Deduplication in Airtable

Note: Users who implemented the original version of these deduplication routines are **strongly** encouraged to update their bases to take advantage of the improvements found in this revision.

Table of Contents

Deduplication in Airtable ... 1
Quickstart ... 1
Introduction ... 2
Deduplication Mechanism ... 3
Overhead, Comparison with Initial Version ... 4
Airtable Deduplication Block ... 5
Deduplication Routines ... 5
Representative Process Flow ... 5
Match Key ... 7
Implementing Deduplication ... 9
Duplicate Detection ... 9
Handling of False Positives ... 12
Persistence and False Positives ... 16
Duplicated Bases and RECORD_ID() ... 18
Merger of Authentic Duplicates ... 20
Sample Bases ... 23
Trouble-shooting Common Problems ... 24
Video Introduction ... 24
Updates and Corrections ... 24
The Author ... 24
Appendix A: Defining {MatchKey} ... 25
Appendix B: Accented/Unaccented Characters ... 29

Quickstart

This document provides detailed information on version 3.0 of my Airtable deduplication routines: The rationale behind them; how they operate; how to add them to a base; potential optimizations; suggestions for implementation; and more – invaluable assistance for anyone wishing to modify or enhance them, but admittedly overkill for the Airtable user who merely wishes to incorporate the functionality in his or her own bases.

For those seeking a relatively painless way to add deduplication to an existing base, I suggest the following, stripped-down introduction to deduplication:

1. Watch the introductory video demonstrating each of the three deduplication 'modules.'

2. To add duplicate detection and marking, follow the steps outlined in **Duplicate Detection**.

1. Information on creating a {MatchKey} formula tailored to one's needs can be found in **Appendix A: Defining {MatchKey}**.

2. These are the core deduplication rule sets, mandatory for any implementation.

3. To add support for false positives – that is, the tagging of potential duplicates that prove not to represent actual dupes – follow the steps outlined in **Handling of False Positives**.

4. To add support for *persistent* tracking of false positives – for instance, to support recurring or ongoing deduplication of a base – follow the steps given in **Persistence and False Positives**.

5. To add support for merging duplicate records, follow the steps outlined in **Merger of Authentic Duplicates**.

Please keep in mind the rule sets presented here are cumulative: That is to say, all implementations require the routines outlined in Item 2; support for false positives require those in Items 2 and 3; persistent false positives require Items 2, 3, and 4; and to merge records requires all the routines from Items 2 through 5.

Most users will be able to copy-and-paste calculation and aggregation formulas from the sample bases; they should work with only minimal adjustments to existing bases.

Introduction

A frequently requested enhancement is for Airtable to support automatic deduplication of records. Exactly what that might mean is not always clear: Reading through the replies to the master thread in Airtable Community reveals a myriad interpretations of the term, ranging from simple enforcing of primary field uniqueness, to highly complex, variably weighted routines whose logic, oddly enough, often just happens to match the requester's specific use case….

Further complicating the matter is that, in the context of an Airtable base, true deduplication – that is, the removal of identical records – is often neither all that valuable nor what is actually desired. Frequently, what the user considers duplicate records are far from identical; in fact, they could differ widely in every field but one and still be a candidate for deduplication.

For instance, take the common task of merging mailing lists: One list contains an entry for 'John Uniquemiddlename Doe' at '123 Main Street, Centreville'; the other has a 'John Uniquemiddlename Doe' at '456 Central Ave., Middleburg.' By one measure, these records are just about as different as they could possibly be – yet, in that they provide comparable data about the same individual, they

are undeniably duplicate records. Not literally duplicates, perhaps, but certainly duplicates semantically.

What determines a duplicate, then, isn't always the byte-by-byte replication of an existing record; instead, sometimes it's merely the presence of the same or similar information in comparable data fields. In that case, we could try to deduplicate our mailing list based on matches against `{Name}`. That way, even if 'John Uniquemiddlename Doe' moves from Main to Central, the system will still know it has a duplicate record to delete.

–––but what if he *didn't* move from Main to Central, but instead moved from Central to Main? How would the program know which address is current and which to delete? Is it possible he might reside on Main Street but have office space on Central? Or what if some day he decides to give up trying to fit 'Uniquemiddlename' in the tiny little space allowed by most forms and from now on to go by 'John U. Doe' – and sometimes just 'John Doe?' Do you retain the oldest 'John Doe' record in the database and eliminate the rest? Or would that be the *newest* record you retained? Maybe you should just delete them all and wait for John to register again….

Clearly, aside from the relatively trivial – and frankly not all that common – need to eliminate redundant, thoroughly identical records, there are few situations in which fully automated deduplication could be tolerated. Instead, rather than duplicate *removal*, what most users appear to need is duplicate *identification*: a method by which probable duplicate entries are identified, tagged, and turned over for the user to vet for possible deletion or consolidation. More correctly, given how intractably it is joined with the base's purpose, structure, and data model, the method must be defined by the user. Accordingly, what he or she actually requires from the platform is not deduplication functionality but a *mechanism* supporting such functionality.

Deduplication Mechanism

This document and the accompanying example bases attempt to provide just such a mechanism. The functions supported include

- Identification of likely duplicates, based upon user-defined match fields.
    - Match fields may be calculated based upon one or more record fields.
    - Match fields may support case-sensitive or -insensitive matches.
    - Match fields may support more complex matching variations (common substitutions, matching of accented with unaccented characters, Soundex support, and the like).
    - Likely duplicates may be marked with dedicated field, Airtable record coloring.

- Marking of false positives — that is, of seeming duplicates that turn out not to be duplicates.
    - Ticking of checkbox 'unflags' all matching false positives.
    - If desired, the list of records marked as false positives can be made persistent so as not to require deduplication during subsequent passes.
- Merging of duplicate records.
    - Authentically duplicate records may be merged, rather than simply deleted.
    - Merge rules are user-defined based on a self join between master and duplicate records.

Overhead, Comparison with Initial Version

Those familiar with the original version of this document may recall it spent much time and emphasis on differentiating between duplicate *detection*, with the immediate generation of an alert when the record just entered duplicated one already in the base, and *deduplication* as a one-time cleanse of redundant entries from an existing base. At the time the initial version was written, this dichotomy seemed important because of the processing deduplication apparently required. However, since then two things have occurred: First, experience with deduplicating relatively large bases – five to ten times the size of the largest previously cleaned – revealed the overhead not to be nearly as daunting as originally thought. Second, I stumbled across Airtable's little-known and un[der]documented support for aggregation *formulas*, as opposed to aggregation *functions*, in rollup fields, which further reduced the operational impact. As a result, this revised version presents only a single method, which can be added to an existing base to clean it and then left in place for ongoing detection.

That said, with a large-enough base, even this latest version of the code can experience a noticeable lag time. For instance, with 10,000 records, the amount of time it takes to flag a false positive typically runs 4 to 5 seconds — at the edge of acceptable, perhaps, but most likely acceptable for most applications. However, using the routines presented here, *merging* a duplicate record with its corresponding master can take 45 seconds or more. Again, for a base requiring relatively few merges, this may still be reasonable; under other circumstances, it may be preferable initially to mark records to be merged and then perform the mergers in a single batch, as the delay to update after a single merger is the same as it is to update after an infinite number of mergers. (This technique is discussed in more detail later.)

Airtable Deduplication Block

Airtable currently has a Deduplication Block in beta test, intended to be used to cleanse a table. It has some very nice features; it also has some unsurprising limitations. No single module could possibly satisfy everyone's expectations for a deduplicator – and some of my requirements are admittedly niche. While I may not be able to replace all of my deduplication routines with the Block, I recently ran it against a 5,100-record client table with about 8% redundancy, and it performed admirably. More importantly, I also ran it against a multi-thousand-record table I'd previously cleaned using my own code, and it trapped several duplicates my routines had missed. I could easily imagine a deduplication work flow that makes use of both customized routines and the standard Block.

Deduplication Routines

Like seemingly everything I do, the deduplication routines are based upon an architectural trick I frequently use: Every record in the `[Main]` table is linked to a single record in a second table (here called `[DeDupe]`). This gives formula fields in the second table visibility across the entire main table, making possible cross-record calculations that are otherwise unattainable. In this instance such all-to-one connectivity is used to generate a long, concatenated string containing key data from all of the main table's records. Detecting duplicates, then, is as simple as searching for a match within this longer string.[1]

Representative Process Flow

The following table captures a representative process flow from the point of view of both the user, in the left-hand column, and the application itself, in the right. As one will see, user input is generally limited to ticking a checkbox, to indicate a flagged match is actually a false positive, or creating a linked record, either through the standard UI or by copy-and-pasting a value from one cell into another, in order to merge authentic duplicates. With each step the user takes, as documented on the left, the right-hand column explains what processing goes on behind the scenes to provide the necessary functionality.

1 Actually, rather than searching for a match, the routine compares the length of the `{MatchString}` with its length after all instances of the record's `{MatchKey}` are removed. If the difference is greater than the length of `{MatchKey}`, it means more than one instance was removed; hence, the record is considered a duplicate.

User Interaction	Base Processing
1. The table subject to deduplication is opened.	1. The table subject to deduplication is opened.
	.a For each record, a {MatchKey} is created, based upon rules determined by operational procedures and the data model.
	.b Every record in the main table is linked to a single record in a second table devoted to deduplication.
	.c Within [Dedupe], all {MatchKey}s from [Main] are rolled up into one concatenated {MatchString}.
2. Possible duplicate records are flagged for inspection.	2. Possible duplicate records are flagged for inspection.
	.a Within [Main], {Dupe?} compares the length of {MatchString} with the length of {MatchString} with all instances of {MatchKey} removed. If the difference is greater than the length of {MatchKey}, {MatchString} must have contained more than one instance of {MatchKey}. {Dupe?} is set to 1.
	.b (Optional.) In addition to {Dupe?} being set to 1, other fields may be set to indicate a duplicate, possibly using emoji, or Airtable conditional color may be used.
3. The user inspects possible duplicate records.	3. The user inspects possible duplicate records.
4. Records determined to be false positives are indicated as such by selecting the {Dupe OK} checkbox.	4. Records determined to be false positives are indicated as such by selecting the {Dupe OK} checkbox.
	.a When {Dupe OK} is ticked, {DupeOKMatch} is set to equal {MatchKey}.
	.b In [DeDupe], {DupeOKMatches} rolls up {Main::DupeOKMatch}.

	.c In [Main], {OKMatchRec} rolls up {DeDupe::DupeOKMatches} with an aggregation formula of
	`IF(`
	`    FIND(`
	`        {MatchKey},`
	`        values&''`
	`        )!=0,`
	`    '|'&RECORD_ID()`
	`    )`
	That is to say, if {MatchKey} is found within {DupeOKMatches}, {OKMatchRec} is set to '|'&RECORD_ID().

	.d In [DeDupe], {OKRecs} rolls up {Main::OKMatchRec}.

User Interaction	Base Processing
	.e In [DeDupe], {MatchOKRecs} is set to equal {HoldOKRecs}&{OKRecs}.

	.f In [Main], {TrueDupe} rolls up {DeDupe::MatchOKRecs} with an aggregation formula of
	`IF(`
	`    {Dupe?},`
	`    IF(`
	`        FIND(`
	`            '|'&RECORD_ID(),`
	`            values&''`
	`            )=0,`
	`        1`
	`        )`
	`    )`
	In other words, {DupeCooked} is set to 1 for records where {Dupe?} = 1 and RECORD_ID() is not found in {MatchOKRecs}.

	.g When the table has been deduplicated, {MatchOKRecs} can be copy-and-pasted into {HoldOKRecs}. This allows a persistent record of false positives to be maintained. Once this copy-and-paste has been performed, all {Dupe OK} checkboxes can be cleared.
5. Records determined to be actual duplicates are handled appropriately, either by deleting the redundant record or by merging duplicate records with a master record.	5. Records determined to be actual duplicates are handled appropriately, either by deleting the redundant record or by merging duplicate records with a master record.
6. (To use the merge mechanism built into the demonstration base.) In [Main], the user selects the + sign in {Link2Master} for the record to be merged. From the list of records presented, the user selects the appropriate master record.	6. (To use the merge mechanism built into the demonstration base.) In [Main], the user selects the + sign in {MasterID} for the record to be merged. From the list of records presented, the user selects the appropriate master record.
6. (Alternative method.) In [Main], the user selects {ID} for the master record, presses Ctrl-C to copy the value, selects {Link2Master} for the record to be merged, and presses Ctrl-V to paste the copied value.	6. (Alternative method.) In [Main], the user selects {ID} for the master record, presses Ctrl-C to copy the value, selects {MasterID} for the record to be merged, and presses Ctrl-V to paste the copied value.
	.a {MatchKey} is not created for records that have {MasterID} set. This in effect removes one instance of {MatchKey} from {MatchString}.

Table 1: User interaction vs base processing for deduplication

Match Key

The, ahem, key to the whole process is the match key — or the {MatchKey}. Earlier, I emphasized the distinction between *method* and *mechanism*. {MatchKey} is the embodiment of **method**. Typically, {MatchKey} is a calculated (formula) field that represents the crucial data that

differentiates one record from another — or, if identical to that of another record, indicates the records are likely duplicates. `{MatchKey}`'s composition should reflect the underlying base, the business processes surrounding its use, and an informed familiarity with common data trends and events. For example, in the example bases, the default format of `{First}` `{MI}` `{Last}` is provided to store individual's names. However, from experience I know the `{MI}` field is used inconsistently, and when it *is* used, frequently an incorrect initial is stored.

Therefore, in defining a `{MatchKey}` for this table, I began with `{First}&{Last}`. True, this will undoubtedly lead to false positives, as Robert Q Jones and Robert R Jones are flagged as a possible duplicate pair, but it will also capture Hubert H. Humphrey slumming as plain old Hubert Humphrey. Next, I wrapped the composite `FirstLast` with an `UPPER()` function, which yielded `FIRSTLAST` — and made deduplication case-insensitive. Finally, to prevent variation in punctuation, hyphenation, and the like from blocking matches, I wrapped the whole thing with a series of `SUBSTITUTE()` functions designed to smooth out insignificant differences. My initial `{MatchKey}` formula looked like this:

```
SUBSTITUTE(
    SUBSTITUTE(
        SUBSTITUTE(
            UPPER(
                {First}&{Last}
                ),
            ' ',''
            ),
        '_'','' 
        ),
    '.',''
    )
```

(For the final, 100-line formula, turn to **Appendix B**.)

However, something important to remember is one is not limited to a single match key. For instance, in the example bases there is a match key called, unsurprisingly, `{MatchKey}` — but there is also a match key called `{CompanyMatchKey}` built from `{Last}&{Company}`. (The example base started out as a working base for a client who had collected professional credentials from its contact list on multiple occasions and in multiple ways — scanned business cards, sign-up sheets at conferences, email response, screen-scraping — and `{Last}&{Company}` seemed the best shot at deduplicating the data.) A second (third, fourth…) match key links to the same `[DeDupe]` record as the first; however, each match key requires its own set of calculation and management fields as outlined in the processing half of **Table 1**.

Implementing Deduplication

There are three main components to the deduplication routines, one mandatory and two optional:

1. Duplicate detection and marking
2. Flagging of false positives and their removal from results
3. Merging of authentic duplicates and their removal from results

It is certainly possible to get by with only the first module, if one's business process allows. For instance, there are users who require textbook deduplication: That is, their data contain true duplicate records, entirely redundant information, of which one will be retained and the others deleted. Such a use case has no need of exception handling: Identify duplicate records; tick the checkbox to the left of all but one; and delete the checked records.

Especially for larger bases, there is an argument to be made for incorporating only those features one knows are needed, as each additional function adds complexity and processing delay. That said, for smaller bases — say, those in the 2,500- to 5,000-record range — one may want to include support for false positives, at least.

Duplicate Detection

To add duplicate detection and marking to a base:

1. **Create a new table called `[DeDupe]`.**
   For now, it should contain only a single field, `{Name}`, and no records.

2. **In the table to be deduplicated, create a `{MatchKey}` field.**
   This is a formula field that should encompass those data considered essential in differentiating one record from another.

   > Note: The field **must** begin and end with a non-alphanumeric character.
   > (I usually use '|', the vertical bar character.)

   Note that `{MatchKey}` does **not** need to be unique to a record, just 'unique-ish' enough to limit the number of false positives. (A person's name is a good example: Unless one is recording secretary of the John Smith Club, `{Name}` would likely be both intuitive and unique-enough to use as the key for a membership base.) In addition, the formula constructing `{MatchKey}` should reflect the business processes underlying the base. To return to an earlier example, if the formula to create `{PersonName}` is

`{First}&' '&{MI}&' '&{Last}`

and `{MI}` is known often to be missing or incorrect, then a good start for `{MatchKey}` might be `{First}&{Last}`. (For more information, see **Appendix A: Defining {MatchKey}**.)

3. **Create a linked-record field from `[Main]` to `[DeDupe]`.**
   For purposes of this tutorial, call it `{Link2DeDupe}`. Toggle off 'Allow linking to multiple records.'

4. **Link every record in the main table to a single record in `[DeDupe]`.**
   I discuss this technique in more detail in a post to Airtable Community. In brief, the easiest way to create these links is to mark-and-copy a period ('`.`') character from a text field, click the header of the `{Link2DeDupe}` field to mark the entire column, and press `Ctrl-V` to paste the value into all cells in the column. This will create a `[DeDupe]` record with the primary field set to '`.`' and linked to all records in `[Main]`.

   Note this establishes links for all existing `[Main]` records, but records later added to `[Main]` will need to be linked to `.` as well.

5. **In `[DeDupe]`, roll up all `{MatchKey}` values from `[Main]` into `{MatchString}`.**
   Use an aggregation function of `ARRAYJOIN(values,'')`; this will concatenate all `{MatchKey}` values with no separator character. As each `{MatchKey}` begins and ends with a non-alphanumeric character, this will result in `{MatchString}` having a value of

   `|MatchKey1||MatchKey2||MatchKey3|`...`|MatchKeyX|`

6. **In `[Main]`, create the rollup field `{Dupe?}`.**
   `{Dupe?}` links to `[DeDupe]` and rolls up `{MatchString}` using the following aggregation *formula*[2]

2 An aggregation **formula** – as opposed to an aggregation **function** – is a variation on the standard Airtable formula, entered in the aggregation function window of the rollup field configuration screen. It allows reference to the rolled-up field to be made through use of the keyword `values`. Only one remote field may be referenced in this way; if the calculation requires the value of another field from the linked table, that field must be indirectly referenced through a lookup or another rollup field in the current table.

A couple of caveats: First, variables accessed through `values` are returned as an array – which is probably not what you want. Most of the time, it will need to be cast to a string by appending `&''`: `values&''`. Second, the aggregation function window in the rollup-field configuration screen does not support the context-sensitive prompting found in Airtable's formula editor. This is likely less of an issue than it might first appear, as most complex formulas are composed offline and then pasted into Airtable.

```
IF(
    MatchKey,
    IF(
        (LEN(
            values&''
            )-LEN(
                SUBSTITUTE(
                    values&'',
                    MatchKey,''
                    )
                )
            )>LEN(
                MatchKey
                ),
        1
        )
    )
```

Essentially, this formula compares the length of `{MatchString}` and the length of `{MatchString}` with all instances of `{MatchKey}` removed. If the difference is greater than the length of `{MatchKey}`, then `{MatchString}` must have contained more than one instance: therefore, `{MatchKey}` is duplicated in the base.

7. **(Optional.) Further flag records for which `{MatchKey}` is a duplicate.**
   To identify duplicate records, one merely needs to create a view filtered to show only those records where `{Dupe?} = 1`. To further highlight such records visually, Airtable Record Coloring may be used, configured to set a color when `{Dupe?} = 1`; alternatively, a formula field may be created that displays an emoji using a formula such as

   ```
   IF({Dupe?},'❌')
   ```

And that's it: The table is ready for deduplication. Potential duplicates will now be identified, flagged, and possibly marked. To identify all potential dupes, create a view filtered to show records where `{Dupe?}` is equal to `1`.

Two things to keep in mind: First, it is up to the user to determine what to do with flagged records. With only the duplicate detection routines implemented, there are only two ways to remove a flagged record from the list of potential duplicates: Delete matching records until the record in question is the only one in the base with its particular `{MatchKey}`, or change one or more of `{MatchKey}`'s components in such a way as to make `{MatchKey}` unique.[3] As the first approach

3 The first version of these routines used just such a technique as a way to 'unflag' false positives; records marked as 'duplicate OK,' indicating it did not actually duplicate another in the base, had their `RECORD_ID()`s appended to `{MatchKey}`. This succeeded in removing it and matching records from the list, but it had the potential to introduce other errors later.

is valid only for certain types of data sets, and the second may cause duplicate records entered later to be overlooked, I created a second 'module,' if you will, containing definitions and routines to allow the flags indicating a potential dupe to be cleared in the case of a false positive without affecting the base's ability to detect later matches.

Handling of False Positives

To add false positive processing to a base that *already* supports duplicate detection:

1. **In `[Main]`, create the checkbox field `{Dupe OK}`.**

2. **In `[Main]`, create the formula field `{DupeOKMatch}`.**
   Configure it with the following formula: `IF({Dupe OK},{MatchKey})`.

3. **In `[DeDupe]`, roll up all `{DupeOKMatch}` values into `{DupeOKMatches}`.**
   `{DupeOKMatches}` links to `[Main]` and rolls up `{DupeOKMatch}` using the aggregation function `ARRAYJOIN(values,'')`.

4. **In `[Main]`, create the rollup field `{DupeOKRec}`.**
   `{DupeOKRec}` links to `[DeDupe]` and rolls up `{DupeOKMatches}` using the aggregation formula

   ```
   IF(
       AND(
           MatchKey,
           values!=''
           ),
       IF(
           FIND(
               {MatchKey},
               values&''
               )!=0,
           '|'&RECORD_ID()
           )
       )
   ```

   In other words, if the record's `{MatchKey}` can be found in `{DupeOKMatches}`, set the field to equal the record's `RECORD_ID()`. This ensures not only the record with `{Dupe OK}` checked but any record with the same `{MatchKey}` is processed as having been a false positive.

5. **In `[DeDupe]`, roll up all `{DupeOKRec}` values into `{DupeOKRecs}`.**
   `{DupeOKRecs}` links to `[Main]` and rolls up `{DupeOKRec}` using the aggregation

function `ARRAYJOIN(values,'')`. This results in a concatenated string containing the `RECORD_ID()` of all records tagged as false positives.

6. **In `[DeDupe]`, create `{OKRecsHold}` and `{OKRecsMatch}`.**
   `{OKRecsHold}` is a single-line text field, currently empty. `{OKRecsMatch}` is a formula field with the formula `{DupeOKRecs}&{OKRecsHold}`. As will shortly be explained in more detail, the reason for these three fields is to allow the `RECORD_ID()`s of records determined to be false positive to be stored persistently, while at the same time allowing the base to reflect in real time the changing status of newly tagged records.

7. **In `[Main]`, create the rollup field `{TrueDupe}`.**
   `{TrueDupe}` links to `[DeDupe]` and rolls up `{OKRecsMatch}` using the aggregation formula

   ```
   IF(
       {Dupe?},
       IF(
           FIND(
               '|'&RECORD_ID(),
               values&''
               )=0,
           1
           )
       )
   ```

   The first six steps were directed at identifying and tagging false positives; this seventh step flags those duplicates so far not determined to be false positives. (Essentially, if a record is identified as a probable dupe, and the user has *not* tagged it as a false positive, `{TrueDupe}` flags it as representing an actual duplication of data.)

The next three items support routines allowing for *persistent* tracking of false positives. As such, they are, strictly speaking, not essential for users who intend to deduplicate a base in a single pass and, afterwards, not worry about duplicates ever again. However, as is the case throughout this base, it is far easier to dike out unwanted functionality somewhere down the line than it would be to add it in later.

8. **In `[Main]`, create the rollup field `{LaterMatch}`.**
   `{LaterMatch}` links to `[DeDupe]` and rolls up `{OKRecsHold}` with the aggregation formula

   ```
   IF(
       AND(
   ```

```
        {TrueDupe},
        values,
        FIND(
            '|'&RECORD_ID(),
            values&''
            )=0
        ),
    MatchKey
    )
```

Essentially, this sets `{LaterMatch}` to equal `{MatchKey}` if the record is a duplicate *and* the record ID is not in the list of approved duplicates.

9. **In `[DeDupe]`, create the rollup field `{LaterMatchStr}`.**
   `{LaterMatchStr}` links to `[Main]` and rolls up `{LaterMatch}` using the aggregation formula `ARRAYJOIN(ARRAYUNIQUE(values),'')`.

10. **In `[Main]`, create the rollup field `{LaterDupe}`.**
    `{LaterDupe}` links to `[DeDupe]` and rolls up `{LaterMatchStr}` using the following aggregation formula:

```
IF(
    AND(
        {Dupe?},
        values
        ),
    IF(
        FIND(
            {MatchKey},
            values&''
            )!=0,
        1
        )
    )
```

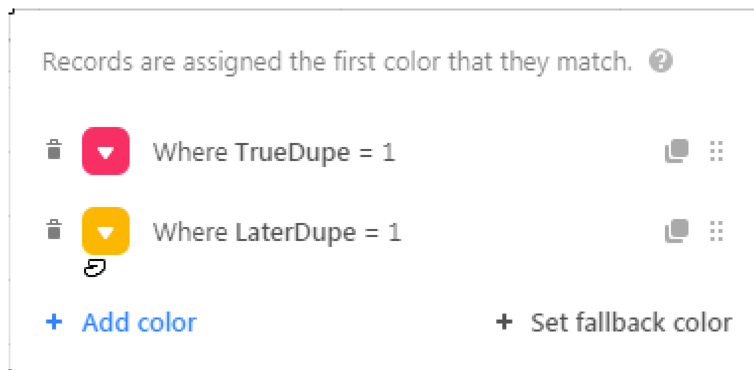

Illustration 1: Color coding to support false positives.

11. Modify views, fields, and color commands used to indicate potential duplications.

In my experience, the best way to configure a deduplication base with support for false positives is to filter views based on `{Dupe?}` while using `{TrueDupe}` and `{LaterDupe}` to control record coloring, as in **Illustration 1**, **above**. That way, one is initially shown all potential duplicates, color-flagged, as in **Illustration 2**, **below**. However, as one works through the base, tagging false positives, the color coding is cleared, indicating the records are no longer of concern — without the irritating, repetitive filtering of active records that would otherwise take place — as seen in **Illustration 3**.

Main | DeDupe

All Fields Grid · 15 hidden fields · 1 filter · Group · Sort · Color

	ID	Dupe OK	Name	MatchKey	Dupe?	DupeOKMatch	DupeOKRec	TrueDupe
1	04879		Anthony J Allen	|ANTHONYALLEN|	1			1
2	05971		Anthony M Allen	|ANTHONYALLEN|	1			1
3	09650		Joshua D Allen	|JOSHUAALLEN|	1			1
4	03175		Joshua J Allen	|JOSHUAALLEN|	1			1
5	09776		Julia R Allen	|JULIAALLEN|	1			1
6	09155		Julia W Allen	|JULIAALLEN|	1			1
7	06088		Helen G Anderson	|HELENANDERSON|	1			1
8	06575		Helen J Anderson	|HELENANDERSON|	1			1
13	08287		David A Bennett	|DAVIDBENNETT|	1			1
14	09208		David L Bennett	|DAVIDBENNETT|	1			1
15	08910		Carl R Brown	|CARLBROWN|	1			1
16	09915		Carl S Brown	|CARLBROWN|	1			1
17	01466		David A Brown	|DAVIDBROWN|	1			1

Illustration 2: Flagging of potential duplicates.

Main | DeDupe

All Fields Grid · 15 hidden fields · 1 filter · Group · Sort · Color

	ID	Dupe OK	Name	MatchKey	Dupe?	DupeOKMatch	DupeOKRec	TrueDupe
1	04879	✔	Anthony J Allen	|ANTHONYALLEN|	1	|ANTHONYALLEN|	|recxxDyx6HQdZXKoH	
2	05971		Anthony M Allen	|ANTHONYALLEN|	1		|recs2ck5mClVpcDp2	
3	09650	✔	Joshua D Allen	|JOSHUAALLEN|	1	|JOSHUAALLEN|	|recN5AGD8j2VQlySk	
4	03175		Joshua J Allen	|JOSHUAALLEN|	1		|rec7myRpDWZEJhefc	
5	09776	✔	Julia R Allen	|JULIAALLEN|	1	|JULIAALLEN|	|recNqgMbitybgRu...	
6	09155		Julia W Allen	|JULIAALLEN|	1		|recf9iqC6QlDYTs4t	
7	06088	✔	Helen G Anderson	|HELENANDERSON|	1	|HELENANDERSON|	|recgHYjbWMfLx4ksq	
8	06575		Helen J Anderson	|HELENANDERSON|	1		|recD4plsqR5wBZuDt	
13	08287	✔	David A Bennett	|DAVIDBENNETT|	1	|DAVIDBENNETT|	|recswXdV4H3NR2jYa	
14	09208		David L Bennett	|DAVIDBENNETT|	1		|rec2CbjbwACjTFzDP	
	08910	✔	Carl R Brown	|CARLBROWN|	1	|CARLBROWN|	|reczBCo7o3LCBRxf5	
16	09915		Carl S Brown	|CARLBROWN|	1		|recxvhgk9mVNtn74t	
17	01466		David A Brown	|DAVIDBROWN|	1			1

Illustration 3: Clearing false positives with {Dupe OK}.

As far as the user is concerned, tagging false positives is as simple as ticking the `{Dupe OK}` checkbox in *any* record containing the `{MatchKey}` in question. That is, if records for Robert T. Jones, Robert Q. Jones, and Robert Jones are all flagged as potential duplicates, selecting `{Dupe OK}` for any of the three will flag them all as false positives. Behind the scenes, though, the base seems almost frenetically busy, what with values being rolled up from `[Main]` to `[DeDupe]` back to `[Main]` back to `[DeDupe]` and, finally, back to `[Main]` once again. However, there **is** a method to the madness:

1. The initial ticking of `{Dupe OK}` identifies one of the records determined to be a false positive.

2. The tagged record surfaces its `{MatchKey}` to the ubiquitously linked `[DeDupe]` record.

3. From there, the `{MatchKey}` is visible to *every* record in the table. Each record compares its own `{MatchKey}` to the list of ones identified as belonging to false positives. If the record finds it *is* a false positive, it identifies itself by surfacing its unique `RECORD_ID()` to `[DeDupe]`.

4. The list of IDs newly associated with false positives is merged with the list of those previously associated and, once again, made visible to the entire table. Each record compares its own `RECORD_ID()` with the list of false positives. Those records found not to be included on that list are flagged as `{TrueDupe}`s.

It requires that many handoffs between `[Main]` and `[DeDupe]` for the progression from a single tagged record to definitive identification of all associated records to be made.

Persistence and False Positives

The reason the algorithm insists upon arriving at a unique, unambiguous, and immutable ID for each affected record has to do with the need for there to be a **persistent** recording of false positives that could span multiple attempts to deduplicate the base. For example, consider the largest of the example bases that accompany this document. Its 10,000 randomly generated records yielded 305 potential duplicates. Of those 305 flagged records, 17 proved to be actual duplicates needing to be merged with a master record; the remaining 288 were deemed false positives. Now imagine I were to add another 15,000 records to the base. Based on my initial experiments, I would expect to find roughly 765 possible dupes, 715 of which would turn out to be false positives.[4]

4 Actually, my estimates were low; I guess the random data generator I used must be optimized for data sets of somewhat fewer than 10,000 items. After another 15,000 records were added to the base, 1,458 were flagged as potential duplicates.

But wait: I already vetted 305 of those flagged duplicates. Must I go through them all again?

I could simply leave `{Dupe OK}` checked, which would keep the vetted false positives suppressed. Unfortunately, that would also suppress any new matching records, some of which might represent authentic duplicates. Clearing `{Dupe OK}` allows incoming dupes to be identified, but it also returns the previously cleared false positives to the pot, requiring them to be vetted again. Fortunately, the routines provide a mechanism that allows `{Dupe OK}` to be cleared, allowing newly found duplicates to be flagged while still inhibiting already vetted matches.

To enable persistent management of false positives:

1. **In `[DeDupe]`, copy `{OKRecsMatch}`, and paste its value into `{OKRecsHold}`.**
   As mentioned earlier, `{OKRecsMatch}` is a formula field consisting of `{OKRecsHold}` concatenated with `{DupeOKRecs}` – that is, of the list of known false positives with those newly identified. By copying its value into `{OKRecsHold}`, one creates an updated list of known false positives not dependent on `{Dupe OK}`.

2. **Once `{OKRecsHold}` is set, clear all checked `{Dupe OK}` fields.**
   The easiest way to accomplish this is to left-click on the `{Dupe OK}` header to mark the entire column and then press the `Delete` key; if prompted, confirm.

Even once `{Dupe OK}` is cleared, the vetted false positives do not return as flagged: The list of `RECORD_ID()`s in `{OKRecsHold}` keeps them suppressed. As new records are added to the

Main | DeDupe

All Fields Grid | 19 hidden fields | Filter | Group | Sort | Color

	ID	First	MI	Last	MatchKey	Dupe?	LaterMatch	TrueDupe	LaterDupe
1	00001	Melissa	F	Badger	|MELISSABADGER|				
2	00002	Martin	M	Martinez	|MARTINMARTINEZ|				
3	00003	Michael	R	Hudson	|MICHAELHUDSON|				
4	00004	James	L	Copeland	|JAMESCOPELAND|				
5	00005	Lindsay	T	Kimball	|LINDSAYKIMBALL|	1			1
6	00006	Jean	D	Garcia	|JEANGARCIA|				
7	00007	Marla	D	Fernandez	|MARLAFERNANDEZ|				
8	00008	Carole	K	Zurita	|CAROLEZURITA|				
9	00009	Norman	R	Manigault	|NORMANMANIGAULT|				
10	00010	William	M	Hardy	|WILLIAMHARDY|				
11	00011	Lindsay	P	Kimball	|LINDSAYKIMBALL|	1			1
12	00012	Thomas	M	Johnson	|THOMASJOHNSON|				
13	00013	Lindsay	Z	Kimball	|LINDSAYKIMBALL|	1	|LINDSAYKIMBALL|	1	1

Illustration 4: Persistent tracking of false positives.

base, duplicate detection continues. Should a record whose `{MatchKey}` matches that of a previously cleared false positive be added, the new duplicate is flagged as shown in **Illustration 4, above**. Note that Records 5 and 11 are previously discovered false positives tracked persistently by the base, and Record 13 is newly added. Using the coloring configuration shown in **Illustration 1**, the previously vetted records are highlighted with gold, the newly identified match in red.

Duplicated Bases and RECORD_ID()

`RECORD_ID()` was chosen as the identifier for persistent tracking of false positives because it is a unique value permanently associated with the record — unlike, say, a value derived from an autonumber field. Although the latter can be considered permanent and unchanging under normal circumstances, it *is* possible to regenerate an autonumber field, potentially renumbering every record in the base. `RECORD_ID()`, on the other hand, *is* permanently associated with a specific record: About the only to change it is to move to a different base.

——and therein lies a potential weakness of using `RECORD_ID()`: Should it ever become necessary to make a copy of the base — to share code or data with another user, to try out some new ideas without jeopardizing a production system, or simply to create a backup of critical data — as soon as the copy is opened, every false positive ever logged by the system will suddenly reappear. `{OKRecsHold}` may still be a list of records to ignore… but now it is a list of records found in a different base.

One possible solution to this problem would be to use a different unique identifier in place of `RECORD_ID()` — assuming such a creature exists in the base. Typically, this might be some sort of transaction-linked counter: invoice number, perhaps, badge scan number, Unix timestamp.[5] Another approach would be to convert the formula fields where `RECORD_ID()` surfaces to single-line text fields before the copy. (This still leaves the question of how best to handle any future deduplication needs….)

Finally, one can always replace the `RECORD_ID()`-driven key field with one not tied to the base's infrastructure — an autonumber, for instance — perform the copy, and repopulate the table with corresponding `RECORD_ID()`s from the new base. To do so, once the base has been updated for persistent false positives — that is, once `{OKRecsMatch}` has been copied into `{OKRecsHold}` and `{Dupe OK}` has been cleared —

5 But **not** the timestamp of the Airtable record, as that suffers the same susceptibility to base copies as `RECORD_ID()`.

1. **Create an autonumber field in `[Main]` called `{N}`.**

2. **Create a formula field called `{Main::AnbrID}`,** for 'autonumber ID,' with the formula `REPT('0',5-LEN({N}&''))&{N}`.

3. **Create a rollup field called `{Main::DupeOKID}`.**
   It should follow `{Link2DeDupe}` and roll up `{OKRecsMatch}` with this aggregation formula:

```
IF(
    values!='',
    IF(
        FIND(
            '|'&RECORD_ID(),
            values&''
            )!=0,
        '|'&AnbrID
        )
    )
```

4. **Create a rollup field called `{DeDupe::DupeOKIDs}`.**
   It should follow `{Link2Main}` and roll up `{DupeOKID}` with the aggregation function `ARRAYJOIN(values,'')`.

5. **Create a single-line text field called `{DeDupe::HoldOKIDs}`.**
   Copy (`Ctrl-C`) the value in `{DupeOKIDs}` and paste (`Ctrl-V`) it into `{HoldOKIDs}`.

6. **Duplicate the base.**

7. **Open the duplicated base.**
   (Note: All previously-suppressed false positives will be flagged as duplicates. This is to be expected.)

8. **In `[DeDupe]`, delete the value from `{OKRecsHold}`.**

9. **In `[Main]`, change the configuration of `{DupeOKRec}`.**
   It should now roll up `{HoldOKIDs}` with the aggregation formula

```
IF(
    values!='',
    IF(
        FIND(
            '|'&{AnbrID},
            values&''
```

```
            )!=0,
        '|'&RECORD_ID()
        )
    )
```

10. **In `[DeDupe]`, copy the value in `{OKRecsMatch}` and paste it into `{OKRecsHold}`.**

11. **In `[Main]`, return `{DupeOKRec}` to its original configuration.**

    It should roll up `{DupeOKMatches}` using the aggregation formula

```
IF(
    AND(
        MatchKey,
        values!=''
        ),
    IF(
        FIND(
            {MatchKey},
            values&''
            )!=0,
        '|'&RECORD_ID()
        )
    )
```

12. **Delete the no-longer-needed fields.**

    These include `{Main::DupeOKID}`, `{DeDupe::DupeOKIDs}`, and `{DeDupe::HoldOKIDs}`. `{Main::N}` and `{Main::AnbrID}` may be retained or deleted as desired. (It may be necessary to remove them from the original base, as well.)

Merger of Authentic Duplicates

Finally, some sort of mechanism is needed to support the management of records found to represent authentic duplicates – at least for those data sets where simply deleting redundant records is not an option. Often, such duplication is handled by merging data from the matching records, especially since each record may contain fields not present in the other(s). The Airtable Deduplication Block includes a powerful yet easy-to-use merge editor that greatly simplifies the task of combining data from multiple records. What these routines offer is, in comparison, more of a placeholder for a merger mechanism than a viable mechanism itself; still, it illustrates how one might integrate a more-robust merge function into the deduplication process so that merged records are permanently removed from the work flow.

> **Note:** The 'merge' routine presented here only establishes a primary–secondary relationship between a master record and one or more subsidiary (duplicate) records, preparatory to the user combining data as appropriate; it does ***not*** actually merge the data.

Unfortunately, as was mentioned earlier, record merging is by far the most processing-intensive function included in these routines. Each time duplicates are merged, an instance of the applicable `{MatchKey}` is removed from `{MatchKeyStr}`. This in turn causes ***every*** calculated field involved in these routines to be recalculated – a process that, in a medium-to-large base (i.e., one of 10,000 or more records), can take over a minute. However, as it takes essentially the same amount of time to regenerate `{MatchKeyStr}` and perform the recalculations that result following the modification of multiple `{MatchKey}`s as it does for a single one, it may be preferable to merge a table's affected records all at once instead of sequentially. The solution presented here supports just such an approach.

To add support for merging duplicates to a base:

1. **In `[Main]`, create a self-join linked-record field called `{MasterID}`.**
   This should link back to `[Main]`. Toggle off 'Allow linking to multiple records.'

2. **In main, create a single-line text field called `{HoldMasterID}`.**

3. **Wrap `{MatchKey}` with `IF(NOT({MasterID}),` ... `)`.**
   The bases provided in support of this document include merge support by default; however, as described in **Appendix A: Defining {MatchKey}**, one can configure `{MatchKey}` without it. If that is the case, to add merger support to the base, an explicit reference to `{MasterID}` must be added to the formula.

To link a duplicate record to the applicable master record, the user enters the master record's ID – that is, the value of its primary field – into the `{MasterID}` field of the subsidiary record. There are three different ways to create this link:

- The user may select the plus sign ('`+`') in the `{MasterID}` field of the subsidiary record. This will cause a list of all records in the table to be displayed. The user then searches or scrolls through the list to find the master record's entry and selects it.

- The user may copy-and-paste the master record's primary field into the subsidiary record's `{MasterID}` field. This is most easily accomplished by selecting the master record's primary field (that is, the left-most cell of the record), pressing `Ctrl-C` to copy the value,

selecting `{MasterID}` for the subsidiary record, and pressing `Ctrl-V` to paste the copied value.

- Given the significant processing overhead associated with merging records, as discussed **above**, it will often be preferable to log the master record IDs for all subsidiary records currently identified in order to process them in a single batch. To accomplish this:

    1. For each subsidiary record identified, the user copy-and-pastes the appropriate master record ID into the `{HoldMasterID}` field of the subsidiary record.
    2. When all subsidiary records have been accordingly updated, the user left-clicks on the `{HoldMasterID}` header to select the entire column.
    3. The user presses `Ctrl-C` to copy all values in the column.
    4. The user left-clicks on the `{MasterID}` header to select the entire column.
    5. Finally, the user presses `Ctrl-V` to paste all copied values into `{MasterID}`.

This last approach is far and away the recommended method to perform multiple mergers, as the amount of processing required to update a single `{MasterID}` field is nearly the same as that required to update the `{MasterID}` field of *all* records in the table.[6]

Illustration 5, **below**, provides a before-and-after view of record merger. Note `{MatchKey}` is not generated for a record where `{MasterID}` is not equal to `BLANK()`. This removes the record entirely from the deduplication process, possibly clearing `{Dupe?}` for the master record. (It is this elimination of `{MatchKey}` that makes this routine so processing-heavy, in that it also forces the recalculation of `{DeDupe::MatchString}`; in turn, the recalculation of `{MatchString}` triggers the recalculation of essentially every other formula or rollup field defined by the routines.)

6 Well, all-but-one; you'd never update *all* records.

Before

	ID	MasterID	HoldMasterID	First	MI	Last	MatchKey	Dupe?
1	00007			Marla	D	Fernandez	|MARLAFERNANDEZ|	
2	00008			Carole	K	Zurita	|CAROLEZURITA|	1
3	00009			Norman	R	Manigault	|NORMANMANIGAULT|	
4	00010			William	M	Hardy	|WILLIAMHARDY|	
5	00011			Lindsay	P	Kimball	|LINDSAYKIMBALL|	
6	00012			Thomas	M	Johnson	|THOMASJOHNSON|	
7	00013			Carole	K	Zurita	|CAROLEZURITA|	1

After

	ID	MasterID	HoldMasterID	First	MI	Last	MatchKey	Dupe?
1	00007			Marla	D	Fernandez	|MARLAFERNANDEZ|	
2	00008			Carole	K	Zurita	|CAROLEZURITA|	
3	00009			Norman	R	Manigault	|NORMANMANIGAULT|	
4	00010			William	M	Hardy	|WILLIAMHARDY|	
5	00011			Lindsay	P	Kimball	|LINDSAYKIMBALL|	
6	00012			Thomas	M	Johnson	|THOMASJOHNSON|	
7	00013	00008		Carole	K	Zurita		

Illustration 5: Merging records, before and after.

Sample Bases

Three sample bases accompany this document; they are identical except for the number of records each contains, with the larger two provided to allow prospective users to predict how responsive the routines might prove for their particular application.[7] To make use of any of these bases, open the shared read-only link and duplicate the base into your own workspace.

Please note the third, largest base requires at least a Pro-level subscription, as it contains 10,000 records. While all three bases make use of record coloring, also available only to Pro- and Enterprise-level subscribers, the normally hidden `{.}` field — so-named as to allow the column to be made as narrow as possible — uses emoji to provide similar color-coding.

Base	Link
14-Record Deduplication Base	https://airtable.com/shrMnI6K6OIKvLNDN
1,000-Record Deduplication Base	https://airtable.com/shr84Pr380qiVyeix
10,000-Record Deduplication Base	https://airtable.com/shrwhXaERZRqXkdGG

Table 2: Read-only shares for the demonstration and sizing bases

7 Sample data courtesy of the always-wonderful FakeNameGenerator, an invaluable assistance when developing and testing bases.

Trouble-shooting Common Problems

Newly added records never register as duplicate.

All records in the table to be deduplicated must be linked to the single record in the `[DeDupe]` table. (See Item **4** in the section on **Duplicate Detection**.)

Newly added records that match earlier-cleared false positives do not register as duplicate.

To identify and flag newly entered records, the `{Dupe OK}` field must be cleared. Normally this is only done *after* copying the value of `{DeDupe::OKRecsMatch}` into `{OKRecsHold}`. See the section on **Persistence and False Positives** for more information.

Clearing `{Dupe OK}` causes all false positives to be flagged again as duplicates.

The value of `{DeDupe::OKRecsMatch}` must be copied into `{DeDupe::OKRecsHold}` *before* clearing `{Dupe OK}`. See the section on **Persistence and False Positives** for more information.

Video Introduction

Currently, there is a rather long (22 minutes) introductory video available. Intended to accompany this document, the video discusses each of the component 'modules,' with an animation detailing each routine's process flow, followed by a narrated screen capture sequence illustrating how the functionality evinces itself within Airtable. The main link streams under HTML5 and may not work with all browsers or, especially, mobile devices. As an alternative, a two-part version is available on YouTube.

Full video: http://paladesigns.com/airtable/dedupe/

Alternative part 1: https://youtu.be/1IQBNKidE0k
Alternative part 2: https://youtu.be/6bHT3CLclAM

Updates and Corrections

The most-recent version of this document can always be found at http://paladesigns.com/airtable/dedupe.pdf.

The Author

W. Vann Hall
Pala Designs

As always, I can be reached through Airtable Community or at wvannhall@paladesigns.com.

Appendix A: Defining {MatchKey}

As mentioned earlier, an appropriately designed `{MatchKey}` is absolutely essential for deduplication properly to support and enhance one's business processes.

The steps involved in designing a well-rounded `{MatchKey}` are as follows:

1. **Identify those fields most useful in determining a record's identity.**
   That is, assuming each record represents one, and only one, thing, which field or combination of fields is most likely to be a unique designator of that thing? For some data types, the choice will be obvious: A serial number, a MAC address, a membership ID. Others, though, may require a little thinking – a person's name, for example. First and last? First, middle, and last? Middle or middle initial? Suffix? Title? Matronymic?

   Another concern is how 'fuzzy' a match can be. For keys assumed to be unique, the answer is, 'not at all,' while others may benefit from a little uncertainty. To pick on names, once again, will *James* always be *James* – or might he be *Jim*? How about *Jimmy*? Or *Jonathan*: *John*, *Jon*, *Johnny*, *Johnnie*, *Jonny*? *Jack*? In some cases, it might be best to use only part of a field – for instance, `LEFT({First Name},1)&{Last Name}` is one possible way around the Jim-and-Johnny problem.

2. **Decide whether the match should be case-sensitive.**
   Unless there is a pretty compelling reason to preserve case sensitivity, matches should be case *in*sensitive. Doing so prevents failing to detect a duplicate as the result of either stylistic differences ('duVal' versus 'DuVal') or tYpos.

   To do away with case sensitivity, wrap `{MatchKey}` in either an `UPPER()` or a `LOWER()` function. This should be the first transformation applied to the core data values, as it simplifies any subsequent ones.

3. **Use `SUBSTITUTE()` to improve S/N.**
   To paraphrase Wikipedia, in science and engineering, the signal-to-noise ratio (often abbreviated S/N or SNR) is a measure comparing the level of a desired signal to the level of background noise. In information theory, the term is often used metaphorically to indicate the level of difficulty experienced in isolating meaningful information from unimportant chatter. For instance, here the signal would be the significant information contained in the targeted fields that allows the user to determine a record's identity. Noise, on the other hand, is predominantly content-less information that introduces superficial variation while conveying little or no additional meaning. The amount of noise in a `{MatchKey}` can be

reduced by using nested `SUBSTITUTE()` statements either to eliminate noise content or to replace it with non-noisy alternative content.

In my experience, there are three general categories of noise:

1. **Punctuation or spacing.**
   Varying use of periods, hyphens, and other punctuation marks, along with inconsistent spacing, often interferes with duplicate detection. Replace such characters with the empty string, `''`, removing them from `{MatchKey}`:

```
SUBSTITUTE(
    SUBSTITUTE(
        SUBSTITUTE(
            SUBSTITUTE(
                SUBSTITUTE(
                    {KeyCore},
                    '.',
                    ''
                    ),
                '-',
                ''
                ),
            ' ',
            ''
            ),
        ',',
        ''
        ),
    '\'',
    ''
    )
```

2. **Accents and alternative characters.**
   Strictly speaking, accented characters are *not* content-less; however, depending on the base's data source, they might as well be. This is especially the case when some of the data originates from the US, given Americans' general unfamiliarity with accented characters and the lack of a standard way to enter such characters using a QWERTY keyboard.

   The solution here is to use `SUBSTITUTE()` to replace accented or regional characters in `{MatchKey}` with their unaccented or English equivalents. *Which* characters to replace depends upon the expected dataset. Clearly, a Spanish-language application would look to substitute a different group of characters than would a Finnish-language one, while a base intended for pan-European support would need to replace a far larger set of characters. In fact, given `SUBSTITUTE()`'s relatively low processing burden, it may prove advantageous to perform at least a minimal transformation of accented characters – covering, say, those found in

Western European languages – even for a base with a purely US scope.

The basic format for all such transformations is `SUBSTITUTE([Proto-MatchKey],'[Accented]','[Unaccented]')`. The code for accented character replacement should go outside – that is, should execute after – any calls to `UPPER()` or `LOWER()`; otherwise, separate routines will be needed to replace both upper- and lower-case accented characters.

Appendix B contains lists and code for a number of such substitutions.

3. **'Semantic' noise.** Finally is what I call 'semantic' noise: Context-dependent optional content — articles, abbreviations, helper words — that may cause two *semantically* identical records to be interpreted as unique. Perhaps the best example again comes from the task of deduplicating a mailing list, this time from the field indicating affiliated organization. Consider 'Coca-Cola,' 'The Coca-Cola Corporation,' 'Coca-Cola, Inc.,' and 'Coca-Cola Corp.' Four ways to say the same thing, with 'Coca-Cola' the signal surrounded by substantial noise. Depending on the industry and nationality involved, there may be a number of possible targets for elimination: 'The,' 'A,' 'Company,' 'Companies,' 'Co.,' 'Incorporated,' 'Corporation,' 'Corp,' 'Inc.,' 'LP,' 'LLP,' 'LLC,' 'Ltd.,' 'S.R.O.,' 'Berhad,' 'BHD,' 'SDN BHD,' 'SND BHD,' 'D.O.O.,' 'gmbh' — the list goes on and on.

   In addition, industry-specific lists may need to be 'de-noised' of words and abbreviations often, um, incorporated as part of a company name. An advertising-related list might need to be stripped of 'Marketing,' 'Mktg,' 'Group,' and 'Agency'; one dealing with automobile dealerships could stand to lose 'Motors,' 'Automobiles,' 'Cars,' and 'Motorcars.'[8] Finally, given the inevitable spill-over of graphic design into corporate naming conventions, for matching purposes it is probably advisable to eliminate such things as ' and ,' '&,' '+,' ''n,' and the like.

   Note the captured space characters in that last example. To prevent unintentional substitutions, the elimination of spaces described in Item **1**, **above**, should be withheld until after semantic cleaning, and individual words to remove should be demarcated by leading or trailing spaces. Otherwise, one runs the risk of 'Theodore Anderson Distinctive Bakery' becoming 'odore erson Disttive Bakery.'

A well-configured `{MatchKey}` should not only capture the assumptions inherent in the data model, it should also reflect operational expertise. To return to an earlier example, according to the data model an individual's name may be expected to consist of a first name, middle initial, and last

8 This represents a different class of semantic noise reduction than the first described, as these terms are not, strictly speaking, mere 'helper' words but components of the company name; e.g., 'The Johnson Marketing Group, Inc.'

name. At the same time, operational experience and institutional knowledge may suggest middle initials are often missing or incorrect. Were one to base `{MatchKey}` on the data model alone, the result could be an unacceptably high rate of false *negatives* — that is, duplicate records not recognized as such, thanks to a missing or invalid middle initial. Accordingly, rather than having the formula `{First}&{MI}&{Last}` at its core, `{MatchKey}` would be better served by starting with `{First}&{Last}`. True, doing so would undoubtedly generate a number of false positives, but the system is designed to handle these relatively painlessly.

Appendix B: Accented/Unaccented Characters

Strictly speaking, not all of the following should be considered accented characters; even 'characters with diacritical markings' isn't quite correct, as in some cases they represent unique characters in their own right, rather than a modified version of another character. And replacing a marked character with an unmarked onee does more than merely simplifying the word for an American audience: Often, it changes one word to another, or turns a word into a meaningless clump of letters.

But we don't care: The changes are not being wrought within the word itself but merely within the `{MatchKey}` used to identify potential matches. (Other methods of encoding text for comparison warp the original text more extremely; for instance, under Soundex the key for 'Catherine' is 'C365.') As with all instances of 'noise reduction,' the goal is to reduce the amount of permissible variation in the field to be compared in order to lower the number of false negatives generated — that is, contextually duplicate records not identified by the system.

Language	Accented Characters
Bosnian	Ć Č Đ Š Ž
Croatian	Ć Č Đ Š Ž
Czech	Á Č Ď É Ě Í Ň Ó Ř Š Ť Ú Ů Ý Ž
Dutch	À Á Ä Ç È É Ë Ì Í Ï Ĳ Ò Ó Ö Ù Ú Ü
French	À Â Ä Ç È É Ê Ë Ì Î Ï Ò Ô Ö Ù Û Ü
German	Ä Ö Ü ß
Icelandic	Á Å Ä Æ É Í Ó Ö Ú Ý Þ
Italian	À È É Ì Î Ò Ó Ù
Latvian	Ā Č Ē Ģ Ī Ķ Ļ Ņ Š Ū Ž
Polish	Ą Ć Ę Ł Ń Ó Ś Ż Ź
Portuguese	À Á Â Ã Ç É Ê Í Ó Ô Õ Ú
Romanian	Ă Â Î Ş Ș Ţ Ț
Serbian	Ć Č Đ Š Ž
Slovak	Á Ä Č Ď É Í Ĺ Ľ Ň Ó Ô Ŕ Š Ť Ú Ý Ž
Spanish	Á É Í Ñ Ó Ú Ü
Swedish	Á Ä Å É Í Ó Ö Ú

Table 3: Characters with diacritical marks, by language

The following somewhat daunting formula for `{MatchKey}` should satisfactorily 'de-noise' match fields in most major Western European languages: English, French, German, Spanish, Italian, and Portuguese. The routine performs substitutions to replace the following characters:

À Á Â Ä Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Ö ß Ù Ú Û Ü

Support for additional languages based on a Latin character set may be incorporated by providing accented:unaccented substitution routines for additional characters as shown in **Table 3**, **above**. (This was taken from the demo bases; as such, it uses `{First}&{Last}` as the basis for `{MatchKey}`, and will need to be adjusted accordingly, depending on the application.)

```
IF(
    AND(
        AND(
            {First},
            {Last}
            ),
        NOT(
            MasterID
            )
        ),

'|'&SUBSTITUTE(

SUBSTITUTE(
    SUBSTITUTE(
        SUBSTITUTE(
            SUBSTITUTE(
                SUBSTITUTE(
                    SUBSTITUTE(
                        SUBSTITUTE(
                            SUBSTITUTE(
                                SUBSTITUTE(
                                    SUBSTITUTE(
                                        SUBSTITUTE(
                                            SUBSTITUTE(
                                                SUBSTITUTE(
SUBSTITUTE(
    SUBSTITUTE(
        SUBSTITUTE(
            SUBSTITUTE(
                SUBSTITUTE(
                    SUBSTITUTE(
                        SUBSTITUTE(
                            SUBSTITUTE(
                                SUBSTITUTE(
                                    SUBSTITUTE(
SUBSTITUTE(
```

```
    SUBSTITUTE(
        SUBSTITUTE(
            UPPER(
                {First}&{Last}
                ),
            '.',''
            ),
        '\'',''
        ),
    ',',''
    ),

                                        'À','A'
                                        ),
                                    'Á','A'
                                    ),
                                'Â','A'
                                ),
                            'Ä','A'
                            ),
                        'Ç','C'
                        ),
                    'È','E'
                    ),
                'É','E'
                ),
            'Ê','E'
            ),
        'Ë','E'
        ),
    'Ì','I'
    ),

                                                    'Í','I'
                                                    ),
                                                'Î','I'
                                                ),
                                            'Ï','I'
                                            ),
                                        'Ñ','N'
                                        ),
                                    'Ò','O'
                                    ),
                                'Ó','O'
                                ),
                            'Ô','O'
                            ),
                        'Ö','O'
                        ),
                    'ß','SS'
                    ),
                'Ù','U'
                ),
            'Ú','U'
            ),
```

```
          'Û','U'
          ),
      'Ü','U'
      ),
      ' ','')&'|')
```