result can almost always be achieved more efficiently by modifying the initial restrict operators.

7 Populating tables automatically

7.1 Setting the populate relation

Imported and computed tables can be populated automatically using their inherited method `populate`. To enable this function, the table's `DeclareDJ` declaration must specify the *populate relation* `P` as

```
ddj=setPopulateRelation(ddj,P);
```

For example, see the declaration for table `Stims` in subsection 9.4, which is populated based on `Stims`.

Then the command `populate(R)` will call `makeTuples(R,key)` sequentially for every unique key of its populate relation, where the custom callback method `makeTuples` implements specific computations.

In most cases, the populate relation is the join of the immediate parent tables. For example, the populate relation for `OriResponses` is

```
ddj=setPopulateRelation(ddj,CellTraceGroups*Stims);
```

However, sometimes the populate relation may need to be restricted to an appropriate subset of parent tuples. For example, to populate `ScansCorrected` only for those `Scans` acquired with the 16× lens or greater, the populate relation would be set as:

```
ddj=setPopulateRelation(ddj,Scans('lens>=16'));
```

`DeclareDJ` writes the populate relation into the class constructor, which you may modify at any time.

7.2 Filling out computed fields

For a given table `R`, the command `populate(R)` will sequentially call `makeTuples(R,key)` for each `key` in `R`'s populate relation that does not already have any matching tuples in `R`. The remaining task is to write the customized `makeTuples` callback function to fill out the missing non-key attributes. `DeclareDJ` automatically creates a template `makeTuples` listing the fields that must be computed.

`makeTuples` consists of the following sections:

1. fetch related data from parent tables or parents' parent tables for the given key.

   Important: `makeTuples` should only retrieve data from tables that are direct ancestors of the present table and their ancestors' subtables. Fetching from tables that are not above the current table in the hierarchy may result in outdated tuples when the referenced data are changed. If data are needed from other tables, the schema should be updated to place the desired data above the current table. This constraint is not currently enforced by DataJoint for the sake of performance.

2. copy the given key into a new tuple and compute the missing fields

3. insert the new tuple(s) into the table

The example below is for the `CellTraces` class, which produces filtered versions of calcium traces taken from `Masks`.

```
% CellTraces/makeTuples(this,key)
function makeTuples(this,key)

% get traces
[masknums,traces] = fetchn(Masks(key),'masknum','gtrace');
fps = fetch1( ScansCorrected(key),'fps');
```

```
% filter traces
traces = [traces{:}];  % traces are in columns
traces = filterTraces(traces,fps,fetch(TraceOpts(key),'*'));

% insert filtered traces
for icell=1:length(masknums)
    tuple = key;
    tuple.masknum = masknums(icell);
    tuple.trace = single(traces(:,icell));
    insert(this,tuple);
end
```

7.3 Subtables

A *subtable* is an imported or computed table that does not have a populate relation. As such, subtables cannot be populated by a call to their `populate` method. Instead, their `makeTuples` callback must be called directly from a parent's `makeTuples`. Thus subtable tuples are always inserted together with their parent tuples in one atomic transaction. In addition, DataJoint disables direct deletes from subtables so that, to delete from a subtable, one must delete from its parent table.

The entity relationship diagram (subsection 9.2) shows subtables in a lighter font weight.

Subtables enable many-to-one dependencies. Matching tuples in a table and its subtable can be considered an inseparable group. Another table can now refer to a parent tuple to establish a dependency on the matching group of tuples in the subtable. For example, a single image segmentation, represented by a tuple in `Segmentations` results in multiple image masks represented by tuples in `Masks`, which will appear all together with its parent tuple or not at all. The `Segmentations/makeTuples` then calls the `Masks/makeTuples` after inserting its own tuple:

```
% Segmentations/makeTuples
function makeTuples( this, key )
insert( this, key );
makeTuples( Masks, key );
makeTuples( MaskSorts, key );
```

As another example, a visual stimulus in `Stims` comprises multiple `StimsOriConditions` and `StimsOriPresentations`. An orientation response in `OriResponses` depends on all the stimulus conditions and stimulus presentations in a given visual stimulus in `Stims`. See subsection 9.2 and subsection 9.4 for details.

7.4 Transaction processing

Note that calls to `makeTuples` are performed within individual atomic transaction (**??**): Inserted tuples do not become visible to the rest of the world until `makeTuples` exits without errors. If `makeTuples` fails to complete, any inserts that it has already done into the table and its subtables are rolled back and never become visible to other processes. Conversely, changes made by other processes do not become visible to `makeTuples` while it is executing, making the world appear stable to a `makeTuples` call.

Transaction processing makes DataJoint transaction-safe: interrupted or failed processes do not leave incomplete data behind.

8 Distributed Jobs

Although `makeTuples` calls are executed within atomic transactions, they are non-blocking: parallel processes cannot tell whether a missing tuple is already being computed by another process. If the same populate calls are executed on multiple processors, they are likely to duplicate each other's efforts. A few extra steps are required to execute efficient distributed computations, as described in this section.

8.1 Defining the job callback

First, pick a manual table that will be used as the job table: jobs will be packaged into chunks for each key of this table. In the Vis2p schema, the `Scans` table is a good candidate for job management.

Define the method `job0(dj,key)` for the job table defining the tasks to be performed for every key. This will typically contain a list of populate calls for dependent tables. Below is an example of the `Scans/job0`

```
function job0(dj,key)
populate(ScansCorrected,key);
populate(Stims,key);
populate(Segmentations,key);
end
```

Test the job script first by calling it directly for a given key:

```
keys = fetch( Scans./Segmentations );
job0(Scans,keys(end));
```

8.2 Parallel execution with `runJob`

Now execute the following statement on any number of processors:

```
runJob(Scans)
```

This call will first create the job reservation table named `JobsFor<JobTable>`, e.g. `JobsForScans`, if it does not already exist. If a tuple is absent in the job reservation table, its matching job is available. Then `runJob` reserves the job by inserting a matching tuple with the `job_status` field set to `'reserved'`. When the job is completed, the tuple is replaced with `job_status` set to `'completed'` (for no errors) or `'error'` (if an error occurred).

You may then query the job reservation table to track job execution. To reset the jobs for a new execution, delete all or some of the tuples from the job reservation table.

Job classes may have multiple job methods: `job0`, `job1`, ... To execute the n^{th} job script, use `runJob(JobTable,n)`.

9 Appendix A: The Vis2p schema example

9.1 Scenario

The schema below represents the following simplified scenario: Our lab records neural activity data from mice. A recording session may include two-photon microscopy recordings with a variety of structural dyes and Ca^{2+}-sensitive dyes, optical imaging using intrinsic signals while the mouse views visual stimuli. The processing chain extracts the neural signals from the raw recordings and computes several attributes of neural activity such as orientation tuning.

9.2 Entity relationship diagram

The entity relationship diagram in Figure 2 depicts the parent/child relationships between tables used in examples throughout this tutorial. To produce similar diagrams for your schemas, use the `erd` method.

```
>> erd(Mice,'./core','./visual_stimuli');
```

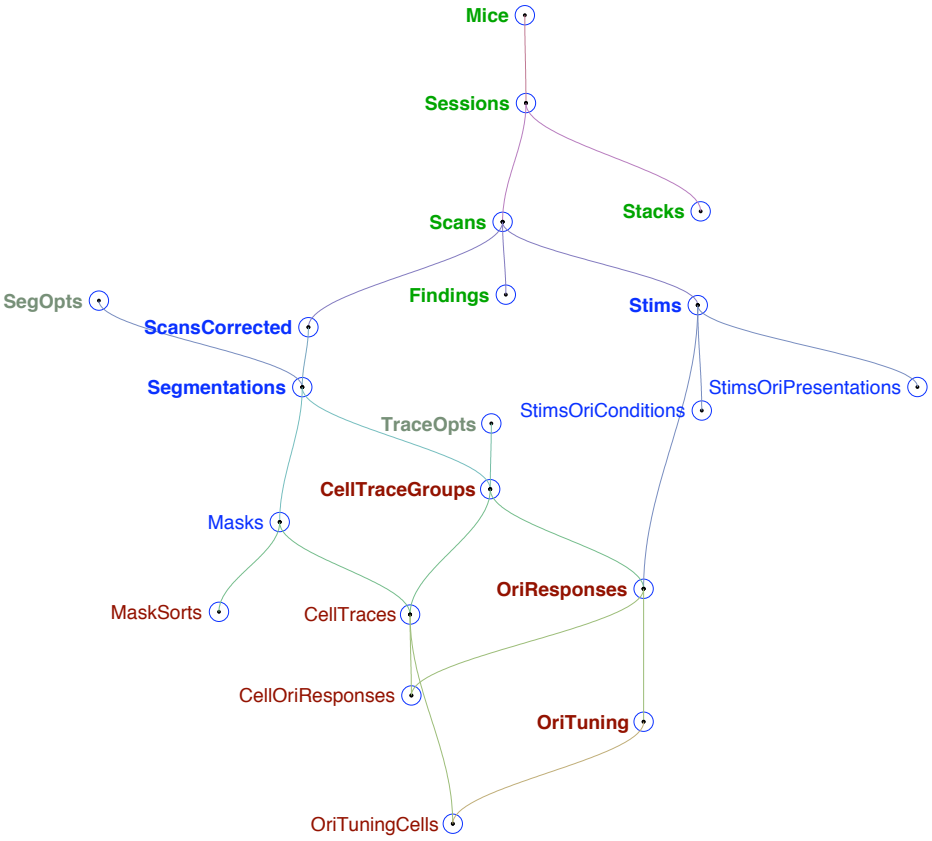

Figure 2: The entity relationship diagram for the vis2p schema used in examples in this tutorial. All dependencies are directed downward from a root table to dependent tables. Lookup tables are marked in gray, manual tables in green, imported in blue, and computed in red. Subtables (subsection 7.3) are indicated in light font while all other tables are in bold.

9.3 Declarations for Mice, Sessions, and Scans

```
%% Mice
ddj=DeclareDJ( Vis2p, 'Mice','manual' );
ddj=addKeyField(ddj,'mouse_id','smallint unsigned','unique mouse id (0-65535)');
```

```
ddj=addField(ddj,'mouse_strain','enum("C57BL6/J","agouti")','');
ddj=addField(ddj,'mouse_dob','date','mouse date of birth');
ddj=addField(ddj,'mouse_sex','enum("F","M","unknown")','');
ddj=addField(ddj,'mouse_notes','varchar(1023)','free-text info about the mouse','');

ddj=execute(ddj);
setTableComment(Mice,'Information about mice');


%% Sessions
dj=DeclareDJ(Vis2p,'Sessions','manual');
ddj=addParent(ddj,Mice);
ddj=addKeyField(ddj,'sessnum','tinyint unsigned','imaging session number for this mouse');
ddj=addField(ddj,'sess_date','date','recording session date');
ddj=addField(ddj,'operator','enum("Dimitri","Manolis","Jacob")', 'experimenter"s name');
ddj=addField(ddj,'anesthesia','enum("none","urethane","isoflurane","fentanyl")',...
   'anesthesia type per protocol');
ddj=addField(ddj,'dye',...
   'enum("none","OGB","OGB+SR101","OGB+Alexa","GCaMP3","OGB+SR101+calcein")', ...
   'fluorescent dye combination');
ddj=addField(ddj,'directory','varchar(512)', ...
   'the present location of the 2-photon data and other recordings');
ddj=addField(ddj,'sess_notes','varchar(1023)',...
   'free-form notes about the imaging session','');

dj=execute(ddj);
setTableComment(Sessions,'Recording sessions');


%% Scans
ddj=DeclareDJ(Vis2p,'Scans','manual');
ddj=addParent(ddj,Sessions);
ddj=addKeyField(ddj,'scannum','smallint unsigned','scan number within the session');

ddj=addField(ddj,'lens','tinyint unsigned','lens magnification');
ddj=addField(ddj,'mag','decimal(4,2) unsigned','scan magnification');
ddj=addField(ddj,'laser_wavelength','smallint unsigned','(nm)');
ddj=addField(ddj,'laser_power','smallint unsigned','mW to prep');
ddj=addField(ddj,'x','int',...
    '(um) objective manipulator x position (assumed same reference thru session)',[]);
ddj=addField(ddj,'y','int',...
   '(um) objective manipulator y position (assumed same reference thru session)',[]);
ddj=addField(ddj,'z','int',...
   '(um) objective manipulator z position (assumed same reference thru session)',[]);
ddj=addField(ddj,'surfz','int',...
   '(um) objective manipulator z position at pial surface',[]);
ddj=addField(ddj,'scan_notes','varchar(256)','optional free-form comments','');

dj=execute(ddj);
setTableComment(Scans,'Two-photon movie scans but not stacks');
```

9.4 Table declarations for visual stimuli

```
%% Stims
ddj=DeclareDJ(Vis2p,'Stims','imported');
ddj=addParent(ddj,Scans);
ddj=setPopulateRelation(ddj,Scans);
```

```
dj=execute(ddj);
setTableComment(Stims,'Visual stimuli and two-photon synchronization');


%% StimsOriConditions
ddj=DeclareDJ(Vis2p,'StimsOriConditions','imported');
ddj=addParent(ddj,Stims);
ddj=addKeyField(ddj,'cond_idx','smallint unsigned','the condition index');

ddj=addField(ddj,'direction','decimal(4,1) unsigned',...
   'degrees, grating direction. 0=up, the clockwise');
ddj=addField(ddj,'contrast','decimal(4,3)','grating contrast');
ddj=addField(ddj,'luminance','float','grating luminance');
ddj=addField(ddj,'spatial_freq','float','(cycles/degree) grating spatial frequency');
ddj=addField(ddj,'temp_freq','float','(Hz) grating temporal frequency');
ddj=omitTimestamp(ddj);

dj=execute(ddj);
setTableComment(StimsOriConditions,'');


%% StimsOriPresentations
ddj=DeclareDJ(Vis2p,'StimsOriPresentations','imported');
ddj=addParent(ddj,Stims);
ddj=addKeyField(ddj,'presnum','smallint unsigned','presentation number');

ddj=addReference(ddj,StimsOriConditions);
ddj=addField(ddj,'onset','double',...
   '(ms) onset time relative to the first frame timestamp');
ddj=addField(ddj,'duration','smallint unsigned',...
   '(ms) duration of the condition presentation');

dj=execute(ddj);
setTableComment(StimsOriPresentations,'');
```

10 Appendix B: Helpful SQL expression syntax

Conditions used in the restrict operator (subsubsection 6.2.1) are specified in SQL syntax. Users need to know only a few SQL operators and functions to take full advantage of the restrict operator. For additional information, consult the MySQL documentation available online.

10.1 comparison operators

The comparison operators =, <>, >, <, >=, and <= are applicable to numbers, dates, timestamps, and strings.

```
Sessions('sess_date>"2011-03-01"')  % all sessions recorded after March 1, 2011
```

10.2 logical operators

The logical operators `or`, `and`, and `not` are applied to results of boolean expressions.

```
Scans('lens>10 and mag>1.5')  % all scans with objective lenses >x10 and mag>1.5
```

10.3 detecting missing values

To detect missing values, one can use the operators `is null` and `is not null`:

```
ScansDetected('raster_correction is not null'); % scans in which raster correction was performed
```

10.4 arithmetic operators and functions

The arithmetic operators +, - (unitary and binary), *, /, % (modulo), and `div` (integer division) are applicable to results of numerical expressions.

In addition, mathematical functions can be applied to results of numeric expressions and their full list may be found at http://dev.mysql.com/doc/refman/5.5/en/mathematical-functions.html.

10.5 membership operators

The membership operators `in` and `not in` can simplify comparisons to a set of values:

```
% sessions with urethane or isoflurane anesthesia performed by anyone
% other than Dimitri or Manolis
Sessions(...
   'operator not in ("Dimitri","Manolis") and anesthesia in ("urethane","isoflurane")')
```

10.6 date and time functions

The full list of date and time functions may be found at http://dev.mysql.com/doc/refman/5.5/en/date-and-time-functior
html.

```
% sessions performed on mice 21 days and older
restrict(Sessions*Mice,'datediff(sess_date,mouse_dob)>=21')
```

10.7 aggregate functions

Aggregate functions can only be used to define new fields using the `pro` method invoked with two relations (subsubsection 6.2.2). They are described in detail at http://dev.mysql.com/doc/refman/5.5/en/group-by-functions.
html.

11 Appendix C: Internal conventions

DataJoint uses a number of internal conventions and constraints. Users do not need to be aware of these constraints for regular use.

11.1 Table names

Table names are converted from the CamelCase names of their classes to underscore-delimited compound names with an underscore preceding every capital letter.

Lookup table names are prefixed with the pound sign `#`; imported tables are preceded with a single underscore `_`; and computed tables are preceded with a double underscore `__`.

For example, the database name for the lookup table `TraceOpts` is `#trace_opts`, manual table `Scans` becomes `scans`, and computed table `TraceQuality` is `__trace_quality`.

11.2 Foreign keys

`DeclareDJ` sets foreign keys between tables to enforce the parent/child dependencies (see subsubsection 4.3.2) and reference dependencies (see subsubsection 4.3.6).

Although it is possible to distinguish the two types of dependencies based on the fields that they link, for faster execution, a naming convention is used: the foreign key names for reference dependencies must begin with `ref_`.

All foreign keys are defined with `UPDATE CASCADE`, although the need for cascading updates never comes up in normal use. Foreign keys are defined with `DELETE CASCADE` for subtables (see subsection 7.3) and with `DELETE RESTRICT` for all other tables.