Summit '97

Normalization Is a Nice Theory

by David Adams & Dan Beckett

Content adapted from ***Programming 4th Dimension: The Ultimate Guide***, by David Adams & Dan Beckett, published by Foresight Technology, Inc.

Normalization Is a Nice Theory

Quick Look

Designing a normalized database structure is the first step when building a database that is meant to last. Normalization is a simple, common-sense, process that leads to flexible, efficient, maintainable database structures. We'll examine the major principles and objectives of normalization and *de*normalization, then take a look at some powerful optimization techniques that can break the rules of normalization.

What is Normalization?

Simply put, normalization is a formal process for determining which fields belong in which tables in a relational database. Normalization follows a set of rules worked out at the time relational databases were born. A normalized relational database provides several benefits:

- ❖ Elimination of redundant data storage.
- ❖ Close modeling of real world entities, processes, and their relationships.
- ❖ Structuring of data so that the model is flexible.

Normalization ensures that you get the benefits relational databases offer. Time spent learning about normalization will begin paying for itself immediately.

Why Do They Talk Like That?

Some people are intimidated by the *language* of normalization. Here is a quote from a classic text on relational database design:

A relation is in third normal form (3NF) if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.

C.J. Date
An Introduction to Database Systems

Huh? Relational database theory, and the principles of normalization, were first constructed by people intimately acquainted with set theory and predicate calculus. They wrote about databases for like-minded people. Because of this, people sometimes think that normalization is “hard”. Nothing could be more untrue. The principles of normalization are simple, common-sense ideas that are easy to apply. Here is another author’s description of the same principle:

A table should have a field that uniquely identifies each of its records, and each field in the table should describe the subject that the table represents.

Michael J. Hernandez
Database Design for Mere Mortals

That sounds pretty sensible. A table should have something that uniquely identifies each record, and each field in the record should be about the same thing. We can summarize the objectives of normalization even more simply:

Eliminate redundant fields.

Avoid merging tables.

You’ve probably intuitively followed many normalization principles all along. The purpose of formal normalization is to ensure that your common sense and intuition are applied consistently to the entire database design.

Design Versus Implementation

Designing a database structure and *implementing* a database structure are different tasks. When you design a structure it should be described without reference to the specific database tool you will use to implement the system, or what concessions you plan to make for performance reasons. These steps come later. After you’ve designed the database structure abstractly, then you implement it in a particular environment—4D in our case. Too often people new to database design combine design and implementation in one step. 4D makes this tempting because the structure editor is so easy to use. Implementing a structure without designing it quickly leads to flawed structures that are difficult and costly to modify. Design first, implement second, and you’ll finish faster and cheaper.

Normalized Design: Pros and Cons

We've implied that there are various advantages to producing a properly normalized design before you implement your system. Let's look at a detailed list of the pros and cons:

Pros of Normalizing	Cons of Normalizing
More efficient database structure. Better understanding of your data. More flexible database structure. Easier to maintain database structure. Few (if any) costly surprises down the road. Validates your common sense and intuition. Avoid redundant fields. Insure that distinct tables exist when necessary.	You can't start building the database before you know what the user needs.

We think that the pros outweigh the cons.

Terminology

There are a couple terms that are central to a discussion of normalization: “key” and “dependency”. These are probably familiar concepts to anyone who has built relational database systems, though they may not be using these words. We define and discuss them here as necessary background for the discussion of normal forms that follows.

Primary Key

The primary key is a fundamental concept in relational database design. It's an easy concept: each record should have something that identifies it uniquely. The primary key can be a single field, or a combination of fields. A table's primary key also serves as the basis of relationships with other tables. For example, it is typical to relate invoices to a unique customer ID, and employees to a unique department ID.

Note: 4D does not implicitly support multi-field primary keys, though multi-field keys are common in other client-server databases. The simplest way to implement a multi-field key in 4D is by maintaining an additional field that stores the concatenation of the components of your multi-field key into a single field. A concatenated key of this kind is easy to maintain using a 4D V6 trigger.

A primary key should be unique, mandatory, and permanent. A classic mistake people make when learning to create relational databases is to use a volatile field as the primary key. For example, consider this table:

```
[Companies]
Company Name
Address
```

Company Name is an obvious candidate for the primary key. Yet, this is a bad idea, even if the Company Name is unique. What happens when the name changes after a merger? Not only do you have to change this record, you have to update every single related record since the key has changed.

Another common mistake is to select a field that is *usually* unique and unchanging. Consider this small table:

```
[People]
Social Security Number
First Name
Last Name
Date of birth
```

In the United States all workers have a Social Security Number that uniquely identifies them for tax purposes. Or does it? As it turns out, not everyone has a Social Security Number, some people's Social Security Numbers change, and some people have more than one. This is an appealing but untrustworthy key.

The correct way to build a primary key is with a unique and unchanging value. 4D's **Sequence number** function, or a unique ID generating function of your own, is the easiest way to produce synthetic primary key values.

Functional Dependency

Closely tied to the notion of a key is a special normalization concept called *functional dependence* or *functional dependency*. The second and third normal forms verify that your functional dependencies are correct. So what is a "functional dependency"? It describes how one field (or combination of fields) determines another field. Consider an example:

```
[ZIP Codes]
ZIP Code
City
County
State Abbreviation
State Name
```

ZIP Code is a unique 5-digit key. What makes it a key? It is a key because it determines the other fields. For each ZIP Code there is a single city, county, and state abbreviation. These fields are functionally dependent on the ZIP Code field. In other words, they belong with this key. Look at the last two fields, State Abbreviation and State Name. State Abbreviation determines State Name, in other words, State Name is functionally dependent on State Abbreviation. State Abbreviation is acting like a key for the State Name field. Ah ha! State Abbreviation is a key, so it belongs in another table. As we'll see, the third normal form tells us to create a new States table and move State Name into it.

Normal Forms

The principles of normalization are described in a series of progressively stricter "normal forms". First normal form (1NF) is the easiest to satisfy, second normal form (2NF), more difficult, and so on. There are 5 or 6 normal forms, depending on who you read. It is convenient to talk about the normal forms by their traditional names, since this terminology is ubiquitous in the relational database industry. It is, however, possible to approach normalization *without* using this language. For example, Michael Hernandez's helpful ***Database Design for Mere Mortals*** uses plain language. Whatever terminology you use, the most important thing is that you go through the process.

Note: We advise learning the traditional terms to simplify communicating with other database designers. 4D and ACI do not use this terminology, but nearly all other database environments and programmers do.

First Normal Form (1NF)

The first normal form is easy to understand and apply:

A table is in first normal form if it contains no repeating groups.

What is a repeating group, and why are they bad? When you have more than one field storing the same kind of information in a single table, you have a repeating group. Repeating groups are the right way to build a spreadsheet, the only way to build a flat-file database, and the *wrong* way to build a relational database. Here is a common example of a repeating group:

```
[Customers]
Customer ID
Customer Name
Contact Name 1
Contact Name 2
Contact Name 3
```

What's wrong with this approach? Well, what happens when you have a fourth contact? You have to add a new field, modify your forms, and rebuild your routines. What happens when you want to query or report based on all contacts across all customers? It takes a lot of custom code, and may prove too difficult in practice. The structure we've just shown makes perfect sense in a spreadsheet, but almost no sense in a relational database. All of the difficulties we've described are resolved by moving contacts into a related table.

```
[Customers]
Customer ID
Customer Name

[Contacts]
Customer ID (this field relates [Contacts] and [Customers])
Contact ID
Contact Name
```

Second Normal Form (2NF)

The second normal form helps identify when you've combined two tables into one. Second normal form depends on the concepts of the primary key, and functional dependency. The second normal form is:

A relation is in second normal form (2NF) if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key.

C.J. Date
An Introduction to Database Systems

In other words, your table is in 2NF if:

1) It doesn't have any repeating groups.
2) Each of the fields that isn't a part of the key is functionally dependent on the entire key.

If a single-field key is used, a 1NF table is already in 2NF.

Third Normal Form (3NF)

Third normal form performs an additional level of verification that you have not combined tables. Here are two different definitions of the third normal form:

> *A table should have a field that uniquely identifies each of its records, and each field in the table should describe the subject that the table represents.*
>
> *Michael J. Hernandez*
> ***Database Design for Mere Mortals***

> *To test whether a 2NF table is also in 3NF, we ask, "Are any of the non-key columns dependent on any other non-key columns?*
>
> *Chris Gane*
> ***Computer Aided Software Engineering***

When designing a database it is easy enough to accidentally combine information that belongs in different tables. In the ZIP Code example mentioned above, the ZIP Code table included the State Abbreviation and the State Name. The State Name is determined by the State Abbreviation, so the third normal form reminds you to move this field into a new table. Here's how these tables should be set up:

```
[ZIP Codes]
ZIP Code
City
County
State Abbreviation


[States]
State Abbreviation
State Name
```

Higher Normal Forms

There are several higher normal forms, including 4NF, 5NF, BCNF, and PJ/NF. We'll leave our discussion at 3NF, which is adequate for most practical needs. If you are interested in higher normal forms, consult a book like Date's ***An Introduction to Database Systems***.

Learning More About Normalization

ACI University in the United States offers a course on designing relational databases that covers normalization in detail. If you cannot attend a course, here are some suggestions for books that treat these concepts in greater detail.

Michael J. Hernandez
Database Design for Mere Morals
A Hands-On Guide to Relational Database Design
Addison Wesley
1997
ISBN 0-201-69471-9

We recommend Michael Hernandez's very approachable book on the database design process. (And at $27.95 you can't go wrong.) Unfortunately, he does not correlate his discussion with traditional normalization terminology.

C. J. Date
An Introduction to Database Systems, Volume I
Fifth Edition
Addison-Wesley Systems Programming Series
1990
ISBN: 0-201-51381-1

This is the classic textbook on database systems. It is widely available in libraries and technical bookstores. It's worth checking out of a library for reference, but if it makes your eyes cross put it down.

Chris Gane
Computer-Aided Software Engineering
The Methodologies, The Products, and The Future
Prentice Hall
1990
ISBN: 0-13-176231-1

Gane, one of the pioneers of RAD (Rapid Application Development), is a widely published author on the benefits of CASE (Computer Aided Software Engineering). His discussions of database normalization are at an intermediate level.

Denormalization

Denormalization is the process of modifying a perfectly normalized database design for performance reasons. Denormalization is a natural and necessary part of database design, but must follow proper normalization. Here are a few words from Date on denormalization:

The general idea of normalization...is that the database designer should aim for relations in the "ultimate" normal form (5NF). However, this recommendation should not be construed as law. Sometimes there are good reasons for flouting the principles of normalization.... The only hard requirement is that relations be in at least first normal form. Indeed, this is as good a place as any to make the point that database design can be an extremely complex task.... Normalization theory is a useful aid in the process, but it is not a panacea; anyone designing a database is certainly advised to be familiar with the basic techniques of normalization...but we do not mean to suggest that the design should necessarily be based on normalization principles alone.

Date, pages 528-529

Deliberate denormalization is commonplace when you're optimizing performance. If you continuously draw data from a related table, it may make sense to duplicate the data redundantly. Denormalization always makes your system potentially less efficient and flexible, so denormalize as needed, but not frivolously.

Summary Data

There are techniques for improving performance that involve storing redundant or calculated data. Some of these techniques break the rules of normalization, other do not. Sometimes real world requirements justify breaking the rules. Intelligently and consciously breaking the rules of normalization for performance purposes is an accepted practice, and should only be done when the benefits of the change justify breaking the rule.

Let's examine some powerful summary data techniques:

❖ Compound Fields

❖ Summary Fields

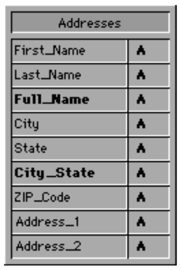
❖ Summary Tables

Compound Fields

A compound field is a field whose value is the combination of two or more fields in the same record. Compound fields optimize certain 4D database operations significantly, including queries, sorts, reports, and data display. The cost of using compound fields is the space they occupy and the code needed to maintain them. (Compound fields typically violate 2NF or 3NF.)

Save Combined Values

Combining two or more fields into a single field is the simplest application of the time-space trade-off. For example, if your database has a table with addresses including city and state, you can create a compound field (call it City_State) that is made up of the concatenation of the city and state fields. Sorts and queries on City_State are much faster than the same sort or query using the two source fields—sometimes even *40 times* faster.



Addresses	
First_Name	A
Last_Name	A
Full_Name	A
City	A
State	A
City_State	A
ZIP_Code	A
Address_1	A
Address_2	A

City_State stores the city and state in one field.

The downside of compound fields for the developer is that you have to write code to make sure that the City_State field is updated whenever either the city or the state field value changes. This is not difficult to do, but it is important that there are no “leaks”, or situations where the source data changes and, through some oversight, the compound field value is not updated.

Conditions for Building Compound Fields

Compound fields optimize situations that meet any of these conditions:

- ❖ The database uses multiple fields in a sequential operation.
- ❖ The database combines two or more fields routinely for any purpose.
- ❖ The database requires a compound key.

Let’s look at each of these situations in detail.

Using Multiple Fields in a Sequential Operation

Sorting on two indexed fields takes about twenty times longer than sorting on one indexed field. This is a case where you can give your system a huge speed improvement by using a compound field. If, for example, you routinely sort on state and city, you can improve sort speeds dramatically by creating an indexed, compound field that contains the concatenated state and city field values, and sort on this combination.

THE SOURCE FIELDS ARE COMBINED ROUTINELY

If you have fields that are routinely combined on forms, reports and labels, you can optimize these operations by pre-combining the fields; then the combined values are displayed without further calculation. This makes these operations noticeably faster. An additional benefit is that combined values makes it easier for end users to construct reports and labels with the information they need.

COMPOUND KEYS

A compound key ensures uniqueness based on a combination of two or more fields. An example of a compound key is the unique combination of an employee's identification number, first name, and last name. In 4D, ensuring that these values are unique in combination requires a programmed multiple-field query or a compound field. Once you have created a compound field to store the key, you can use 4D's built-in uniqueness-verification feature and save the time spent querying. Or you can query on the compound field and find the results more quickly than with an equivalent multi-field query.

When to Create Compound Fields

The three cases discussed here show obvious examples of where compound fields improve speed. Compound fields can also provide a significant improvement in ease of use that justifies their use in less obvious examples. Your users can easily query, sort, and report on compound fields. They don't have to learn how to combine these values correctly in each of 4D's editors, and they don't have to ask you to do this for them. Ease of use for end users almost always counts in favor of this technique even when the time savings are small.

Summary Fields

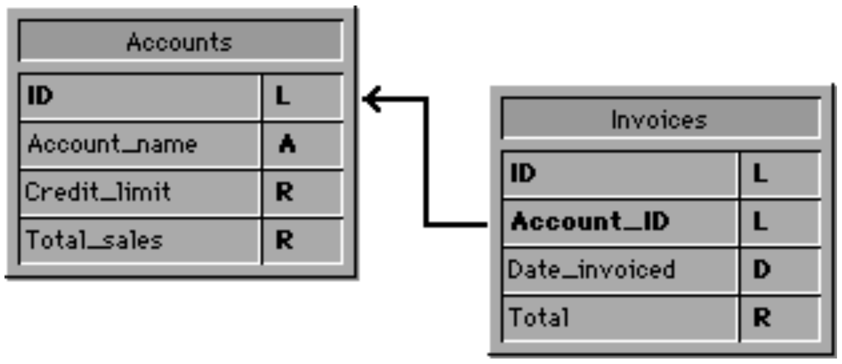
A summary field is a field in a one table record whose value is based on data in related-many table records. Summary fields eliminate repetitive and time-consuming cross-table calculations and make calculated results directly available for end-user queries, sorts, and reports without new programming. One-table fields that summarize values in multiple related records are a powerful optimization tool. Imagine tracking invoices without maintaining the invoice total!

Summary fields like this do not violate the rules of normalization. Normalization is often misconceived as forbidding the storage of calculated values, leading people to avoid appropriate summary fields.

There are two costs to consider when contemplating using a summary field: the coding time required to maintain accurate data and the space required to store the summary field.

Example

Saving summary account data is such a common practice that it is easy not to notice that the underlying principle is using a summary field. Consider the following table structure:



A very simple accounting system structure.

The Total_sales field in an Accounts record summarizes values found in one or more invoice records. You don't *need* the summary field in Accounts since the value can be calculated at any point through inspection of the Invoices table. But, by saving this critical summary data, you obtain the following advantages:

- ❖ A user does not need to wait for an account's total to be calculated.
- ❖ 4D's built in editors—Quick Report, Order by, Query, and Graph—can use this data directly without performing any special calculations.
- ❖ Reports print quickly because there are no calculations performed.
- ❖ You can export the results to another program for analysis.

What to Summarize

Summary fields work because you, as a designer, had the foresight to create a special field to optimize a particular operation or set of operations. Users may, from time to time, come to you with a new requirement that is not already optimized. This does not mean that your summary system is not effective, it simply means you need to consider expanding it. When considering what to optimize using summary fields, choose operations that users frequently need, complain about, or neglect because they are too slow. Because summary fields require extra space and maintenance, you should use them *only* when there is a definite user benefit.

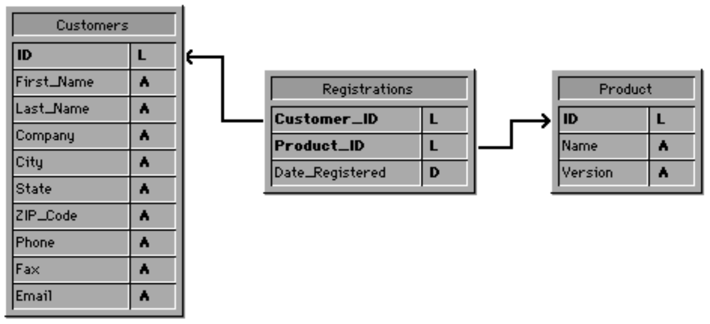
Summary Tables

A summary table is a table whose records summarize large amounts of related data or the results of a series of calculations. The entire table is maintained to optimize reporting, querying, and generating cross-table selections. Summary tables contain derived data from multiple records and do not necessarily violate the rules of normalization. People often overlook summary tables based on the misconception that derived data is necessarily denormalized.

Let's discusses the use of summary tables by examining a detailed example.

The ACME Software Company*

Consider a problem and corresponding database solution that uses a summary table. The ACME Software Company uses a 4D Server database to keep track of customer product registrations. Here is the table structure:



ACME's registration system structure

** Any resemblance between ACME Software Company and persons or companies living or dead is purely coincidental.*

ACME uses this system to generate mailings, fulfill upgrades, and track registration trends. Each customer can have many registrations, and each product can have many registrations. In this situation you can easily end up with record counts like these:

Original Record Counts

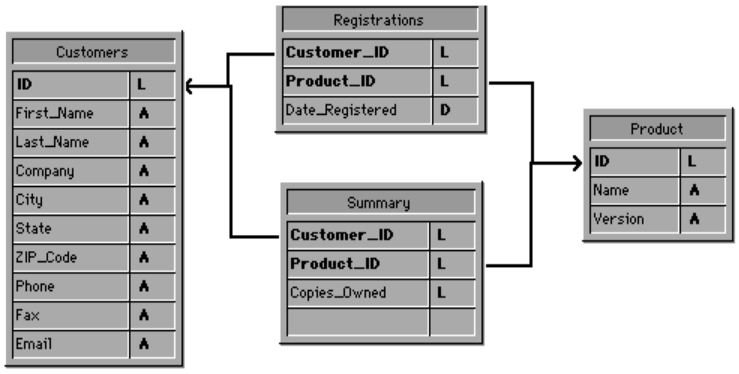
Table	Count
Customers	10,000
Products	100
Registrations	50,000

With even a modest number of products and customers, the registrations table quickly becomes very large. A large registration table is desirable, as it means that ACME will remain profitable. However, it makes a variety of other tasks time-consuming:

1) Finding customers who own the product SuperGoldPro Classic.
2) Finding customers who own three or more copies of SuperGoldPro Workgroup.
3) Finding customers who own three or more copies of SuperGoldPro Workgroup and exactly two copies of SuperGoldPro ReportWriterPlus.
4) Showing a registration trend report that tells, for each product, how many customers have one copy registered, how many have two copies, and so on.

You *can* answer these questions with the existing structure. The problem is that it is too difficult for users to do themselves, requires custom programming, and can take a *long* time to execute.

In a case like ACME's, keeping a summary table to simplify and optimize querying and reporting is well worth the extra space and effort. Here is a modified table structure:



ACME's registration system structure with a summary table

The Summary table forms a many-to-many relationship between Customers and Products, just like the Registrations table. From this structure view it does not seem that the Summary table offers any benefit. Consider, however, these representative record counts:

Record Counts with Summary Table

Table	Count
Customers	10,000
Products	100
Registrations	50,000
Summary	13,123

The Summary table stores only one record for each product that a customer owns. In other words, if the customer owns ten copies of a particular product they have ten registration records and one registration summary record. The relationship between the number of records in the Summary table and the Registrations table is data-dependent. You need to examine your data to find when a summary table is efficient. Consider the range of possible values:

Possible summary table record counts

Registrations	Summary	Reason
50,000	1	One customer registered every product sold. (Try to double your customers!)
50,000	10,000	Multiple-copy registrations are common.
50,000	37,500	Multiple-copy registrations account for roughly a half (25,000) of all registrations.
50,000	50,000	No customer owns two copies of any product. (Time to increase the marketing budget!)

In cases where the record count is lower in Summary than in Registrations, an equivalent join from Summary is faster. Not only this, you can now find customers based on complex registration conditions with simple indexed queries.

Let's look at how you would satisfy the requests listed at the beginning of this example with and without the Summary table.

TASK 1: A SIMPLE QUERY

Find customers who own the product SuperGoldPro Classic.

Steps required to complete task 1.

Without Summary table	With Summary table
Query Registrations for SuperGoldPro Classic.	Query Summary for SuperGoldPro Classic.
Join from Registrations to Customers.	Join from Summary to Customers.

The steps are exactly the same in this example. The difference is that the Summary table never contains more records than the Registration table, and normally contains far fewer. The speed of a join is dependent on the size of the initial selection, so reducing the initial selection directly reduces the time required to complete the join.

TASK 2: A MORE COMPLEX QUERY

Find customers who own three or more copies of SuperGoldPro.

Steps required to complete task 2.

Without Summary table	With Summary table
Query Registrations for SuperGoldPro	Query Summary for records with SuperGoldPro and three or more registrations
Sort Registrations by Customer_ID	
Create an empty set in Customers	
Step through each record in Registrations, counting up how many copies a customer has. If a customer owns three or more copies, add them to your set.	Join from Summary to Customers.

With the summary table in place it is no more difficult or time consuming to answer this more precise question. A simple two-field indexed query is followed by a join. An interface that gives end users access to this feature is easy to construct. Without a summary table you have to perform a sequential operation on every matching registration—a laborious and time-consuming operation.

TASK 3: A COMPOUND QUERY

Find customers who own three or more copies of SuperGoldPro Workgroup and exactly two copies of SuperGoldPro Report WriterPlus.

Steps required to complete task 3.

Without Summary table	With Summary table
Query Registrations for SuperGold-Pro Workgroup.	Query Summary for SuperGoldPro Workgroup registrations with a registration count of three or more.
Sort Registrations by Customer_ID.	
Create an empty set in Customers.	
Step through each record in Registrations, counting up how many copies a customer has. If a customer owns three or more copies, add them to your set.	Join from Summary to Customers.
Create a set of the found customers	Create a set of the found customers.
Create an empty set in Customers	
Query Registrations for SuperGold-Pro ReportWriterPlus.	Query Summary for SuperGoldPro Workgroup registrations with a registration count of exactly two.
Step through each record in Registrations, counting up how many copies a customer has. If a customer owns two copies exactly, then add them to your set.	Join from Summary to Customers.
	Create a set of the found customers.
Intersect the two Customer sets to locate the correct customers.	Intersect the two Customer sets to locate the correct customers.

Each of these solutions appears complex, but with the summary table in place the routine needed is short (commands are shown without full parameters) and quick:

```
QUERY([Summary])
RELATE ONE SELECTION([Summary];[Customers])
CREATE SET([Customers])
QUERY([Summary])
RELATE ONE SELECTION([Summary];[Customers])
CREATE SET([Customers])
INTERSECT
USE SET
```

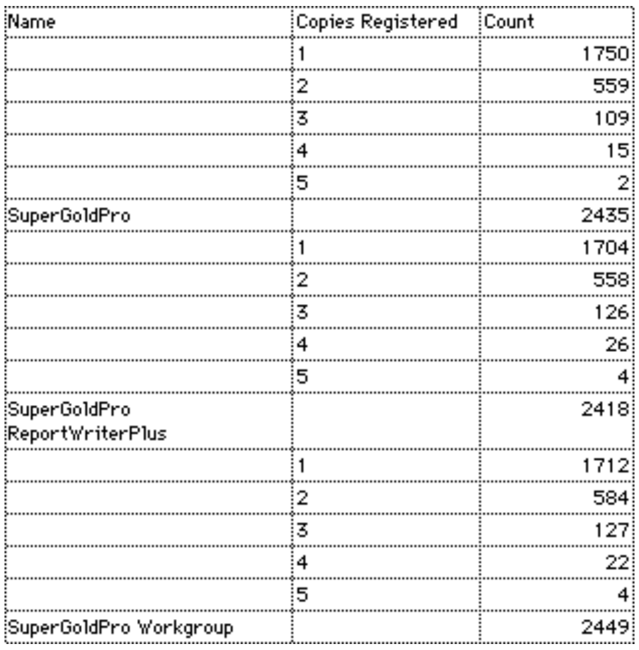
The queries and joins are all indexed, non-sequential operations, and set operations are extremely fast regardless of selection size. The alternative method, without the summary table, requires error-prone custom programming, direct record access, sequential inspection of records, and excessive network activity under 4D Server.

Notice in the previous examples that after the complicated calculations needed without a summary table are completed, *all the work is lost.* You have found the correct customers for that query, but if another user wants the same query *the entire process is repeated.* If the same user wants a slightly different set of conditions they have to wait through another long query. This gives the user the impression that the system is slow, hard to work with, and hard to get data out of. By saving these calculations in a table for quicker and easier access you improve life for your users. In addition, you can now easily generate a variety of reports on the summary data itself using the Quick Report editor, or any other system you like.

TASK 4: A TREND REPORT

Show a registration trend report that tells us for each product how many customers have one copy registered, how many have two copies, and so on.

The easiest way to provide this report with a summary table in place is with a Quick Report:



Name	Copies Registered	Count
	1	1750
	2	559
	3	109
	4	15
	5	2
SuperGoldPro		2435
	1	1704
	2	558
	3	126
	4	26
	5	4
SuperGoldPro ReportWriterPlus		2418
	1	1712
	2	584
	3	127
	4	22
	5	4
SuperGoldPro Workgroup		2449

A sample Quick Report showing registration totals and trends.

This report requires *no* code, extra calculations, or special programming. All it requires is the time to sort and count the records, like any sorted Quick Report. A user can create, modify, or export this simple report.

Cost of Maintaining a Summary Table

In order for a summary table to be useful it needs to be accurate. This means you need to update summary records whenever source records change. In our example, this means that summary records need to be changed every time a registration is added, deleted, or modified. This task can be taken care of in the validation script of the record, in the **After** phase of the form (4D 3.x), in a trigger (preferred), or in batch mode. You must also make sure to update summary records if you change source data in your code. Keeping the data valid requires extra work and introduces the possibility of coding errors, so you should factor this cost in when deciding if you are going to use this technique.