FileMaker
An Apple Subsidiary

Design:
Performance

Design: Performance

About this Guide

This guide outlines best practices for designing FileMaker solutions that perform at an optimum level. Solutions that are optimized for performance accomplish two main things:

1. They are easier for developers to maintain.
2. They are more delightful and satisfying for everyone to use.

Who can benefit from the guide?

This guide is designed for all levels of FileMaker developers who want their solutions to run at peak performance. It is simple enough for those that are new to FileMaker and are just starting to develop their first solutions.

It also has enough substance and depth for those with many years of experience who are looking for more ways to maximize the performance of their existing solutions.

What you will learn

You will learn about the top factors that affect performance and how to implement techniques that will significantly boost the actual (and perceived) performance of your FileMaker solutions by focusing on these key areas:

1. Layout Performance
2. Schema Performance
3. Solution Logic Performance

If this sounds good to you, let's get started!

Introduction

Why Performance Matters

Performance is an important design factor for all software solutions. It is something that you should consider early in your development process and continue to improve throughout the lifetime of your solution.

Performance of your solutions is especially important when used with FileMaker Go for iOS and FileMaker WebDirect. These technologies offer your users an unprecedented opportunity to access their information from virtually anywhere. Both of these technologies can be useful within an office environment, where a fast, local-area-network is available. They are also well suited for mobile and remote users, who will often be accessing your solution across slower, wide-area-networks, such as the Internet.

A FileMaker solution that performs well in an office environment may suffer degraded performance when accessed by users over a network with limited bandwidth. It's also possible to create a solution with enough complexity or data that even local users find its performance degrades over time.

It's more important than ever that developers keep performance in mind when designing, developing and deploying solutions. Users want solutions that are not just easy to use, but which save them time and effort. This is especially true with today's mobile workforce, as well as for users of your system who need to access it using a web browser.

Measuring Performance

Performance can be measured in a number of different ways. Examples include:

- how fast a solution launches
- how quickly a user can complete a task
- how long it takes a user to retrieve desired information

While some of these measures can be assessed objectively, using a stopwatch, others will depend on the perceptions of individual users or developers.

Performance for a hosted solution is heavily dependent on the amount of work the client or server has to do (i.e., performing calculations, drawing layouts, etc.) and, more specifically, how much data must be sent across the network to support that work.

To develop a solution that performs well you will need to plan ahead and design to minimize both of these factors.

Planning for Performance

Understanding how FileMaker works under the hood is the first key to reducing client and server work. The decisions made while planning and developing can dramatically affect the performance of a solution. This guide provides in-depth information about how FileMaker works and how to make smarter decisions while designing the interface, schema, and logic of a solution.

Consider the ideas in this guide when you are planning your solution, and before you start developing. Following these guidelines will enhance the usability and appeal of your solutions. You can also apply these techniques to solutions that were developed in a prior version of the FileMaker Platform. Such investments will improve the level of performance for those solutions as you upgrade them and over time.

Layout Performance

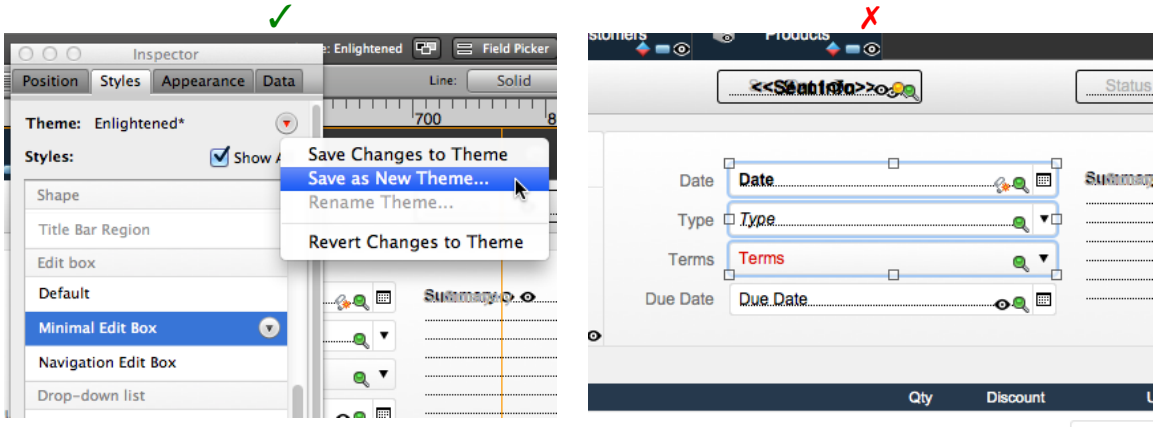
Layouts are the primary method by which information is triggered to be sent over the network. The objects on your layouts and how those objects are styled, determine the amount of data sent over the network - and in turn, the performance of your solution.

Your aim when seeking the best layout performance is to design layouts that send the optimal amount of information with the least amount of overhead. In order to optimize the amount of data sent over the network, you will want to focus on two key components:

1. **Themes and styles** – create and save custom themes and styles and avoid using local styles where possible.
2. **Layout objects** - keep your layouts responsive and intuitive by using fewer objects on each layout.

Themes and Styles

Create and save custom themes and styles and avoid using local styles.



Themes and styles define the visual elements

Understanding how themes and styles work together will help you maximize the visual impact of your solutions without sacrificing performance.

A theme is a set of coordinated instructions that determine the default color, size and font of different objects on a layout. These instructions are known as styles. Styles are made up of various properties that describe the attributes of an object, such as its fill color or border style.

Themes also provide a convenient way to store color and typography for your entire solution. This can make developing and maintaining a solution significantly easier. It will also reduce the amount of styling information that needs to be stored in the solution, or sent from the server to the client.

Themes can lead to faster layout refreshes

A theme is only downloaded from the server to the client once, after which it is cached on the client and referenced by each layout that uses that theme. This is an efficient method to improve solution performance, as it allows many separate layouts to share a single set of instructions.

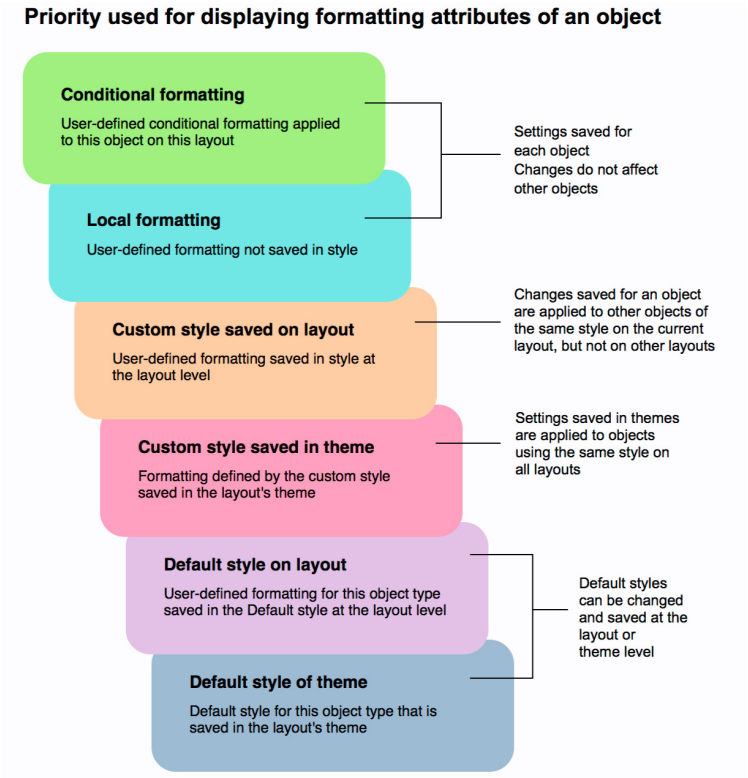
FileMaker stores information about how to style an object in a series of stacked layers. Each layer serves a particular purpose:

- **Theme Styles** – These are styles that come with the theme or were saved to the theme by the developer. These styles form the foundation of the stack, and are the most efficient place to define a style when styling for performance. When these styles are updated, they are updated across all layouts that use the same theme in the same FileMaker file.

- **Layout Styles** – These are new styles that were created on a specific layout, but haven't been saved back to the theme. They provide efficient styling of multiple similar objects a particular layout, but do not impact objects on other layouts in the FileMaker file.

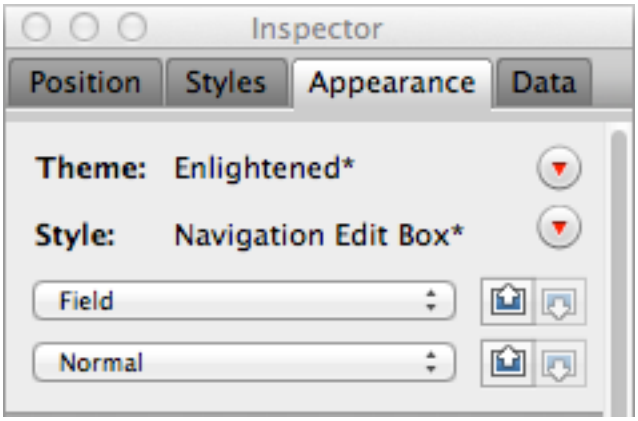
- **Local formatting** – These are changes made in the Inspector that have not been saved to any specific style on the style tab. They are entirely local to the object they are styling and are the least efficient way to style an object.

In Module 3 of the FileMaker Training Series: Advanced for FileMaker 13, you will find a useful diagram that illustrates the hierarchy that organizes these layers:



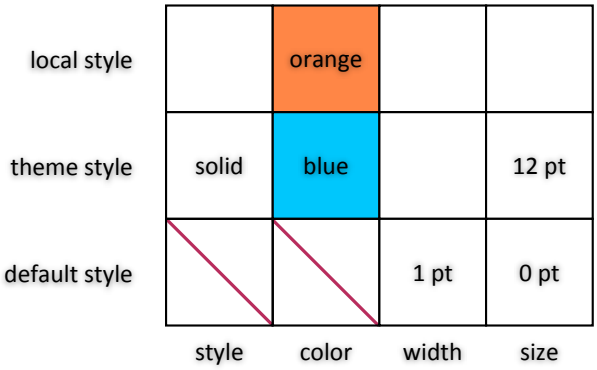
In a solution that is optimized for performance, each object will be styled by as few of these layers as is necessary.

Fortunately, in Layout mode, the **Inspector** provides the developer with visual indicators when an object is locally styled, and when there are Layout Styles not yet saved to the Theme:



You can read about these indicators and how to use them to guide you when managing styles in the FileMaker Help documentation.

Here's a simplified example of just such a stack for a specific object property, such as the topmost border of a specific field:

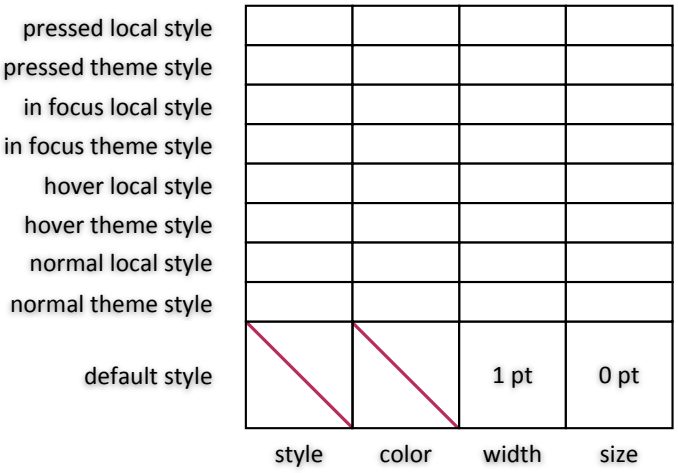


Notice how we are defining the color of this property twice – first as part of the theme style, and then again as a local style. Every additional layer of style information increases the volume of instructions that must be stored in the solution and then read and applied to layout objects as they are rendered. The more layers of style information required to render each specific object, the more information has to be sent across the network from the server to the client, and the harder FileMaker has to work to render the object.

Theme styles are applied to similar objects across a layout and take up relatively little storage space in a solution. They are usually quick and easy for FileMaker to apply to objects, as it only has to store and read the instructions once.

On the other hand, local styles add additional bulk to the solution, and require FileMaker to take additional steps to read the instructions and then apply them to the specific object to which they were assigned.

This might not seem like much additional information, but the effect on performance increases when you consider all of the additional styles that might be associated with up to 4 object states as well as all the 30 or more attributes you can define for different objects:



The effect can become even worse if multiple objects are using local styles, which can happen when using the **Format Painter** or copying and pasting of styles, especially between different types of objects.

These behaviors can be seen most easily when using the Classic theme, which has no theme styles and only uses local styles. In this scenario, every attribute of every object is stored and drawn separately. If you plan on using the Classic theme, only use it as a starting point from which to create your own custom theme and styles.

Themes provide efficiency in design and updates
With saved themes, you are able to keep the visual aspects of the solution separate from the content. This allows you to quickly create layouts with a consistent look and feel, and troubleshoot problems efficiently. By controlling the appearance of all of a layout's elements from within a theme, you increase your speed as a developer, as well as the speed of the solution.

Quick Tips

Do

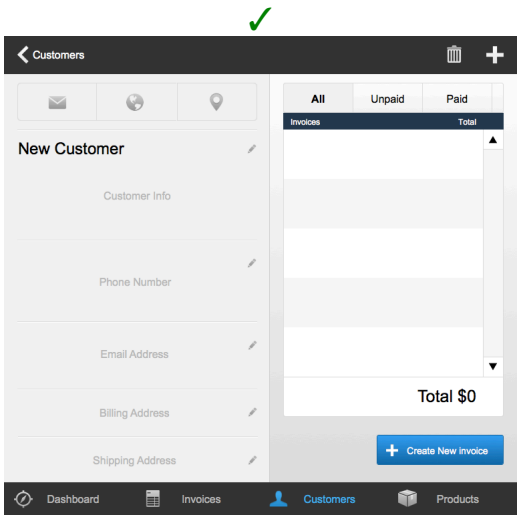
- Create custom themes of your own.
- Save custom styles to a theme.

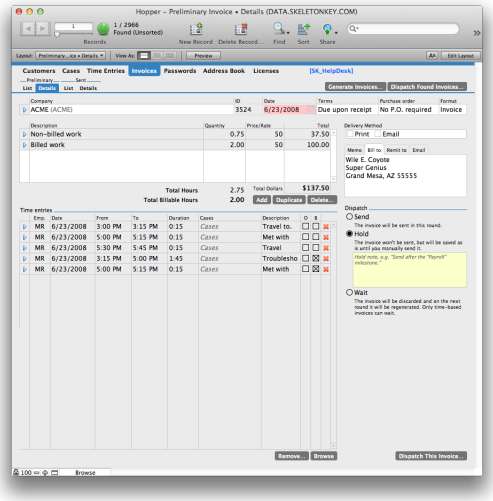
Don't

- Use a lot of local styles.
- Remove all styles from an object.
- Overuse **Conditional Formatting**.
- Copy and paste styles between objects (unless you plan to save them to a theme style).
- Use the Classic theme, as it only has locally styled objects *(unless you plan to use it as a starting point for creating a custom theme).*

Layout Objects

Keep your layouts responsive and intuitive by using fewer objects.





Layout objects communicate context

The objects on a layout communicate a variety of important facts to the user, such as which record is being viewed, what actions can be performed and what information can be accessed.

Fewer objects on a layout can help a layout render faster
It can be tempting to create a layout that contains everything a user might want to see about the current record. This might include fields that display data from the current record, and which display or aggregate data from several local or related records. This is true in list views as well, where the impulse to provide several summary fields can be strong.

Every field that is displayed on a layout takes time to be rendered on the screen. As information is updated and committed to the solution by users, a layout may need to refresh in order to display the information that has been changed.

Fewer objects can also dramatically speed up screen refresh times, as fewer objects need to be redrawn when information changes.

Fewer objects on a layout can help reduce network traffic
Depending on how you've designed your database schema (which is covered in the next section), having fewer objects on a layout can reduce the amount of information that needs to be sent from the server to the client.

For example, if a portal is being used to display related records, the data for those related records is usually only sent to the client when the portal is displayed to the user. Rather than having a portal of related records visible immediately upon viewing the record, it can be helpful to relocate that portal to a layout where it will only be shown when it is needed.

Alternatively, you could place that portal on the second panel of a **Slide Control**. While this will add another object to the layout, it will also allow a user who does not need to see that related data to effectively bypass that data altogether, and thus avoid paying an unnecessary performance penalty. It's important however to use this technique sparingly, so as to avoid creating overly complex layouts with several different or even nested slide or tab controls.

Quick Tips

Do

- Create simpler layouts with fewer objects.
- Be deliberate about when you reveal related records or use calculations based on related records.
- Sparingly use tab and slide controls to reveal information as needed.

Don't

- Create layouts that contain a large number of objects.
- Create a single layout that shows all the fields for a record.
- Display several portals or related records at the same time.

Additional Resources

For more information on improving layout performance, review the following FileMaker Support resources:

Introduction to styles and themes in FileMaker Pro
http://help.filemaker.com/app/answers/detail/a_id/11895/

Style Picker, Style Editing, and Theme Saving in FileMaker Pro
http://help.filemaker.com/app/answers/detail/a_id/11970/

How do you create and save your own themes in FileMaker Pro?
http://help.filemaker.com/app/answers/detail/a_id/11674/

FileMaker Training Series: Advanced
http://www.filemaker.com/support/training/fts.html#fts-adv

iOS: Performance
https://fmdev.filemaker.com/docs/DOC-3843

Schema Performance

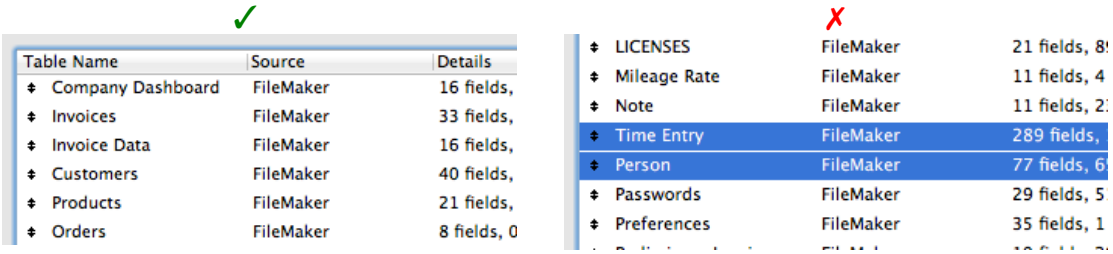
The design of your schema is the foundation upon which the rest of your solution is built. The schema is comprised of the tables, fields, and relationships (also known as 'joins') that store and interlink your data. In the FileMaker Platform, the schema also helps determine how that data will be apportioned when it's sent over the network.

Your aim when seeking the best schema performance is to organize the tables and joins so that the optimal amount of information is available when needed, but to otherwise limit the movement of unnecessary data. In order to provide just the right amount of information needed, you will want to focus on three key components:

1. **Tables, fields and records** – split your larger tables into several, smaller related tables.
2. **Found sets** – only send records that are needed.
3. **Relationships –** use an organized approach to joins.

Table, Fields and Records

Split larger tables into several, smaller related tables.



Tables, fields and records organize how data is stored and moved

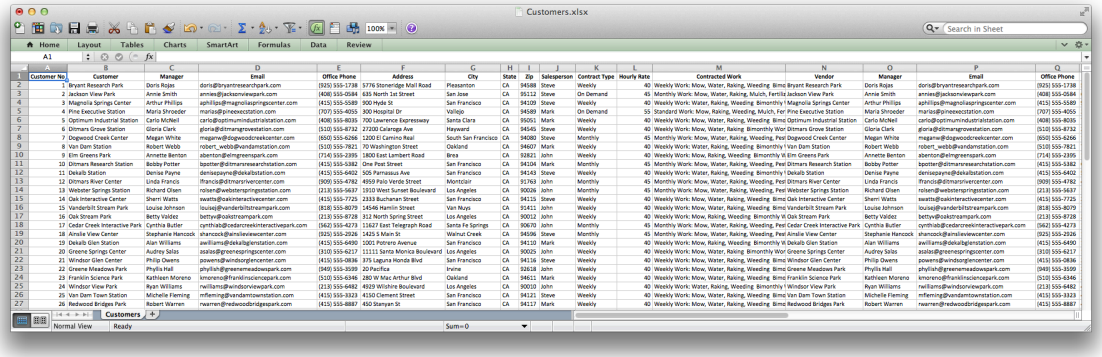
FileMaker moves data from the server to the client one entire record at a time. With only a few exceptions, this means that all of the fields in a table for any given record move as a single block of information.

When an edit is made to the record by a user and committed into the solution, the same thing happens in reverse – that entire block of information moves from the client back to the server.

As edits are made to records by one user, small notifications are sent to all other users about the obsolescence of any record data they may have already downloaded. When a user later returns to a record they had previously viewed, but which another user had since changed, the whole record is then downloaded again.

Learn Relational Design

It's easy to create a FileMaker solution from an existing spreadsheet. Unfortunately, many spreadsheets are very 'flat', consisting primarily of one very big table, with lots of columns:



When a spreadsheet like this is converted into a FileMaker solution, it creates one very wide table. Each of the rows in that spreadsheet is now a record in that table, and all of those columns are now fields in that table. As described above, each of these records will move as a block of data, and each block will contain all of those columns.

There are a number of good reasons why data in one big table should be broken out into several different tables. For example, analysis of the data may indicate that some of the fields describe a customer, while others describe transactions conducted with that customer. Two tables might be a better solution for organizing these two unique sets of information.

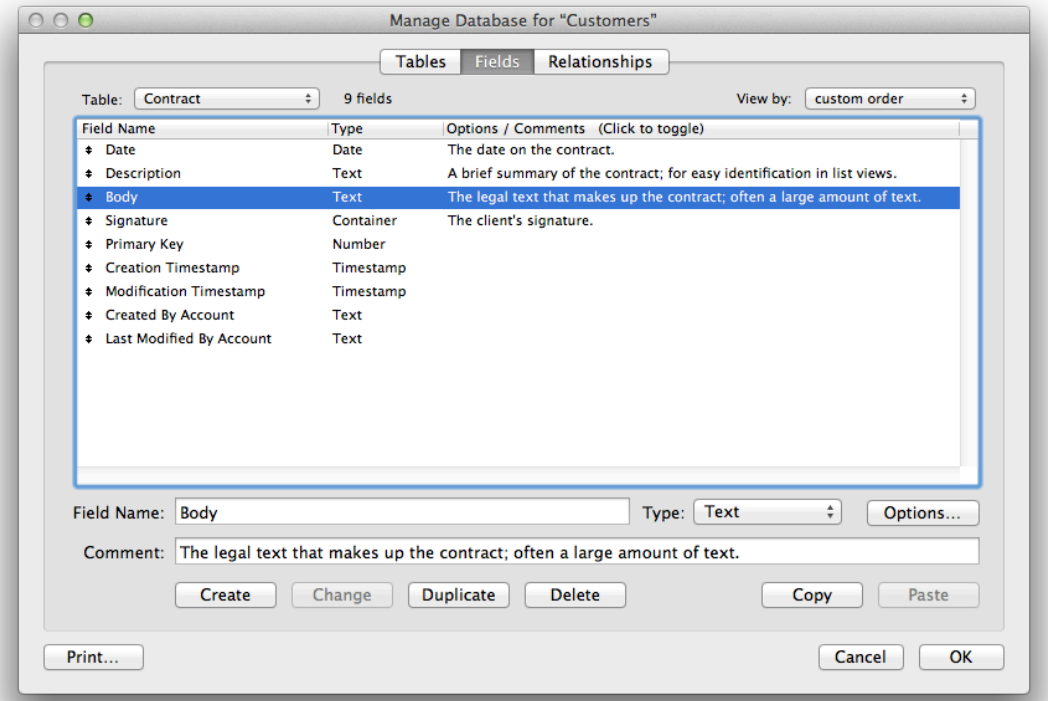
Unless the data you are managing is extraordinarily simple (i.e., a very simple list), odds are the data in your solution is varied, can be organized into different sets of similar types, and there are relationships between the different sets which is critical to both managing and understanding the data in the first place.

Whether you're trying to convert a large table of data into a smaller set of related tables, or you're planning a new solution and want to avoid creating large or bulky tables in the first place, a good foundation in relational design can make a big difference in the performance of your solutions.

For a good overview of relational data modeling, we strongly suggest you review the FileMaker Training Series: Advanced module on Data, which is referenced in the Resources section below.

Tables can be used to stage how data moves

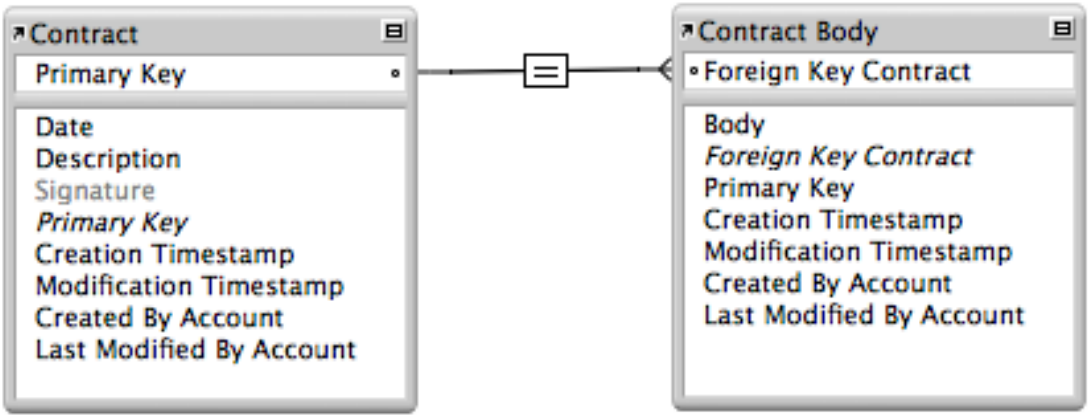
When a developer designs a database, they often put all the fields that discretely define an entity into a single table. Take this simple example, which shows the fields that make up a 'Contract' record:



The 'Body' field has been identified as often containing large amounts of text. Since this field is part of the record, it will be downloaded from the server to the client when the user reads any of the other fields in the table.

If a user then edits any part of this record – for example changing the Date field – the large block of text in that field will also be sent back to the server. The data in that field and the other fields in the same table will forever travel as a set.

A developer can mitigate this issue by splitting this table into two separate but related tables:



By doing this, the large 'Body' field is isolated from the main Contract record. The client will only download the content of that large text field when that specific field is displayed to the user. This makes the main

Contract records smaller, faster to download initially, faster to upload when changes are made, and faster to re-download if necessary.

This is often referred to as taking a 'narrow' approach to table design. Narrow tables allow a developer to exert more discrete control over what portions of a record will move from the server to the client. This can have a dramatic boost on performance, as only the data that is needed for a given layout or action will be moved.

This can be an especially useful technique for solutions that will be accessed from iOS or over FileMaker WebDirect, where you may be more likely to find use cases that only require a subset of fields from a much larger schema.

Another scenario where a narrow approach would be useful is when fields that would normally be in the same table are changed with dramatically different frequencies.

For example, a relatively small 'Quantity Available' number field in a Product Inventory table may need to be updated several times a day, perhaps as frequently as every minute, and by several different clients. At the same time, a number of other fields that describe the Product –Name, Description, Category, Price – may hardly ever change.

If all of these fields remain in the same table, all of them will be downloaded initially, and then discarded, and then re-downloaded again, as various clients each take their turn at making updates to the 'Quantity Available' field.

Moving the busier 'Quantity Available' field to a separate table allows the server and clients to send much less data and spend much less time communicating about those frequent updates. By separating the fields, the more general Product data will not need to be repeatedly discarded and re-downloaded with each update, but may only need to be downloaded by any client once per session.

Blank layouts don't trigger the movement of data
A blank layout with no fields on it will not trigger any data from the associated table to be downloaded. This behavior can prove to be a convenient method for controlling when data from a table moves to the client.

The simplest way to take advantage of this fact is when a user first logs in. To do this, you'll need to perform two simple steps:

1. After developing the solution locally on your desktop, be sure to exit the solution from a blank layout. This will make that same blank layout the first layout that is displayed to a user when the solution is hosted. One common approach to this simple habit is to use an **OnLastWindowClose** script trigger that performs this type of 'clean-up' when developing.

2. Configure the **File Options** for your solution carefully. If you use the **Switch to layout** option, make sure the layout you select has no fields or objects on it that will trigger the movement of data the user does not need. If you use the **OnFirstWindowOpen** or **OnWindowOpen** Script Triggers, make sure they do not perform scripts that will trigger the movement of data the user does not need.*

*The section below, titled 'Solution Logic Performance' will discuss how scripted actions and calculations can also trigger the movement of data.

Quick Tips

Do:

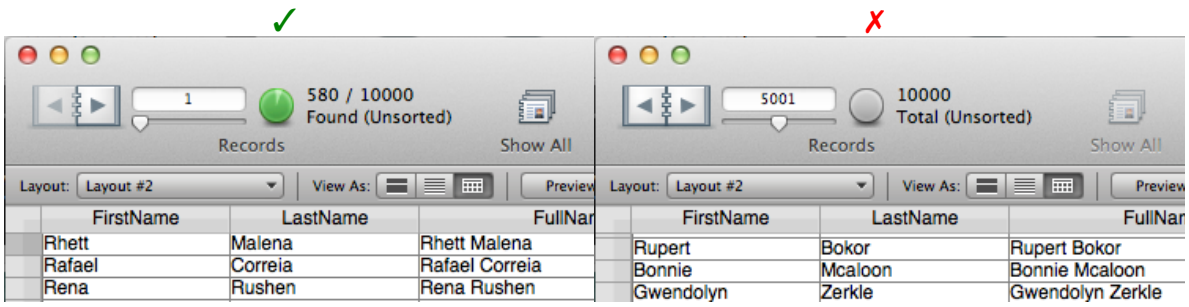
- Separate wider tables into sets of narrower, related tables.
- Use a separate table for fields that change frequently.
- Use a separate table for fields that contain a lot of text.
- Use blank layouts strategically, to avoid moving data by accident.

Don't:

- Create wide tables with lots of fields.
- Intermingle 'busy' fields and 'bulky' fields in the same table.

Found Sets

Only send records that are needed.



The found set establishes a limit on how much data is sent to the client

Regardless of how big each record is, the number of records that move between the server and the client can play an important role in how your solution performs.

In **Form** view, up to 25 records of data will move from the server to the client when a user traverses a record. In **List** view or **Table** view, FileMaker moves all the records that are currently displayed in the window, which can easily exceed 40 or 50 records at a time, especially on larger displays.

Manage the user's found set

It's important that a user is given all the data they need to perform their job. It's equally important to limit the amount of extraneous data you send to a user. Data that is not needed can reduce solution performance as well as confuse users.

There are a number of simple ways a developer can accidentally send a user unnecessary records. Common scenarios include traversing layouts with record data during the login process, or when navigating between different layouts. Unnecessary records might be downloaded when a **QuickFind** is executed across too many fields, or if a scripted process doesn't exit from an error gracefully, like in the example above.

Another reason to manage the user's found set is to keep them from having to sort or scroll through a large set of records in order to locate the record they really want. Sorting or scrolling through all the records in a found set will move all the data for all of those records.

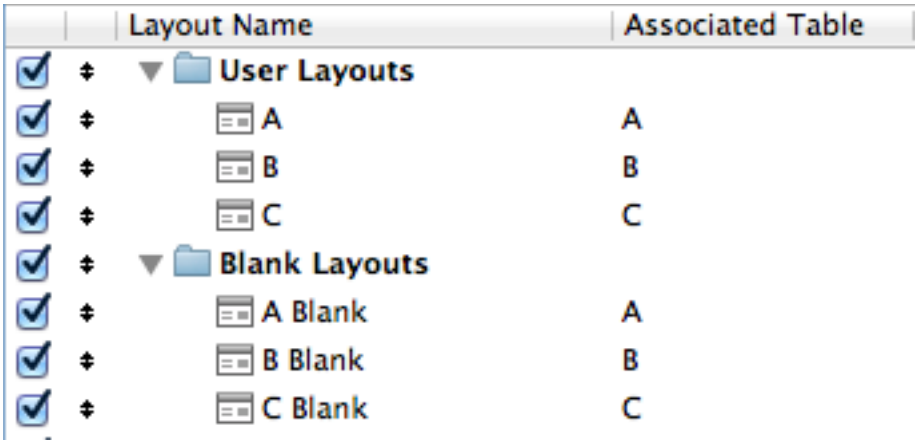
As the developer of a solution, you should always be thinking about which records might be sent from the server to the client. Your goal is to keep that communication as lean as possible, which means deliberately managing how records are sent to the user.

The default found set for any context is all records

When developing a FileMaker solution locally on your desktop, you may have noticed that most of your layouts remember the last found set that you were viewing. This behavior changes when you host your solution on FileMaker Server.

When a client opens a new session for a solution hosted with FileMaker Server, the found set for each table occurrence is all records from the associated base table. This means that from the moment the user logs into the solution they are confronted everywhere with data they may not need.

Fortunately you can mitigate this effect through some simple scripted actions. For example, some developers create a series of blank layouts in their solution, each one based on a different primary table occurrence:



A script is then triggered when the user first logs in. The script loops through those blank layouts and eliminates the found set in each one:

```
Go to Layout [ “A Blank” (A) ]
Show All Records
Show Omitted Only
Go to Layout [ “B Blank” (B) ]
Show All Records
Show Omitted Only
Go to Layout [ “C Blank” (C) ]
Show All Records
Show Omitted Only
```

Now the file is prepared for minimal data transfer. A user can now navigate from one primary layout to another without moving any record data at all.

Note that you do not need to do this for all layouts in your solution – just one representative layout for each primary table occurrence that your users are likely to traverse during their session.

Sorting and summarizing move all the records in the found set
Sorting or summarizing a found set of records is another way that developers often move unnecessary records to the client.

The reason for this is twofold:

1. Sequencing the records in a found set and by a certain field requires that the client read the value in that field and from each record in the found set.

2. Similarly, in order to summarize a field – for example to total the value from a number field – the client has to read that value from every record in the found set as well.

It’s important that you don't automatically sort a found set, or display summary fields, unless you are certain that the found set on the layout will be the correct found set or will not contain any records.

Otherwise you risk not just sending the client the first 25-50 records worth of data, but every record in the found set, no matter how big that record set may be.

Quick Tips

Do:

- Deliberately orchestrate which records you send to the client.
- Consider using blank layouts to eliminate the found set at login for every different primary table occurrence.

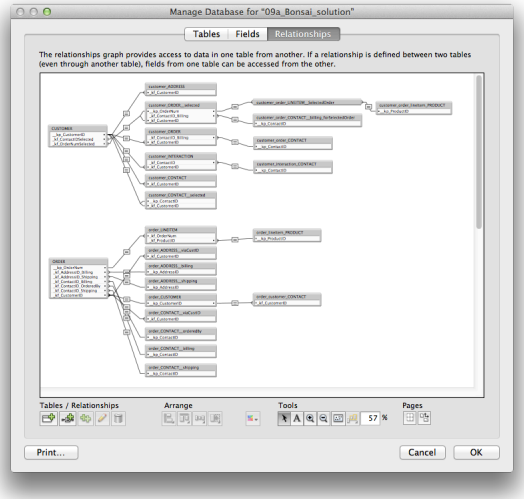
Don't:

- Automatically sort your found set *(unless the found set only contains the records you want).*
- Automatically show summary fields on a layout after performing a find *(unless the found set only contains the records you want).*

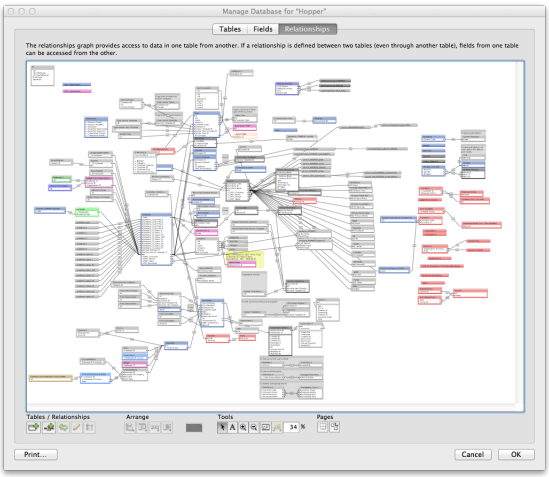
Relationships

Use an organized approach to joins.

✓



✗



Relationships define how information in different tables is interrelated.

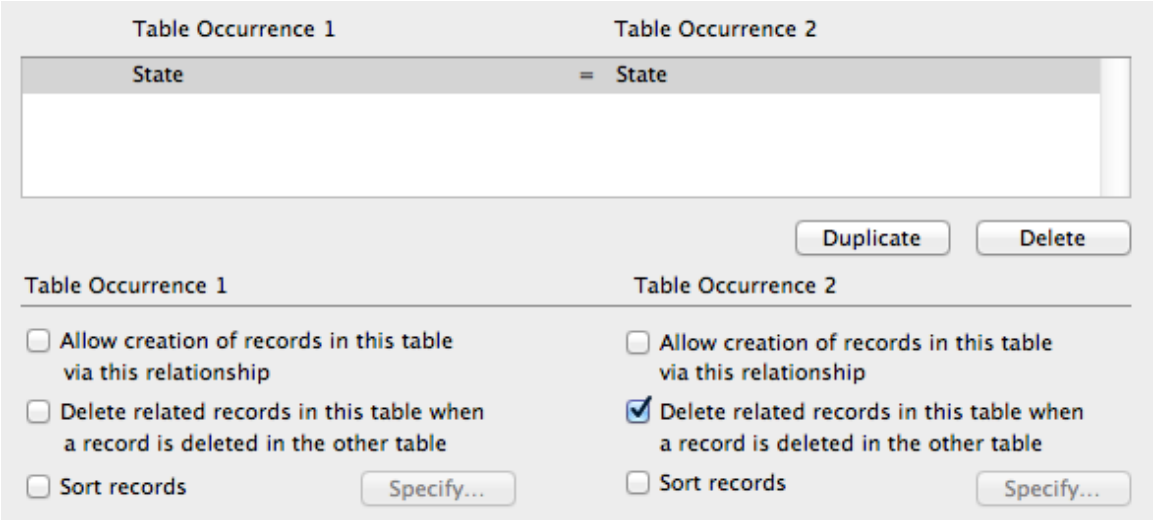
The **Relationships Graph** allows the developer to create and manage connections between data tables in the solution. These connections can be leveraged in a number of powerful ways. The most common method is to create a portal that shows related records from one table in the form view of a record in another table.

Relationships are often based on simply matching one field in one table with a field in another table. This type of connection is called an equijoin. Equijoins are one of the most common joins and do the majority of all relational work across numerous database platforms.

Use Table Occurrence Groups to keep context and scope contained

As we've described above, there are a lot of good reasons to split your larger tables into several, smaller tables for performance. But more tables generally mean more table occurrences, and the potential for more relationships.

As relationships are created, various settings can be enabled, such as 'Delete related records in this table when a record is deleted in the other table':



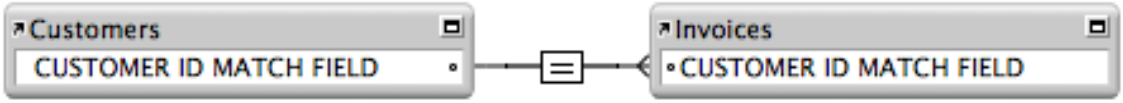
This condition is known as a dependency. Another example of a dependency is a calculation field in one table that uses fields from a related record:

= Count (Table Occurrence 2::State)

Each time you launch a FileMaker solution for the first time, something like a 'map' of the solution's schema is downloaded to the client. The schema consists of the tables, fields and relationships that are used in the solution. The client uses this 'schematic map' to adjust records and field values across the solution and maintain and respect all dependencies.

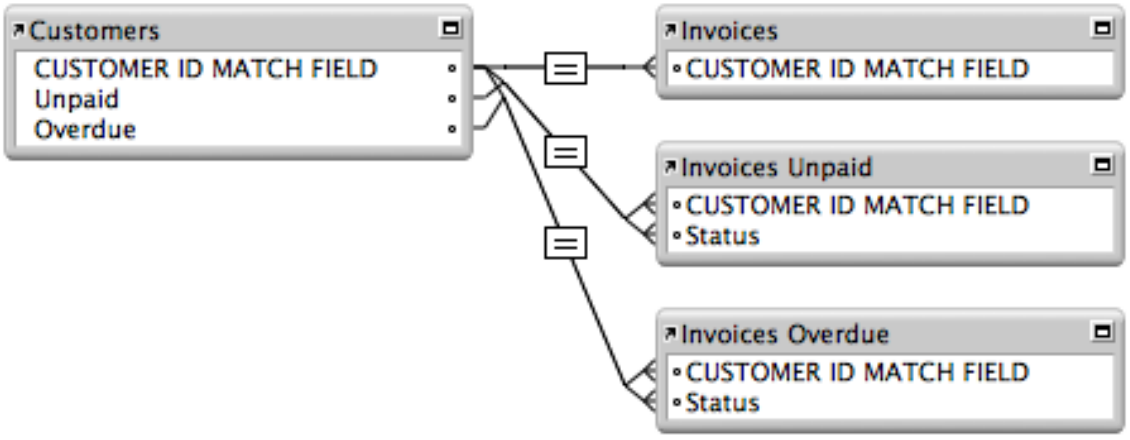
A developer can reduce the overhead of a complex 'schematic map' by creating small groups of related table occurrences (aka Table Occurrence Groups or TOGs) that focus on serving the needs of a particular layout or task. Using this approach will reduce both the number and complexity of the dependencies contained in the map, which can help the client resolve dependencies more quickly.

Use filters to streamline your relationships graph
The primary relationship between two tables is often established using an equijoin. This is fine if you want to display a portal that contains all Invoices that are related to a particular Customer:



Frequently secondary relationships are also desired. For example, a user may want to see a portal with only 'unpaid' invoices or invoices that are 'overdue'. There are three simple ways you can achieve this sort of filtering effect, and each has a different impact on performance.

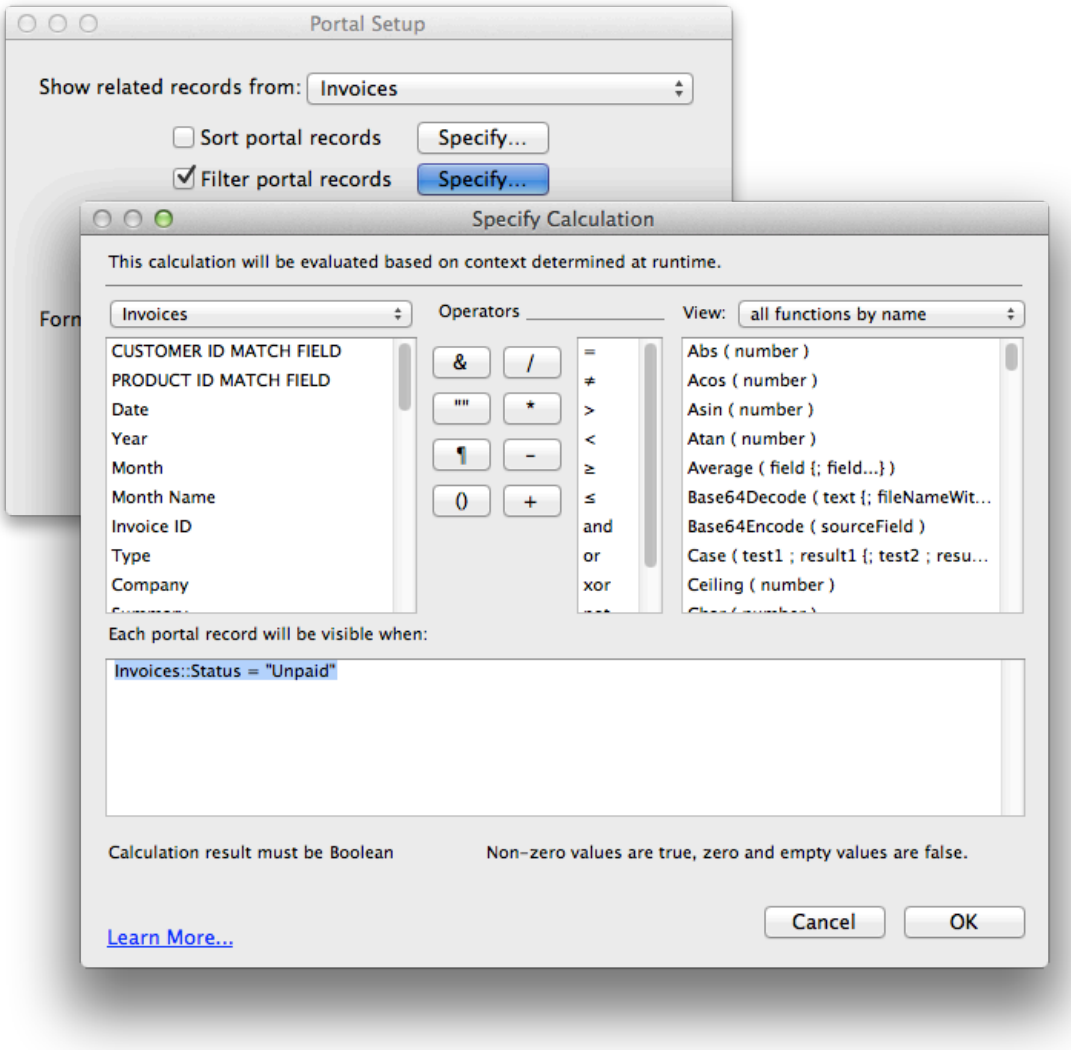
Option 1: Create additional fields, as many new table occurrences as you need, and joins that use additional predicates. Create a portal based on each of these new table occurrences.



Pros: You can summarize across each relationship and get an aggregate value for just the related records of one type or another. You can use the relationships to navigate to a found set of related records of a type.

Cons: Several additional table occurrences and joins need to be added to the graph. More dependencies. As additional statuses need to be added, more fields and table occurrences will be needed.

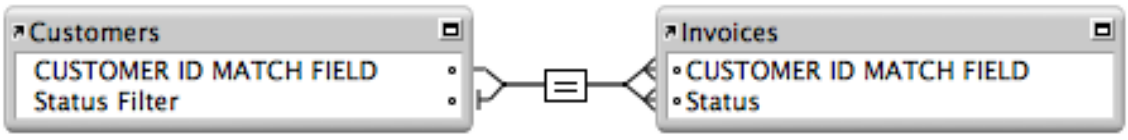
Option 2: Use the primary relationship on several different portals, and apply a filter in the portal setup for each one:



Pros: No additional fields or table occurrences are needed.

Cons: Portal filters are often applied on the client, which means all of the records for the portal will be sent to the client before the filter is applied. You cannot summarize across each filtered portal to get an aggregate value for just the related records of one type or another. You cannot navigate across the relationship to a found set of related records by type. In both cases this is due to the fact that the base relationship on the portal is still always the primary one.

Option 3: Create one new field to use when filtering the related data and assign a value list to that field:



Pros: Only one additional field added. No new table occurrences are added.

Cons: You can only filter the related records one way at any given time. One additional field might be needed if you want to filter the portal to show all related records and not just those of a type.

While both Options 2 and 3 above have limitations, they both contribute to streamlining the relationship graph by limiting the number of new fields, table occurrences and joins that are needed.

You should, of course, use whichever approach to filtering relationships is the best fit for your solution.

Use ExecuteSQL for joins that are out of context
Sometimes a developer needs to show information from a record, or set of records, that is not directly related to the table occurrence that is assigned to the layout. An example of this might be a user dashboard or 'main menu', where key metrics might need to be displayed, such as 'Unpaid Invoices' or 'New Customers This Month to Date' or 'Total Value of Current Inventory'.

For years FileMaker developers have crafted one-of-a-kind calculation fields, extraneous table occurrences, and complex joins in an effort to make these unrelated values available to the user interface. In addition to adding clutter to the schema, these constructs are often hard to change, inflexible and poor performing.

FileMaker 13 (and previously, FileMaker 12) offers the developer a new calculation function – **ExecuteSQL** – as an option for gathering this unrelated information without the need for any additional schematic elements. **ExecuteSQL** effectively allows the developer to dynamically query any table occurrences in the solution from anywhere else in the solution, regardless of any relational path that does or does not exist between the current context and the targeted table occurrences.

Learning the syntax of the **ExecuteSQL** function will be unfamiliar territory for many FileMaker developers, as it relies on a basic understanding of SQL SELECT statements. Despite this unfamiliarity, this is an important function to learn and use regularly. **ExecuteSQL** provides the developer with one of the most flexible options for accessing related information, and all without creating any formal changes to the solution schema.

Quick Tips

Do

- Use smaller Table Occurrence Groups to keep your **Relationships Graph** efficient and organized.
- Use **Portal Filters** or predicate filtering instead of creating extra fields, joins or table occurrences.
- Learn how to use the **ExecuteSQL** function to prevent cluttering your schema with rarely used table occurrences and relationships.

Don't

- Connect all your table occurrences into one big group.

Additional Resources

FileMaker Training Series: Advanced
http://www.filemaker.com/support/training/fts.html - fts-adv

Approaches to Graph Modeling
https://fmdev.filemaker.com/docs/DOC-1113

ExecuteSQL
http://www.filemaker.com/help/13/fmp/en/html/func_ref3.33.6.html

FileMaker 13 SQL Reference
https://fmhelp.filemaker.com/docs/13/en/fm13_sql_reference.pdf

Solution Logic Performance

Solution logic allows a developer to embed business rules within their solution, and to automate tasks and processes. From a performance perspective, the solution logic of calculations and scripts provides a developer with tools that can channel user activity in such a way as to prevent or avoid inefficiency.

Your aim when seeking the best solution logic performance is to employ methods that maximize the productivity of your users by consciously avoiding the unnecessary handling of data, and by delegating busywork. In order to maximize the performance of your solution's logic, you will want to focus on three key components:

1. **Calculations** – write efficient calculations by fully understanding the calculation engine.
2. **Storing Data –** store calculated values that can be indexed and summarized.
3. **Scripts –** automate processes and avoid the movement of unnecessary data.

Calculations

Write efficient calculations by fully understanding the calculation engine.

✓

```
Let (
   [ $sumTaxable = Sum ( Line Items::Taxable Amount ) ;
     $sumTaxableDisc = $sumTaxable * DiscountRate
   ] ;
      If ($sumTaxable - $sumTaxableDisc  ≥ 1 ;
      Round (($sumTaxable - $sumTaxableDisc) * Sales Tax Rate ; 2) ;
      0)
)
```

✗

```
If ( (Sum ( Line Items::Taxable Amount ) -
(Sum ( Line Items::Taxable Amount ) * Discount Rate))
≥ 1 ; Round((Sum ( Line Items::Taxable Amount ) -
(Sum ( Line Items::Taxable Amount ) * Discount Rate))
* Sales Tax Rate ; 2) ; 0)
```

Writing efficient calculations starts with understanding the engine

The calculation engine is one of the most powerful features in the FileMaker Platform. The calculation engine makes a prominent appearance in many script steps, and it appears as a supporting element in virtually every other feature of the platform, such as:

- Manage Database
- Manage Security
- Conditional Formatting
- Tab Control Setup
- Slide Control Setup
- Popover Setup
- Portal Setup
- Web Viewer Setup
- Custom Menus

To get the most out of the calculation engine, it's important to understand some basic behavior. This includes learning how some commonly used functions perform relative to one another, as well as some ways you can leverage calculations to make your solution perform better.

When calculations are performed, they do so in context

The first thing to understand about calculations is that they are executed in their own context. This means a few different things:

1. A calculation has an underlying table occurrence, which is either defined in the calculation itself or determined by which layout the user is on when the calculation is performed.

2. A calculation can incorporate record data from fields in that underlying table occurrence, as well as from related table occurrences. By doing so the calculation will trigger the movement of record data.

3. Some calculations cannot be stored. Typically this happens when the calculation references related fields or a field set to be stored globally, although a developer can specifically set a

calculation to be unstored. An unstored calculation does not cache its value, which means it must dynamically recalculate its value when needed.

4. Some calculations are dynamically recalculated at all times, such as conditional formatting calculations or calculations that are part of a script step.

Not all calculations and functions are created equal
Since calculations are so common, and since calculations can trigger the movement of data, it's important that a developer understands how to pick the most efficient functions for common scenarios.

Consider this sales tax calculation:

```
If (
(Sum ( Line Items::Taxable Amount ) - (Sum ( Line Items::Taxable Amount ) *
Discount Rate)) ≥ 1 ;
Round((Sum ( Line Items::Taxable Amount ) - (Sum ( Line Items::Taxable Amount )
* Discount Rate)) * Sales Tax Rate ; 2) ;
0)
```

While technically accurate, this calculation is highly inefficient, as it requires FileMaker to perform the same summarization of related records several times.

Using a **Let** function to calculate these values once makes the calculation perform faster and more readable for the developer:

```
Let (
    [ $sumTaxable = Sum ( Line Items::Taxable Amount ) ;
      $sumTaxableDisc = $sumTaxable * DiscountRate
    ] ;
       If ($sumTaxable - $sumTaxableDisc  ≥ 1 ; Round (($sumTaxable -
$sumTaxableDisc) * Sales Tax Rate ; 2) ; 0)
       )
```

Some Operators create shortcuts
You can get a similar speed boost from understanding how the ‘and’ and ‘or’ operators behave. Both can be used to strategically exit a calculation with a valid result, but without having to perform every expression in the calculation.

For example, let’s imagine you have two expressions in one calculation, but only one of them needs to be true. Let’s also imagine that one of those two expressions is likely to perform faster than the other. Putting the faster expression first and separating it from the other expressions with an ‘or’ operator can speed up the calculation.

In this example, the second expression will not need to evaluated at all if the first *is* true:

[a faster expression] *or* [a slower expression]

Now let’s imagine both of your expressions need to be true. The ‘and’ operator can exit the calculation if the first condition *is not* true:

[a faster expression] *and* [a slower expression]

In these examples you are leveraging the built-in behavior of the operators to steer the calculation toward the most efficient behavior. The same behavior works with **Case** function as well. As a **Case** function goes through the test expressions, it will stop once it finds a true result and will not calculate the remaining tests. Therefore always put faster calculating expressions first.

Quick Tips

Do

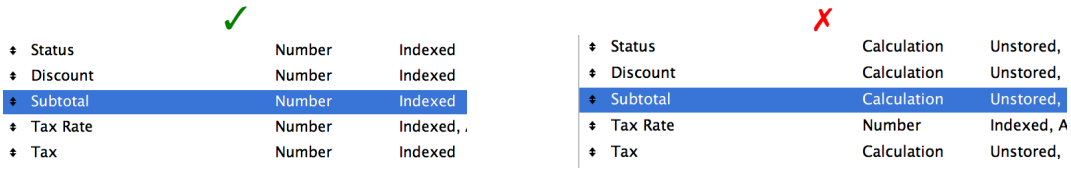
- Learn how context can impact a calculation.
- Take advantage of how **Operators** act when more than one expression needs to be true, or when only one needs to be true.

Don't

- Skimp on fully understanding the intricacies of the FileMaker calculation engine.

Storing Data

Store calculated values that can be indexed and summarized.



Create caches of data to boost performance

Sometimes values that you could calculate on the fly will perform much better if they are cached. Storing these values can reduce the number of records that have to move from the server to the client in order for a value or condition to be determined.

For example, let's imagine you wanted to find customers who had related sales totaling more than $10,000. The traditional way to do this would be to search an unstored calculation field 'Total Sales' in the customer table:

= Total (Invoices::Amount)

While this will produce accurate results, the only way for this find to be resolved is for the server (in a hosted environment) or the client (when using the file locally) to:

1. Read all the records in the Invoices table.
2. Temporarily construct the Total for each Customer.
3. Find across the temporary results.

If you stored the Total value in a number field in the Customer table, it can be indexed, which will make the find process dramatically faster. In addition, storing this value means you can display it in a list or form view, or sort by it, or summarize it, without moving all of the related records to do so.

Storing data takes planning

Storing this value means you will have to plan for anything that could change in the solution and impact that value, such as:

1. When a user creates a new Customer record, the value will need to be set to 0.
2. When a user commits a newly created Invoice record, the value will need to be updated.

3. When a user commits changes made to an existing Invoice record, the value will need to be updated.

4. When a user deletes an Invoice record, the value will need to be updated on the related Customer.

5. When one user is changing a Customer record, other users will not be able to edit the value in that same Customer record.* This means the value will need to be updated when that one user commits (or reverts) the Customer record.

Planning for all of these scenarios may seem complex, but the impact on performance will be worth the effort. At first, storing a value that you traditionally calculated on the fly may even feel a bit redundant, or risky. In reality, this is a common technique that is used throughout the programming world, as it can reliably improve system performance, especially for reporting.

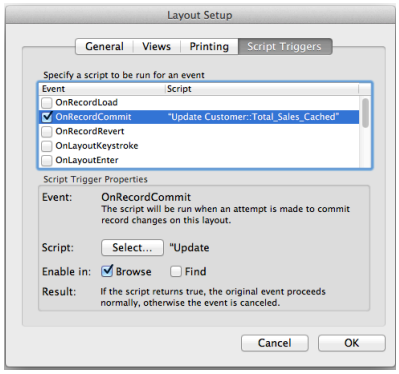
Scripts can be used to store data
While some caching can be done with auto-enter at the field level, a developer has much more discrete control of when such caching happens by using a script.

Following on the example above, imagine you want to update a new **Total Sales Cached** field when an Invoice record's **Status** field is changed from **Pending Payment** to **Paid**.

First you would need to create a script like this one:

```
If [ Invoice::Status = "Paid" ]
      Set Field [ Customer::Total_Sales_Cached; Sum
      ( Invoice::Amount ) ]
End If
```

Then you would need to trigger that script when the record is committed, using the **OnRecordCommit** script trigger, which is configured in the **Layout Setup** dialog box:



Store data using the most appropriate method
There are a number of different ways to cache unstored values, and some may work better in certain scenarios than others. Examples include:

1. Auto-enter, which can be used to populate a field with a value either initially, when the record is created, or when the user edits and then commits the record.

2. Script triggers, which can be used to populate a field with a value at several different moments, such as **OnRecordCommit**, **OnRecordRevert**, **OnLayoutExit** or **OnLayoutEnter**.

3. A scheduled script on the FileMaker Server, perhaps set to run each evening and use a **Replace** command to populate a field across a large set of records.

4. Using the **Perform Script on Server** script step to perform a similar server-side operation, but on demand.**

*The FileMaker Platform prevents two users from editing the same record at the same time using record locking, which is a standard approach to conflict avoidance in relational database platforms. Be sure to use **Set Error Capture** and test for error number 301 (Record is in use by another user) when appropriate, and handle accordingly.

**This powerful new script step is discussed in the next section of this document.

Quick Tips

Do

- Cache unstored calculations to allow them to be indexed for finds.
- Cache unstored calculations to allow them to be displayed more quickly.
- Plan for updating cached values as users create and edit data.

Don't

- Let your users perform finds on unstored calculation fields.
- Start caching values without planning for when and how they should be updated.

Scripts

Automate processes and avoid the movement of unnecessary data.

✓	✗
Set Error Capture [On]	Set Error Capture [On]
Enter Find Mode []	Go to Layout ["Invoices" (Invoices)]
Go to Layout ["Invoices" (Invoices)]	Enter Find Mode [Pause]
Pause/Resume Script [Indefinitely]	Perform Find []
Perform Find []	

Scripts can steer a user toward better performance

Scripts are a useful tool that allow developers to automate processes and avoid the movement of unnecessary data. Even simple scripts you employ every day may be accidentally moving data to your users. A bit of careful planning can avoid these inefficiencies.

Scripting to navigate around data

Let's imagine you have scripted a process that performs a common find in another table, for example finding all customer records that are marked active. That script might perform a simple series of actions like this:

```
Go to Layout [ "Customer List" ]
Perform Find [ Specified Find Requests: Find Records; Criteria:
Customer::Status: "Active" ] [ Restore ]
```

Assuming this is the first time the user is visiting the 'Customer List' layout, if there are any fields on that layout from its underlying table occurrence, then there are at least two ways this script could move unnecessary data to the client:

- When the **Go to Layout** step is performed, he will find all records are in the found set, and the server will send the client data from the first 25 records.

- When the **Perform Find** script step is performed, it could fail to find any records. The default behavior when this happens is to display the previous found set.

There are a few different script steps and methods we can use to combat these issues. While only a few of these edits might be needed for any specific scenario, the revised script below combines several of them at once, for the purpose of illustrated how they can also work together:

```
Set Error Capture [ On ]
Freeze Window
Go to Layout [ “Customer Blank” ]
Perform Find [ Specified Find Requests: Find Records; Criteria:
Customer::Status: “Active” ] [ Restore ]
```

These simple changes eliminate the threat of accidental data movement:

- Enabling **Set Error Capture** allows the found set to stay at zero records if no records are found.
- The **Freeze Window** script step tells the client to not render the changes that would otherwise be displayed on screen. Even if there were fields on the target layout, they would not be drawn while the window is frozen, and therefore could not trigger data to move.
- The ‘Customer Blank’ layout contains no fields, so when the user is first moved to it, there are no fields on it to trigger the movement of data.
- When the find is performed, it doesn’t matter if it’s successful or not, as no actual record data is triggered to move when the screen finally renders.*

*It’s important to note that no record data moves from the server to the client when the found set changes.

After a find is performed, the server sends the client a small list (called a bit map) that includes the internal record IDs of all the records that should be in the found set. This bit map is very small, and it doesn’t contain any user-defined data. User-defined data is the data that is stored inside the fields, and which constitutes the record data itself.

Scripting to avoid the accidental movement of data
An even simper example is a move to a different layout where a user is expected to perform a find. Here’s the original script:

```
Set Error Capture [ On ]
Go to Layout [ “Note” (Note) ]
Enter Find Mode [ ] [ Pause ]
Perform Find [ ]
```

As it’s written now, this script can move unnecessary data when it first navigates to the ‘Note’ layout, as the user may encounter an existing found set of records in that layout.

A simple change in sequence (red text) and the addition of one new step (green text) can make a big difference:

```
Set Error Capture [ On ]
Enter Find Mode [ ]
Go to Layout [ “Note” (Note) ]
Pause/Resume Script [ Indefinitely ]
Perform Find [ ]
```

In this revised script, the change in mode before the move to the ‘Note’ layout will keep any unnecessary record data from being sent to the user.

Perform Script on Server lets a client offload tasks to the server
Sometimes an operation that needs to be performed is simply hard to optimize. For example, updating a field across a large found set, or deleting a large number of records at one time can both demand that a lot of data to be sent across the network.

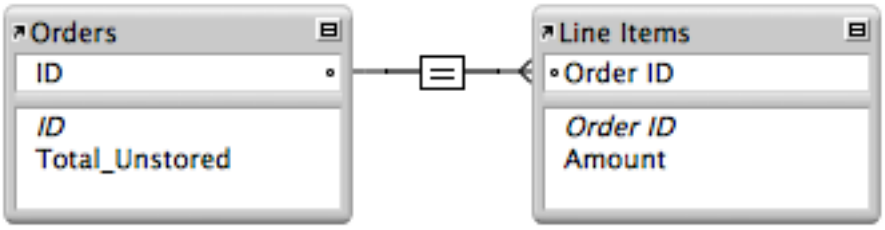
FileMaker 13 introduces an exciting new script step that enables users of FileMaker Pro and FileMaker Go to offload some of these data-intensive tasks to FileMaker Server. This new script step is called **Perform Script on Server**.

In most instances, when using FileMaker Pro on the same local-area-network as the FileMaker server, performing a script on the server will likely have little performance benefit. However, when using FileMaker Pro over a wide-area-network, or when using FileMaker Go, performing a script on the server can be many times faster.

Certain scripts are better candidates for offloading to the server:

- Scripts that create, modify or delete a large number of records.
- Scripts that aggregate or summarize data, such as scripts for reports or virtual lists.
- Scripts that need to perform a find on an unstored or unindexed field.

For example, imagine a simple schema of Orders and Line Items:



Your user needs a count of all Order records that have a total greater than $1,000, but the 'Total_Unstored' field is an unstored aggregate function:

```
= Sum ( Line Items::Amount )
```

Traditionally the only option for getting the needed count is to perform a find on the 'Total_Unstored' field:

```
Go to Layout [ "Order" (Orders) ]
Perform Find [ Specified Find Requests: Find Records; Criteria:
Orders::Total_Unstored: "> 1000" ] [ Restore ]
Set Variable [ $Count; Value:Get ( FoundCount ) ]
```

Alternatively, you could use the **ExecuteSQL** function:

```
Set Variable [ $Count; Value:ExecuteSQL ( "SELECT COUNT (ID) FROM ORDERS
where TOTAL_UNSTORED > 1000" ; "" ; "" ) ]
```

In either case, in order to evaluate the correct answer the client has to download all of the records from both tables.

Using the **Perform Script on Server** script step, and a **Get Order Count** script that uses either method described above (and which sets its result on exit), the same count could be retrieved in a fraction of the time:

```
Perform Script on Server [ "Get Order Count" ][ Wait for completion ]
Set Variable [ $Count; Value: Get ( ScriptResult ) ]
```

While the server still has to manipulate the same volume of data to derive the answer, it has no data to send over the network other than the resulting count.

Quick Tips

Do

- Plan your scripts by thinking through what data could move with each step.
- Consider how the sequencing of your script steps can make a difference to its performance.
- Use the **Freeze Window** script step to postpone screen rendering until the records you want are ready to be displayed.
- Enable **Set Error Capture** to control what happens when no records are found.
- Let **Perform Script on Server** perform tasks that need to manipulate a large volume of record data.

Don't

- Manipulate large numbers of records using scripts.
- Navigate across several different table occurrences and layouts without using the **Freeze Window** script step.

Additional Resources

Performance Optimization of FileMaker Databases
http://help.filemaker.com/app/answers/detail/a_id/5268/

Perform Script On Server
http://www.filemaker.com/help/13/fmp/en/html/scripts_ref1.36.3.html

General information about the Perform Script On Server script step
http://help.filemaker.com/app/answers/detail/a_id/12010/

About the Author

Special credit goes to Mark Richman of Skeleton Key for the development of this guide.

Learn more about Mark at: https://fmdev.filemaker.com/people/mark.richman