

A Practical Guide to FileMaker 22 OData

The canonical functions that work, server-side aggregation, cross-table joins, and a catalog of the queries that return `200 OK` and the wrong answer.

Brian Ouimette · Proof · FileMaker Server 22.0.6 · May 2026

Introduction

At Proof we lean heavily on AI for daily work — writing, agentic programming, client integrations. I used AI to help organize and write up these findings, generate the sample data, and build the Postman collection that runs every test in this guide. I personally ran each scenario against a real FileMaker 22 server and went through every word here. The Postman collection is available in the downloads at the end so you can run the same queries yourself.

A few months back I was teaching a Claude agent to query one of our FileMaker servers over OData. It seemed like basic work on the surface — the agent needs to answer questions like "which vendors did we pay the most last quarter" without a human in the loop. What I didn't expect was how much I'd end up learning about FileMaker 22's OData surface while I was at it.

This guide covers FileMaker 22's OData function library — the canonical functions that work, with examples for each — then server-side aggregation through `$apply`, then `$compute` and cross-table joins, and finally the things that fail quietly: queries that return `200 OK` and the wrong answer. We start with the function library because everything that follows uses it. `cast()` shows up inside `aggregate()`. `contains()` and `tolower()` show up inside cross-join filters. `year()` and `month()` show up everywhere there's a date involved.

A scope note. Everything below is on FileMaker Server 22.0.6. Most of the functions work on earlier versions too, but some of the more interesting combinations — `cast()` inside an aggregate, especially — are 22-specific. The `$apply` aggregation section is new to FileMaker 22 entirely; earlier versions don't have it at all.

GETTING STARTED

The Schema

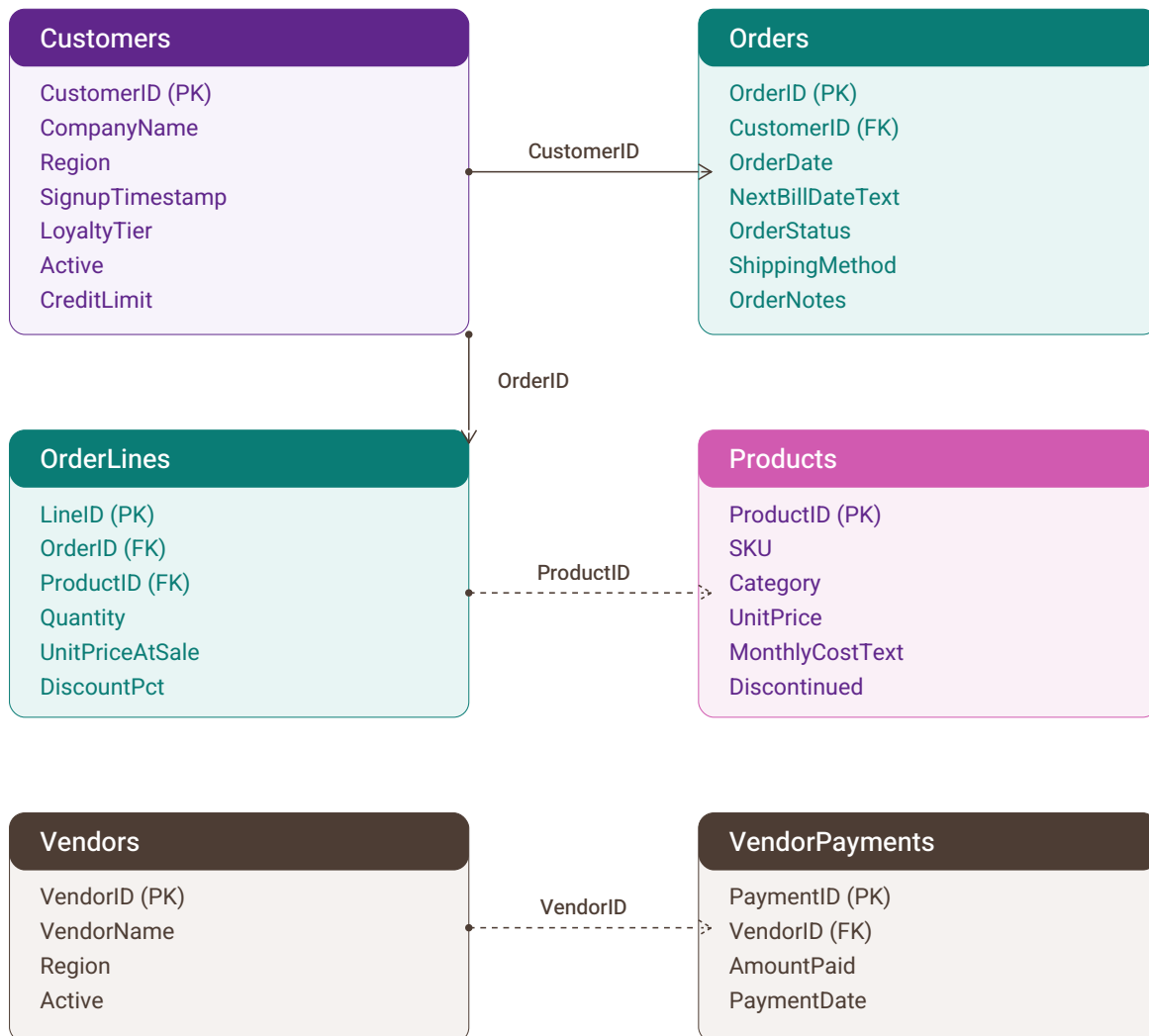
Every example in this guide uses the same fictional schema: an inventory-and-sales database with Customers, Products, Orders, OrderLines, Vendors, and VendorPayments. About 750 customers, 450 products, 6,000 orders, 16,000 order lines. The seed CSVs and import notes are published in the downloads at the end so you can run every query yourself.

OData_Demo

Inventory and sales schema

•——> graph relationship — use \$expand

•-----> join via shared key — use \$crossjoin



The OData base URL assumes the file is hosted as `OData_Demo.fmp12`, so every URL in this guide starts with:

```
/fmi/odata/v4/OData_Demo/
```

Substitute your own filename to follow along.

Fetch a Single Record

Not everything needs a full table query. When you know the primary key value of a record, two URL patterns let you go straight to it.

The first returns the full record as a single JSON object — not wrapped in a `value` array:

```
/fmi/odata/v4/OData_Demo/Orders('ORD-001')
```

This is the fastest way to retrieve a known record. No query is involved — the server does a direct key lookup and returns the result immediately. The response shape is a flat JSON object rather than the `{ "value": [...] }` envelope you get from a table query.

The second adds a field name and `/ $value` to go one level further, returning just the raw value of that field with no JSON wrapper at all:

```
/fmi/odata/v4/OData_Demo/Orders('ORD-001')/OrderStatus/$value
```

That returns the plain text value of `OrderStatus` for that record. It works on text, numeric, and date fields. For container fields it returns the binary value directly.

WHAT FILEMAKER TREATS AS THE KEY

OData designates the first non-null, unique field in a table as the key — the identifier used in these URL patterns. If your table has a proper primary key field defined (non-null, required unique), that's the one. If no such field exists, FileMaker falls back to the ROWID system field, which maps to `Get(RecordID)`. Worth knowing because if your table doesn't have a primary key defined, your key lookup URLs will use internal ROWID values rather than your own identifiers — and those values aren't stable across imports or restores.

Field Naming Rules

FileMaker will warn you when a field name contains characters that can't be used in a calculation formula — and those same names will cause problems in OData queries. Two categories to know:

Special characters — field names containing any of `+ - * / ^ & = < > ( ) [ ] { } , ; : " $ %` will break OData URL parsing. The parser treats these as reserved characters, not as part of an identifier, so the query either fails or misreads the field reference entirely.

Reserved words — field names that match `AND`, `OR`, `XOR`, or `NOT` will parse-fail as field references. The server reads them as logical operators, not field names. The workaround is URL-encoded quoting (`%22fieldname%22`), the same escape used for the `id` / `ID` case covered in the Aggregation section.

Spaces — must be URL-encoded as `%20` in field references.

Leading underscore — field names starting with `_` can behave unpredictably on FileMaker OData. This is a FileMaker quirk rather than an OData spec issue. If you're designing a schema for OData use, avoid it.

Comparison Operators

Before the function library, the building blocks. Every `$filter` expression is built on six comparison operators:

OPERATOR	MEANING	EXAMPLE
<code>eq</code>	equal	<code>OrderStatus eq 'Active'</code>
<code>ne</code>	not equal	<code>OrderStatus ne 'Cancelled'</code>
<code>lt</code>	less than	<code>UnitPrice lt 10.00</code>
<code>gt</code>	greater than	<code>UnitPrice gt 10.00</code>
<code>le</code>	less than or equal	<code>UnitPrice le 10.00</code>
<code>ge</code>	greater than or equal	<code>UnitPrice ge 10.00</code>

All the functions in the library compose with these — `contains(CompanyName, 'Trading')` is a boolean expression that fits wherever `OrderStatus eq 'Active'` fits. Multiple conditions chain with `and` and `or`. One thing worth knowing now rather than later: `ne` excludes null rows.

`OrderStatus ne 'Cancelled'` returns rows where `OrderStatus` is non-null and not `'Cancelled'` — rows with a null status fall out silently. The Silent Failures section covers the fix and the full null behavior.

String Functions

Twenty-three canonical OData functions are verified working on FileMaker 22.0.6, grouped by what they do.

String functions: `concat`, `tolower`, `toupper`, `length`, `contains`, `startswith`, `endswith`, `substring`, `trim`

Type coercion: `cast`

Date and time functions: `now`, `mindatetime`, `maxdatetime`, `date`, `year`, `month`, `day`, `hour`, `minute`, `second`

Numeric functions: `round`, `floor`, `ceiling`

Canonical OData functions on FileMaker 22

Verified against 22.0.6

Strings

<code>contains</code>	<code>startswith</code>
<code>endswith</code>	<code>substring</code>
<code>tolower</code>	<code>toupper</code>
<code>length</code>	<code>trim</code>
<code>concat</code>	

Case-sensitive by default.
`concat` is binary — nest for 3+ values.

Type coercion

`cast`

Supported targets:

`Edm.String`, `Edm.Int32`, `Edm.Decimal`,
`Edm.Boolean*`, `Edm.Date`,
`Edm.TimeOfDay`, `Edm.DateTimeOffset`

Works in `$filter`, `$compute`, and inside
`aggregate()`. *Boolean cast does not
work for text-stored 'true'/'false'.

Date and time

<code>now</code>	<code>date</code>
<code>year</code>	<code>month</code>
<code>day</code>	<code>hour</code>
<code>minute</code>	<code>second</code>
<code>mindatetime</code>	<code>maxdatetime</code>

Component extractors work on real
Date and Timestamp fields, not on
text-stored date strings.

Numerics

`round` `floor` `ceiling`

Wrap text-stored numerical values in
`cast()` before passing to these.

Context-specific

`indexof` works in `$compute`, not `$filter`

Not implemented on FileMaker 22

`replace`, `fractionalseconds`, `isof`, `geo.distance`, `geo.length`, `geo.intersects`
Engine returns "is an invalid function" — at least a loud failure. The function namespace is closed.

`contains`, `startswith`, `endswith`

The bread and butter of partial-match filters. All three are case-sensitive.

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=contains(CompanyName,'Trading')  
/fmi/odata/v4/0Data_Demo/Customers?$filter=startswith(CompanyName,'North')  
/fmi/odata/v4/0Data_Demo/Customers?$filter=endswith(Email,'tidewater.example')
```

The `endswith` example returns 31 rows against our seed data — the customers whose email is at the `@tidewater.example` domain. Useful for "find me everyone at this company" queries when you have email addresses but no normalized company field.

tolower, toupper

Reach for these whenever the user search is case-insensitive but the data isn't:

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=tolower(ContactName) eq 'victor saito'  
/fmi/odata/v4/0Data_Demo/Products?$filter=contains(toupper(ProductName),'STOVE')
```

Either function can wrap any text field, and either can be combined with `contains` / `startswith` / `endswith` for case-insensitive partial matching.

length

Returns the character count of a text field. I use this for two things: filtering on data quality (`length(field) eq 0` to find empty strings, since `field eq ''` is a parse error — more on that in the Silent Failures section), and finding rows with unusually long values that might point at a data issue.

```
/fmi/odata/v4/0Data_Demo/Products?$filter=length(ProductName) gt 18
```

That returns 51 rows against the seed data — the products with longer-than-typical names.

substring

Extracts a portion of a string by position. Zero-indexed, with start and length:

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=substring(ContactName,0,4) eq 'Vict'
```

That matches `Victor` and any other contact whose name starts with those four characters. In practice I find I almost always reach for `startswith` instead of `substring(field,0,N)`, because it reads more clearly. `substring` earns its keep when you need a slice from the middle of a value — extracting an account-number prefix, pulling a fixed-width code out of a structured identifier, that kind of thing.

concat

Concatenates two strings. It's binary — for three or more values, you nest:

```
/fmi/odata/v4/0Data_Demo/Customers?$compute=concat(concat(ContactName,' - '),CompanyName) as Display&$select=CustomerID,ContactName,CompanyName&$top=3
```

RESPONSE

```
{
  "value": [
    {
      "CustomerID": "CUST-...",
      "ContactName": "Victor Saito",
      "CompanyName": "Brookline Group",
      "Display": "Victor Saito - Brookline Group"
    },
    ...
  ]
}
```

`concat` works inside `$filter` too — `concat(FirstName,LastName) eq 'KennethSiebel'` for an exact-match "find this person regardless of how the name is split" query. We'll cover `$compute` properly in the Computed Fields section.

trim

Strips leading and trailing whitespace. Handy when you have an imported field that was zero-padded or space-padded:

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=trim(ContactName) eq 'Victor Saito'
```

For our seed data this is a no-op since the names are clean. On a legacy schema where a field has stray trailing spaces, it's the difference between getting rows back and getting zero rows back with no error.

in operator

The `in` operator matches a field against a list of values — a cleaner alternative to chaining multiple `or` conditions:

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=ContactName in ('Victor Saito','Maeve Dvorak')
```

That returns every customer whose name matches any value in the list — 2 rows against the seed data. Same result as `ContactName eq 'Victor Saito' or ContactName eq 'Maeve Dvorak'`, but easier to read and maintain when the list grows.

It works with numeric fields too. `UnitPrice in (107.39,157.78)` against Products returns exactly the 2 rows matching those prices.

Parameter aliases

Parameter aliases let you define a value once in the query string and reference it by name inside a filter expression. The alias is defined as a query parameter with an `@` prefix:

```
/fmi/odata/v4/0Data_Demo/Products?$filter=UnitPrice gt @p1&p1=500
```

That returns the 10 products in the seed data priced above \$500. The alias can appear in arithmetic expressions and be referenced more than once in the same filter:

```
/fmi/odata/v4/0Data_Demo/Products?$filter=UnitPrice gt @p1 and UnitPrice le @p1 mul 2&p1=50
```

That returns the 101 products in the \$50–\$100 range. `@p1 mul 2` evaluates to 100 at query time.

WATCH OUT

The `@` in the query parameter name must appear literally in the URL — do not percent-encode it as `%40`. The server identifies aliases by the literal `@` at the start of the parameter name; encoding it causes the server to treat the reference in the filter as an unknown field name and return `8309`.

Type Coercion: cast()

`cast()` shows up across the whole surface — inside `$filter`, inside `$compute`, inside `$apply=aggregate(...)`. The supported target types are:

Edm.String	– text or any field behaving as string
Edm.Int32	– integer numbers
Edm.Decimal	– decimal numbers
Edm.Boolean	– true/false (limited; see Silent Failures)
Edm.Date	– date only
Edm.TimeOfDay	– time only
Edm.DateTimeOffset	– full timestamp

A practical use: filtering on a numeric value when the field is text-stored.

```
/fmi/odata/v4/0Data_Demo/Products?$filter=cast(MonthlyCostText,Edm.Decimal) gt 5.00
```

A second: typing a value in the response when the consumer needs a specific type and the storage is a mismatch.

A third — and this is the one that matters most for reporting work — `cast()` is supported inside `aggregate()`, which means you can sum text-stored numerical values without first fetching every row. That's covered in the Aggregation section.

DON'T REACH FOR CAST() PREEMPTIVELY

Early on I assumed I should reach for `cast()` whenever there might be a type ambiguity, defensively. The agent picked up that habit and wrote a lot of unnecessary `cast()` s into queries. A few of those queries quietly broke. The lesson: reach for `cast()` when the server actively asks for it — an `8309` "incompatible data types" error. The unnecessary casts are the ones most likely to either fail or, worse, succeed and produce a different result.

A related question worth separating out: if a numeric field is text-stored, is the right answer `cast()` or just fixing the field type in FileMaker? If you own the schema, fixing the field type is the more permanent fix — `cast()` is a workaround, and like all workarounds it adds a layer that can fail. If you don't own the schema, or can't change it, `cast()` is the right tool.

A malformed target type — `Edm.NotAType`, for example — returns the generic `8310 "internal data formatting error"`. If you see `8310` somewhere in your `cast()` work, double-check the target type spelling.

Date and Time Functions

now, mindatetime, maxdatetime

Constants. `now()` returns the current server time as a `DateTimeOffset`; `mindatetime()` and `maxdatetime()` are the lower and upper bounds of the temporal range. The first one is what you'll use most:

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=cast(SignupTimestamp,Edm.DateTimeOffset) le now()
```

That returns every customer signed up before right now. The genuinely useful version is the date-arithmetic form, but OData doesn't have a "subtract days from now" function — you'd compute the cutoff client-side and pass it as a literal:

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=cast(SignupTimestamp,Edm.DateTimeOffset) ge  
2025-04-30T00:00:00Z
```

`mindatetime` and `maxdatetime` come in handy when you want to express "anytime" as an explicit bound rather than omitting the filter.

date, year, month, day, hour, minute, second

These extract components from a Timestamp — they let you slice by year, quarter, month, hour, in ways that would otherwise need either a precomputed field on the table or much messier client-side processing.

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=year(OrderDate) eq 2025&$count=true&$top=1  
→ { "@count": 2753, "value": [...1 row...] }  
  
/fmi/odata/v4/0Data_Demo/Orders?$filter=year(OrderDate) eq 2025 and month(OrderDate) eq  
11&$count=true&$top=1  
→ { "@count": 265, "value": [...1 row...] }
```

The combinations are where it gets useful. These patterns reappear in the Aggregation section when we look at server-side aggregation by month and by quarter.

TEXT-STORED DATE FIELDS

`year()`, `month()`, etc. work cleanly on real Date and Timestamp fields. They don't work on text-stored date fields. For those, you're back to either string comparison (`startswith(NextBillDateText, '2026-01')` for "in January 2026") or `cast(NextBillDateText, Edm.Date)` followed by component extraction.

The `date()` function extracts the date portion of a Timestamp. Useful for "did this happen on this calendar day" queries that ignore the time portion:

```
/fmi/odata/v4/0Data_Demo/Customers?$filter=date(SignupTimestamp) eq 2025-11-15
```

Numeric Functions

The three rounding functions. They expect a numeric input — wrap text-stored numerical values in

`cast()`:

```
/fmi/odata/v4/0Data_Demo/Products?$compute=round(cast(MonthlyCostText,Edm.Decimal)) as  
Rounded&$select=ProductName,MonthlyCostText&$top=5
```

`round()` rounds to the nearest integer (banker's rounding on .5 boundaries — not arithmetic rounding, which is worth noting if you care about that distinction). `floor()` rounds down.

`ceiling()` rounds up.

Their direct use in `$filter` is less common than wrapping them around a `cast()` for typed comparison, but they have one nice use worth highlighting — bucketed reporting. If you wanted to bucket products into round-dollar tiers, you'd compute the bucket as

`floor(UnitPrice div 50) mul 50` (the \$50, \$100, \$150, \$200 tiers), put it in `$compute`, and group on it client-side. We'll cover `$compute` in-depth in the Computed Fields section.

indexof — Context-Specific Behavior

`indexof` parses cleanly and looks like a general-purpose function, but its working surface is narrower than the spec suggests. On FileMaker 22.0.6, it works inside `$compute` and fails inside `$filter`.

The working form is finding the position of a character in a string and returning it as a computed field:

```
/fmi/odata/v4/OData_Demo/Customers?$filter=contains(Email,'@')&$compute=indexof(Email,'@') as AtPos&$select=ContactName,Email&$top=3
```

RESPONSE

```
{
  "value": [
    { "ContactName": "Victor Saito", "Email": "victor@brookline.example", "AtPos": "6" },
    ...
  ]
}
```

Two things to notice. First, `AtPos` rides along automatically — `$compute` aliases are response-only. Second, the value comes back as a JSON string (`"6"`), not a number. If a downstream expression needs a numeric, wrap with `cast()`:

```
/fmi/odata/v4/OData_Demo/Customers?$filter=contains(Email,'@')&$compute=cast(indexof(Email,'@'),Edm.Int32) as AtPos&$select=ContactName,Email&$top=3
→ "AtPos": 6
```

The same expression inside `$filter` fails:

```
/fmi/odata/v4/OData_Demo/Customers?$filter=indexof(ContactName,'Saito') ge 0
→ 8309 "An expression contains incompatible data types"
```

So `indexOf` is for "compute the position and return it"; for "filter to rows where a substring exists" you still want `contains()`.

There's one more constraint that costs time to debug. The natural follow-up — combining `indexOf` with `substring` to slice everything after a delimiter — works as long as you don't introduce arithmetic. This works:

```
$compute=substring(Email,cast(indexof(Email,'@'),Edm.Int32)) as FromAt
→ "FromAt": "@brookline.example"
```

The result includes the `@` because we sliced from its position. To get just the domain, you'd want `indexOf + 1` as the start position. That breaks:

```
$compute=substring(Email,cast(indexof(Email,'@'),Edm.Int32) add 1) as Domain  
→ 8310 "internal data formatting error"
```

The error is loud but unhelpful. For the domain-extraction case specifically, the workaround I've landed on is to fetch `indexOf(Email,'@') as AtPos` alongside `Email` and slice client-side.

Functions Not Implemented

A short note on a few things you might reach for from general OData knowledge:

- `replace()` — returns `8309 "replace" is an invalid function`.
- `fractionalseconds()` — also not implemented.
- The geo functions (`geo.distance`, `geo.length`, `geo.intersects`) — not implemented. There's no spatial type support.
- `isof()` — the type-test function from the OData spec. Not implemented.

A note on \$orderby

`$orderby` accepts only field references, not function expressions. You can't `$orderby=length(ProductName) desc` to find the products with the longest names. The order-by clause is restricted to bare field names — and a related restriction means it can't reference computed aliases either.

In practice, this means a class of "sort by computed value" queries has to be done client-side after fetching the relevant rows. The aggregated set that is returned is small, so the effort to sort client-side is minimal. Examples: "longest-name", "most-recent" by computed cutoff, or "highest-LineTotal" by computed value.

Checking your server version

You can confirm what you're working with by hitting `/$metadata` on your OData endpoint:

```
/fmi/odata/v4/0Data_Demo/$metadata
```

It returns the full schema definition for the hosted file — field names, types, relationships — and the service version. Claris continues to extend OData support in future FileMaker releases, so the working set described here will grow. The `/$metadata` endpoint is how you'll know what your specific server version supports.

Server-Side Aggregation: \$apply

If you've been writing FileMaker OData clients before now, you know the pattern that `$apply` replaces. Want a sum over a filtered set? Fetch the rows, reduce them client-side. Want a "top vendors by spend" report? Same answer. Pull hundreds of rows over the wire, reduce them locally, throw the rows away. It works, but when the answer is one number, a query that returns 800 rows is extremely inefficient.

`$apply` moves that work to the server. Three transformations matter on FileMaker 22: `aggregate`, `groupby`, and the multi-aggregate form of `aggregate`. The aggregation methods are `sum`, `min`, `max`, `average`, and `countdistinct`.

FILEMAKER 22 ONLY

`$apply` is brand new to FileMaker 22. Earlier versions don't have it at all. If you're on 21 or below, the workaround for everything in this section is the same: fetch raw rows with `$filter` and `$select`, and aggregate client-side.

The simplest case

The most basic possible aggregation is summing a single field. Total of `UnitPriceAtSale` across every order line in the database:

```
/fmi/odata/v4/OData_Demo/OrderLines?$apply=aggregate(UnitPriceAtSale with sum as TotalPrice)
```

RESPONSE

```
{ "value": [ { "TotalPrice": 2625741.40 } ] }
```

A few things to notice in that one query. First, the alias (`as TotalPrice`) is required. Drop it and you get a `-1002` parse error. Second, the alias name matters a lot — covered under "Reserved-word aliases" below. Third, `$select` doesn't apply here. The alias is response-only — once that clicked, a lot of other behaviors made sense. The agent had been thinking of OData query options as composable building blocks, and aliases break that assumption. They only produce response keys, not queryable fields.

The grouping case

`groupby` is what makes `$apply` interesting. It takes a list of dimensions and rolls the aggregate up over each one:

```
/fmi/odata/v4/0Data_Demo/Orders?$apply=groupby((ShippingMethod),aggregate(OrderID with countdistinct as OrderCount))
```

RESPONSE

```
{
  "value": [
    { "ShippingMethod": "Pickup",    "OrderCount": 1231 },
    { "ShippingMethod": "Overnight", "OrderCount": 1224 },
    { "ShippingMethod": "Ground",    "OrderCount": 1196 },
    { "ShippingMethod": "Express",    "OrderCount": 1177 },
    { "ShippingMethod": "Freight",    "OrderCount": 1172 }
  ]
}
```

Heads-up on runtime: this query takes around 14 seconds against the 6,000-row Orders table.

Tolerable for a one-off report, but worth knowing that `countdistinct` over a stable primary key is the expensive piece — covered under "When to use countdistinct" below.

ID AND ID ARE RESERVED WORDS

On FileMaker, `id` and `ID` are reserved words — if your schema uses one as a field name, the bare reference fails with `-1002 "syntax error in URL at: 'id'"`. The escape hatch is URL-encoded quotes: `%22id%22`. Our seed schema uses named primary keys (`OrderID`, `LineID`, etc.), so the rest of this guide doesn't run into this — but the next time you query a schema that does, you'll know what the error means.

Per the Claris OData docs, the aggregate functions explicitly supported on FileMaker are `sum`, `min`, `max`, `average`, and `countdistinct`. Bare `count` inside an aggregate isn't on that list. For counting rows inside an aggregation, `countdistinct` over a stable primary key is the supported and reliable form.

Worth being explicit about `$count=true`: it ignores `$top` and `$skip` entirely. So

`$filter=OrderStatus eq 'Cancelled'&$count=true&$top=5` returns 5 rows in the response and

`@count: 405` — the full count of the filtered set. That's why `$count=true&$top=1` is the reliable pattern for a cheap filtered count.

One thing I tried early on, just because it's shorter: a bare `groupby((dim))` with no aggregate clause, as a way to get distinct values. On FileMaker 22 that returns

8309 "All non-aggregated field references must be in the groupby clause." The fix is straightforward — include a trivial aggregate, like `countdistinct` on the primary key.

The multi-aggregate form

Here's a piece I haven't seen written up elsewhere in the community, and it's the one that changed how I write OData reports. You can compute multiple aggregates in a single `aggregate()` call by comma-separating them:

```
/fmi/odata/v4/0Data_Demo/OrderLines?$apply=aggregate(
  Quantity with sum as TotalUnits,
  UnitPriceAtSale with average as MeanUnitPrice,
  LineID with countdistinct as LineCount
)
```

RESPONSE

```
{
  "value": [{
    "TotalUnits": 34246,
    "MeanUnitPrice": 165.79,
    "LineCount": 15838
  }]
}
```

Three numbers, one query, one round trip — and guaranteed consistency. If you fetched these three numbers as three separate calls, you'd be at the mercy of any concurrent writes between them. Multi-aggregate gives you an atomic snapshot.

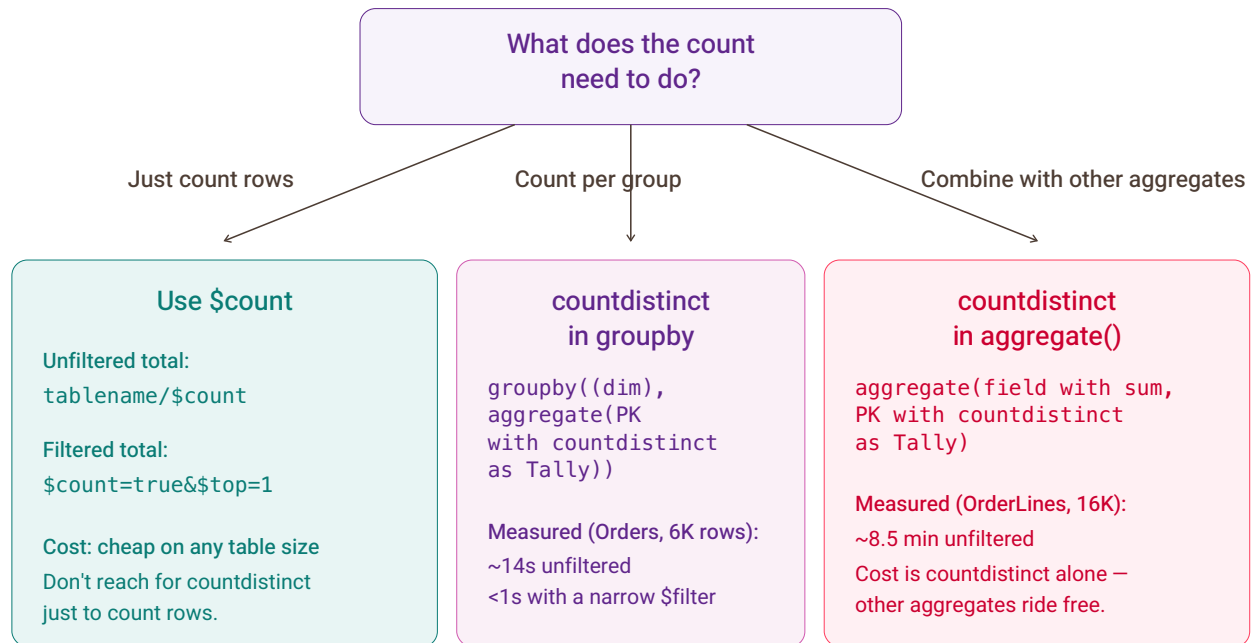
PERFORMANCE NOTE

On the demo server against 16,000 `OrderLines` rows, the multi-aggregate above took about eight and a half minutes to return. The cause is specific: `countdistinct` is the expensive piece. On the same table, `sum`, `average`, `min`, and `max` on bare fields each return in under two seconds. A four-way multi-aggregate using all of them takes 1.7 seconds total. Adding `countdistinct(LineID)` to that mix takes the total to eight and a half minutes.

When to use countdistinct

Counting rows on FileMaker 22 OData

countdistinct cost scales with rows scanned — pick the cheapest tool that fits the question



The pattern

It's not multi-aggregate that's slow, and it's not groupby that's slow. countdistinct over a large row set is slow. Filter first, group second, and the rest of the surface is cheap.

The performance section above traces the multi-aggregate slowness to one specific aggregator:

`countdistinct`. Three regimes, all measured against the seed data:

QUERY	ROWS SCANNED	TIME
Filtered subset, then groupby + <code>countdistinct</code>	~846 (Q4 Orders)	fast (<1s)
Unfiltered medium table + groupby (5 groups) + <code>countdistinct</code>	6,000 (Orders)	~14s
Unfiltered large table + multi-aggregate including <code>countdistinct</code>	16,000 (OrderLines)	~8.5 min

The decision rule:

- For "how many rows" questions, don't use `countdistinct` at all. Use `/Table/$count` for unfiltered totals or `?$filter=...&$count=true&$top=1` for filtered totals. Both are cheap. `countdistinct` for the same purpose can be hundreds of times slower.
- For row counts that need to ride alongside other aggregates — say, `aggregate(Quantity with sum, LineID with countdistinct as LineCount)` — `countdistinct` is the right tool, but be aware of the cost. If the table is large, expect minutes.
- Inside `groupby`, `countdistinct` is fine when the table is small or the filter narrows the scan. Avoid it on large unfiltered tables; consider a top-level `$count=true` instead and live with the separate call.

The pattern across this section: it's not multi-aggregate that's slow, and it's not `groupby` that's slow. `countdistinct` over a large row set is slow. Keep that one variable in mind and the rest of the surface is cheap.

Inline expressions and `cast()` inside `aggregate()`

The multi-aggregate form also expands what `aggregate()` accepts as input. It's not just bare fields — arithmetic expressions and `cast()` wrappers both work, and they compose with each other.

The simplest case is summing a text-stored numerical value. FileMaker schemas frequently have numbers stored as text. In our seed schema, `Products.MonthlyCostText` is exactly that. To sum it, plain `aggregate(MonthlyCostText with sum as TotalCost)` returns

8309 "Parameter number 1 to the function SUM is not of the correct type." Wrap it:

```
/fmi/odata/v4/0Data_Demo/Products?$apply=aggregate(cast(MonthlyCostText,Edm.Decimal) with sum as TotalCost)
```

Now it sums. This single capability eliminates workarounds that used to be required for any schema with text-stored numerical values.

The next step up: arithmetic expressions, with or without `cast()`.

`aggregate(Quantity mul UnitPriceAtSale with sum as TotalRev)` returns the actual revenue summed across all order lines, in one row, in a single call. The same form works with `average`, `min`, and `max`, and `cast()` can wrap the whole expression:

```
aggregate(cast(MonthlyCostText,Edm.Decimal) mul 12 with sum as TotalAnnualCost)
```

Expressions also work inside `groupby`. Per-product revenue across all order lines, in one call:

```
/fmi/odata/v4/0Data_Demo/OrderLines?$apply=groupby((ProductID),aggregate(Quantity mul UnitPriceAtSale with sum as TotalRev))
```

Mix and match: bare fields, `cast()`-wrapped fields, and arithmetic expressions can all coexist in the same multi-aggregate.

What this doesn't let you do: combine an inline expression in `aggregate()` with `$compute` to get both per-row computed values AND an aggregated answer in the same query. `$compute` and `$apply` don't compose at all on FileMaker 22. More on that in the Computed Fields section.

Reserved-word aliases

The `as <Alias>` slot in `aggregate()` treats the alias as a SQL identifier — and if it collides with a SQL reserved word, the server returns `8310 "internal data formatting error"`, a generic error that doesn't reveal the real cause.

Aliases that fail with `8310`: `Count`, `Sum`, `Avg`, `Max`, `Min`, `Value`, `Date`, `User`, `Table`, `Time`, `Select`, `Null`, `Where`, `Group`, `Order`, `Having`, `Between`, `Cast`, `Timestamp`, `From`, `Insert`, `Update`, `Delete`, `Join`, `Index`, `View`, `Schema`, `Primary`, `Foreign`, `Constraint`. The full Claris SQL reserved word list applies. It's case-insensitive, exact-match.

The escape hatch is to quote the alias:

```
/fmi/odata/v4/0Data_Demo/OrderLines?$apply=aggregate(UnitPriceAtSale with sum as %22Count%22)
```

The easier path is to not collide in the first place. Aliases like `Total`, `Subtotal`, `Tally`, `Rollup`, `TotalRevenue`, `TotalUnits`, `RecordCount`, `ActiveCount` are all safe and read better in the response. `Average` and `Countdistinct` pass — the block is specifically against single-token reserved words.

Filtering before aggregation

`$filter` and `$apply` are separate top-level query options, joined by `&`. The OData specification allows `filter()` as a transformation nested inside `$apply`, but that form parse-fails on FileMaker 22. The way to scope an aggregate to a subset is to add `$filter` as a separate parameter:

```
/fmi/odata/v4/OData_Demo/Orders?$filter=OrderDate ge 2025-10-01 and OrderDate le 2025-12-31&$apply=groupby((ShippingMethod),aggregate(OrderID with countdistinct as OrderCount))
```

That's our Q4 2025 order distribution by shipping method, server-aggregated. Against the seed data, that returns 846 total orders distributed across the five shipping methods — a clean five-row response.

DATE LITERAL SYNTAX

The date literals `2025-10-01` and `2025-12-31` above are unquoted. On a real Date field, that's correct. Quoting them would still work here. But on a text-stored date field the quoted form silently produces wrong results. Unquoted date literals on date filters, every time, regardless of whether quoting would also work. The Silent Failures section covers this in depth.

What \$apply will not do

A short list of things that look like they should work but don't, on FileMaker 22.0.6:

- **Bare `count` without `distinct`.** `aggregate(field with count as Tally)` parse-fails. Use `countdistinct` over a primary key.
- **The `/` chain operator between `$apply` transformations.** The oasis spec permits pipelines like `$apply=filter(...)/groupby(...)/aggregate(...)`, but the `/` itself parses as a path separator and the request fails. Multi-step pipelines get split into separate calls.
- **Sorting on an aggregate alias.** `$orderby=TotalRevenue desc` returns `8309 "field does not exist"` because the alias is response-only. To get the largest-N response, fetch the full aggregated set and sort client-side. The aggregated set is small by definition.
- **`compute()` as a transformation inside `$apply`.** The working mechanism on FileMaker 22 is the top-level `$compute` query option, covered in the next section.

There's also `nest`, `addnested`, `join`, and `outerjoin`, which appear in the FileMaker OData specification. I haven't been able to find a working form for any of them on 22.0.6.

Response size and paging

FileMaker OData returns a maximum of 10,000 rows per response. Server-side aggregation via `$apply` is generally the right tool when you want a summary — the server does the work across all matching rows and returns only the result. Grouping 16,000 order lines by shipping method returns 5 rows, well under the limit, regardless of how many rows the server processed. The limit applies to what comes back, not what gets computed.

Where aggregation can still run into the limit: a `groupby` on a high-cardinality field. Grouping 16,000 order lines by `LineID` would produce 16,000 result rows and hit the ceiling.

When results exceed 10,000, FileMaker includes a `@odata.nextLink` in the response containing the URL for the next page. Follow it until no `nextLink` appears. The manual paging pattern with `$top` and `$skip`:

```
/fmi/odata/v4/OData_Demo/OrderLines?$top=10000&$skip=0  
/fmi/odata/v4/OData_Demo/OrderLines?$top=10000&$skip=10000  
/fmi/odata/v4/OData_Demo/OrderLines?$top=10000&$skip=20000
```

Each call moves the window forward by the page size until the response contains fewer than `$top` rows and no `nextLink`.

"If you've been writing FileMaker OData clients that fetch rows and reduce them client-side, FileMaker 22 lets you push a substantial amount of that work to the server. Most of the queries I used to ship that returned hundreds of rows now return one row, or one row per group, in a single call."

Computed Fields: \$compute

`$compute` is a top-level OData 4.01 query option that adds a computed property to each row in the response. On FileMaker 22, it's a top-level query option joined to the rest of the querystring with `&`, just like `$filter` or `$select`.

The classic use case is producing a line total. Our `OrderLines` table has `Quantity` and `UnitPriceAtSale` but no precomputed line total. With `$compute`:

```
/fmi/odata/v4/OData_Demo/OrderLines?$compute=Quantity mul UnitPriceAtSale as  
LineTotal&$select=LineID,Quantity,UnitPriceAtSale&$top=5
```

RESPONSE

```
{  
  "value": [  
    { "LineID": "LINE-...", "Quantity": 10, "UnitPriceAtSale": 632.89, "LineTotal": 6328.90 },  
    ...  
  ]  
}
```

Three things to notice. First, the alias is added to the response automatically — `LineTotal` should not appear in `$select`. Aliases generated by `$compute` are response-only. They can't be referenced in `$select`, `$filter`, or `$orderby` on the same request. To filter or sort by the computed value, fetch the rows and process them client-side.

Second, the arithmetic operators are OData-style words rather than symbols: `add`, `sub`, `mul`, `div`, `mod`. So `Quantity * UnitPriceAtSale` is `Quantity mul UnitPriceAtSale`. The same operators work directly inside `$filter` predicates: `Quantity mul UnitPriceAtSale gt 1000`.

Third, `cast()` is supported inside `$compute`. If you have a text-stored numerical value and you want it typed in the response without changing the schema:

```
/fmi/odata/v4/OData_Demo/Products?$compute=cast(MonthlyCostText,Edm.Decimal) as  
MonthlyCost&$select=ProductID,ProductName,MonthlyCostText&$top=10
```

You can have multiple computes in one call by comma-separating them:

```
/fmi/odata/v4/OData_Demo/OrderLines?$compute=Quantity mul UnitPriceAtSale as LineTotal,UnitPriceAtSale mul DiscountPct div 100 as DiscountAmount
```

The canonical OData functions from the Function Library section work inside `$compute` the same way they work inside `$filter`: `concat(FirstName,LastName) as FullName`, `tolower(Email) as EmailLower`, `length(ProductName) as NameLength`.

\$compute and \$apply don't compose

Putting `$compute` and `$apply` in the same querystring fails. With or without groupby, in either order, with or without overlap between the computed alias and the aggregated fields, the server returns `8309`. The most common error message —

"All non-aggregated field references must be in the groupby clause" — is technically correct but unhelpful: it doesn't say the underlying problem is that the two options don't compose.

The practical implication: if you want both per-row computed values AND an aggregated answer in one round trip, you can't have it. Two queries, two round trips — or, for cases where the aggregate just needs the computed expression and a separate per-row response isn't required, use the inline-expression-in-aggregate form from the Aggregation section

```
( aggregate(Quantity mul UnitPriceAtSale with sum as Rev) ).
```

Joining Tables

Between the function library and `$apply`, we can already compute per-product revenue using `$apply=groupby((ProductID),aggregate(Quantity mul UnitPriceAtSale with sum as Rev))`. The piece we can't get without leaving `OrderLines` is the category — that lives on the `Products` table. So the question "what's our revenue by product category?" is fundamentally cross-table.

A subtle thing about the cross-table version: once you've joined `OrderLines` to `Products` via `$expand` or `$crossjoin`, you can't aggregate the joined result with `$apply`. So for "revenue by category" specifically, you end up doing the aggregation client-side over the joined rows. That's where `$compute` earns its keep — adding `Quantity mul UnitPriceAtSale as LineRev` to the join query gives every joined row a precomputed revenue, and the client-side group-by-Category becomes a one-liner instead of a fetch-and-multiply pass.

\$expand — for tables related in the graph

OData's standard mechanism for including data from a related entity is `$expand`. On FileMaker, it works when the FileMaker relationship graph defines a relationship between the two tables.

Sharing a logical join key isn't enough — there has to be a real relationship in the graph.

When the relationship exists, `$expand` is straightforward. Our `Orders` table relates to `Customers` through `CustomerID`. To get the customer's company name and region alongside each order:

```
/fmi/odata/v4/OData_Demo/Orders?
$expand=Customers($select=CompanyName,Region)&$select=OrderID,OrderDate,OrderStatus&$top=3
```

RESPONSE

```
{
  "value": [
    {
      "OrderID": "ORD-...",
      "OrderDate": "2026-03-15",
      "OrderStatus": "Delivered",
      "Customers": [
        { "CompanyName": "Northwind LLC", "Region": "NA-East" }
      ]
    },
    ...
  ]
}
```

Two oddities worth knowing about. The first is that one-to-one relationships come back wrapped in a single-element array. The customer of an order is logically singular, but the response gives you `"Customers": [{ ... }]`. Not a typo — it's the OData expansion shape applied uniformly. Plan for it when parsing.

The second is that `$top` and `$count=true` are not supported inside `$expand`. Both return `-1002`. The full related set is always fetched. For high-fanout relationships you can't paginate the child side from inside the parent query — you'd fetch parent and child as separate queries with explicit `$filter` scoping on the child side.

When the relationship doesn't exist in the graph, `$expand` returns:

```
-1019 "No relationship exists between tables 'A' and 'B'"
```

There's also a near neighbor worth knowing: `-1021 "Table 'X' not defined in database"`. That one means the navigation property name is wrong — a typo or a casing mismatch — not that the relationship is missing. `-1021` is "the name is wrong"; `-1019` is "the name is right but no FM relationship exists."

\$crossjoin — for tables not in the graph

The job `$crossjoin` is built for: you have two tables that share a logical join key but aren't related in the FileMaker relationship graph. `$expand` won't work — it requires a graph relationship. Editing the graph isn't always an option, especially on schemas you don't own. `$crossjoin` lets you do the join server-side anyway.

The piece that's easy to miss the first time through is that `$crossjoin` is a path segment, not a query parameter. It sits where the table name normally goes in the URL, with the tables you want to join listed inside the parentheses.

Here's the shape, joining `OrderLines` to `Products` to get product details on each order line:

```
/fmi/odata/v4/OData_Demo/$crossjoin(OrderLines,Products)?$filter=OrderLines/ProductID eq Products/ProductID&$expand=OrderLines($select=Quantity,UnitPriceAtSale),Products($select=Category,ProductName)&$top=3
```

Three things going on in that URL. First, `$crossjoin(OrderLines,Products)` replaces the table-name slot. Second, the join condition `OrderLines/ProductID eq Products/ProductID` lives inside `$filter`. Third, `$expand=OrderLines($select=...),Products($select=...)` projects the fields you want from each side.

RESPONSE

```
{
  "value": [
    {
      "Quantity": 1,
      "UnitPriceAtSale": 392.14,
      "Category": "Outdoor",
      "ProductName": "Modern Stove"
    },
    ...
  ]
}
```

Each row in the response represents one matched pair across the join, with the projected fields from both sides flattened together. Run that against the seed data without `$top=3` and you get 15,838 rows — one per order line that has a matching product.

Three-way joins work the same way. Cross-join `Orders`, `OrderLines`, and `Products`, with the join conditions and a business filter all in the same compound `$filter`:

```
/fmi/odata/v4/0Data_Demo/$crossjoin(Orders,OrderLines,Products)?$filter=Orders/OrderID eq OrderLines/OrderID and OrderLines/ProductID eq Products/ProductID and Orders/OrderStatus eq 'Delivered'&$expand=Orders($select=OrderID,OrderStatus),OrderLines($select=Quantity),Products($select=Category)&$top=3
```

A few rules for cross-join queries that aren't obvious:

`$apply` **doesn't work with** `$crossjoin`. Combining the two returns `-1002 "syntax error in URL at: '/'"`. Server-side aggregation across a cross-join isn't supported on FileMaker 22. If you need a rolled-up answer, project the joined rows with `$expand($select=...)` and aggregate client-side.

Field-name collisions in `$expand` selections. If both sides of the cross-join select the same field name, the response will contain literal duplicate JSON keys. Most parsers — Python's `json.loads`, JavaScript's `JSON.parse` — silently ignore all but the last occurrence. You lose data without knowing it. Pick non-overlapping `$select` lists per side, or rename one side's projection if you need both values.

The same collision applies if you omit `$expand($select=...)` entirely. Without it, every field from every table appears flat in the response — including key fields that may be shared — and any name shared across the joined tables collapses by the same last-wins rule. Explicitly projecting with `$expand($select=...)` is the practical safeguard.

Type-mismatched join keys need `cast()`. If one side of the join key is numeric and the other is text, wrap one side: `cast(OrderLines/ProductID,Edm.String) eq Products/ProductID`.

Field references must be table-prefixed when the field exists on both sides. Anything that lives on multiple tables in the join needs to be qualified, or the engine returns `8309 "cannot determine the table associated with field 'X'"`. When the field is unique to one table, the engine auto-disambiguates.

A real example: vendor spend with the apostrophe trap

Cross-joining `Vendors` with `VendorPayments` to get every payment with vendor details attached:

```
/fmi/odata/v4/0Data_Demo/$crossjoin(Vendors,VendorPayments)?$filter=Vendors/VendorID eq VendorPayments/  
VendorID and Vendors/Active eq  
1&$expand=Vendors($select=VendorName),VendorPayments($select=AmountPaid,PaymentDate,PaymentReference)&$  
top=10
```

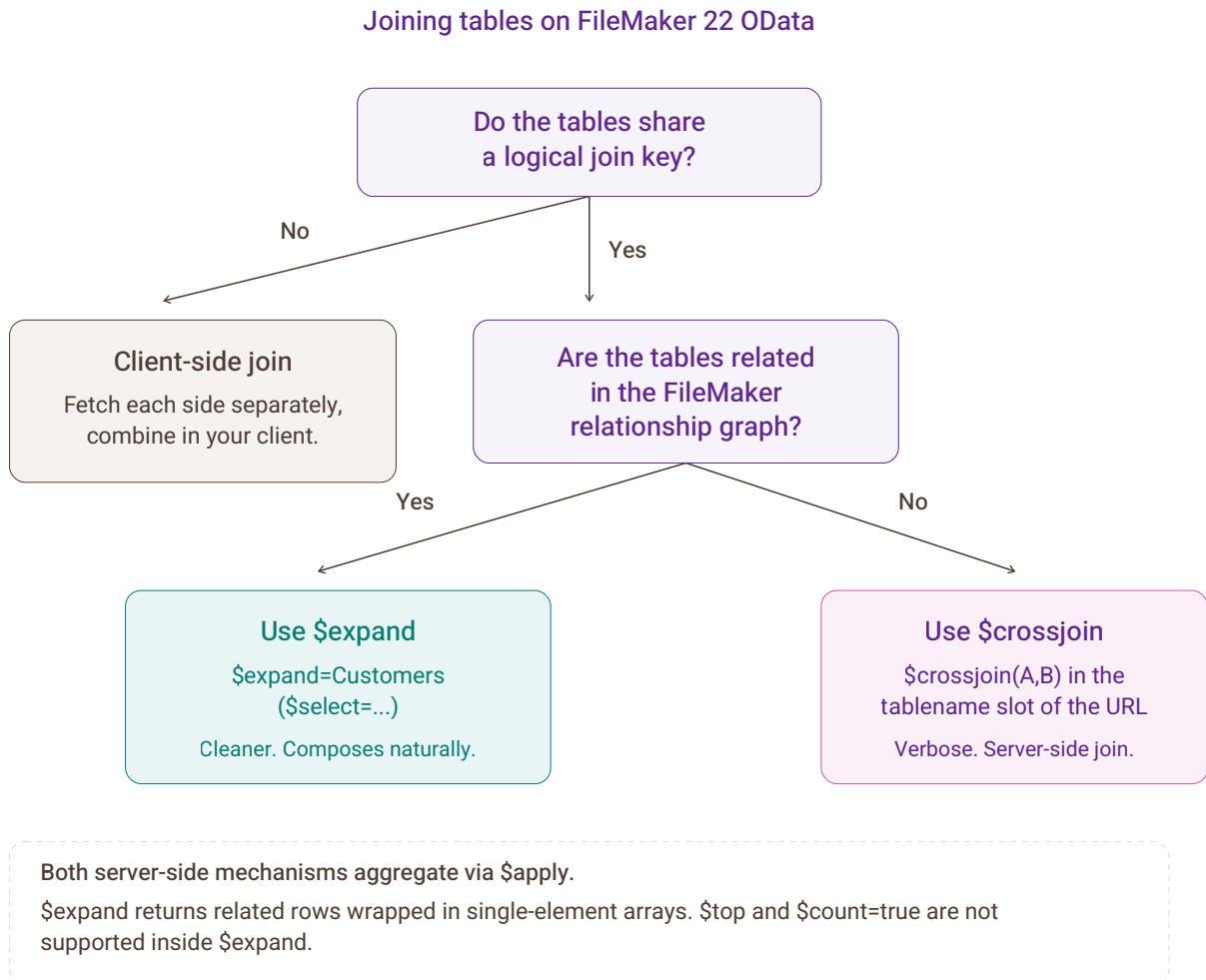
Active-vendor payment rows in the seed data: 3,374. The response gives you `VendorName`, `AmountPaid`, `PaymentDate`, and `PaymentReference` per row, ready to roll up client-side into vendor totals, monthly summaries, or whatever shape the report needs.

One detail worth pointing out: if your literal string contains an apostrophe — say, a customer named O'Brien — the way to escape it is SQL-style doubling, not URL encoding:

```
contains(OrderNotes,'O''Brien')
```

That's two consecutive single quotes. The URL-encoded form (`%27`) is silently dropped from the search pattern at decode time — `contains(OrderNotes,'O%27Brien')` returns the rows that contain `OBrien`, with no error and no warning. This is covered at length in the Silent Failures section.

Choosing between \$expand and \$crossjoin



If the tables are related in the FileMaker relationship graph, prefer `$expand`. The syntax is cleaner, the response shape is more natural, and it composes with `$filter` and `$select` the way you'd expect.

If they're not — but they share a logical join key — use `$crossjoin`. It's more verbose, but it lets you do server-side joins without having to first edit the relationship graph. For schemas you don't own, that matters.

If they're not, and they don't share a logical key, you can't join them on the server. Fetch each side and combine client-side.

A note on performance: `$crossjoin` and `$expand` are both doing the work server-side, and I have not seen significant performance differences between them. The cost is in the response payload, and `$crossjoin` is sometimes less efficient there because the response is wider. Use `$select` projections inside `$expand($select=...)` blocks to keep the payload tight.

When Queries Fail Quietly

A loud failure is a gift. The server tells you something is wrong, you read the error, you fix the bug. Most of the OData errors on FileMaker 22 are loud — the parse-level `-1002`, the field-or-type `8309`, the formatting `8310`, the relationship `-1019` and `-1021`. These have specific causes and specific cures.

This section is about the other kind: queries where the server returns `200 OK`, the JSON parses cleanly, and the answer is wrong. I started writing these down because I was teaching an AI agent to query FileMaker, and an agent will not notice that 5,419 is suspiciously close to but not equal to 5,595. It will simply use the number it got. A human developer might glance at a result and think "wait, that's a few hundred short of what I expected" and start digging. Agents don't have that intuition, and increasingly, neither do the integrations we're all building.

FileMaker 22 OData — silent failures

Seven patterns where the server returns 200 OK but the answer is wrong

1. `ne 'X'` excludes nulls

Wrong: `$filter=Status ne 'Cancelled'`

Returns rows where Status is non-null and not 'Cancelled'. Null rows are silently excluded.

Right: `Status ne 'Cancelled' or Status eq null`

Three-valued logic: null is neither equal nor unequal — name the null case explicitly.

2. Lambda slash-form drops the predicate

Wrong: `tags/any(t:t eq 'X')`

Parses fine, predicate is dropped, returns every row in the table. Equivalent to `$filter=true`.

Right: `contains(field, 'X')`

Use `contains()` for membership tests on list/JSON-encoded fields.

3. URL-encoded apostrophe is silently dropped

Wrong: `contains(name, 'O%27Brien')`

%27 is stripped at decode time. Pattern becomes 'OBrien'. Returns zero rows on real data.

Right: `contains(name, 'O' 'Brien')`

SQL-style doubling — two single quotes, not %27. Same trick for any string literal.

4. Quoted date literal falls back to string compare

Wrong: `cast(d, Edm.Date) ge '2026-01-01'`

Falls back to lexical comparison. Returns plausible but wrong rows on text-stored date fields.

Right: `cast(d, Edm.Date) ge 2026-01-01`

Drop the quotes. Unquoted literal is treated as an OData temporal value.

5. `\$count=true` with `\$top=0` returns zero

Wrong: `$count=true&$top=0`

Returns @count: 0 regardless of filter. The cheap-row-count idiom is broken on FileMaker 22.

Right: `$count=true&$top=1`

Or use the path-based form ``tablename/$count`` for unfiltered totals.

6. Misspelled query option is silently dropped

Wrong: `$flter=Status eq 'X'`

Unrecognized options are silently dropped. The filter is ignored. Full table is returned.

Right: `$filter=Status eq 'X'`

Keep query options in a constant or enum. Don't type them as bare strings.

7. `contains(field, '')` matches every row

Wrong: `contains(name, '')`

Every string contains the empty string. Returns every row. Looks filter-shaped, isn't.

ne and three-valued logic

This one is different from the rest. The server isn't doing anything wrong — it's doing exactly what the query says. The gap is between what a developer usually means by "not cancelled" and what `ne 'Cancelled'` actually asks for in SQL three-valued logic.

Consider a query for "all orders that are not cancelled":

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=OrderStatus ne 'Cancelled'&$count=true&$top=1
```

That returns 5,419 rows against our seed data. The total order count is 6,000, and there are 405 cancelled orders, so the expected result is 5,595 — but it comes back 176 short. The 176 missing rows are orders with a null `OrderStatus`. In SQL three-valued logic, `null ne 'Cancelled'` evaluates to `null`, not `true`, so those rows fall out of the result.

The fix is explicit:

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=OrderStatus ne 'Cancelled' or OrderStatus eq  
null&$count=true&$top=1  
→ { "@count": 5595 }
```

Whether you want the null-inclusive or null-exclusive count is a business decision — but it should be a deliberate decision. The default behavior of `ne` will quietly make it for you. This is at the top of the list because it's the trip hazard you'll hit most often in real integrations — and because experienced FileMaker developers, whose intuitions are shaped by FileMaker's own find behavior (which doesn't have this issue), are the ones least prepared for it.

Lambda operators silently dropped

The Claris OData guide lists lambda operators (`any` and `all`) among the unsupported features on FileMaker. What's worth flagging is the way "unsupported" surfaces, because it's not consistent. Most unsupported features fail loudly. Lambda operators are the exception: the query parses, executes, returns `200 OK`, and the predicate is silently dropped.

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=OrderNotes/any(n:n eq 'urgent')&$count=true&$top=1
```

Against our seed data, this returns `@count: 6000`. The predicate is gone. There's no error, no warning, no hint in the response that the filter was ignored.

The workaround is to use `contains()` for membership tests:

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=contains(OrderNotes,'urgent')
```

URL-encoded apostrophes vanish from literals

If your filter literal contains an apostrophe — `O'Brien`, `O'Reilly`, `D'Angelo` — the SQL-standard way to escape it inside a single-quoted string is to double it:

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=contains(OrderNotes,'O''Brien')
```

That's two consecutive single quotes. Against our seed data, it returns 80 rows, which is the actual count of orders with `O'Brien` in their notes.

The wrong way is to URL-encode the apostrophe as `%27`:

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=contains(OrderNotes,'O%27Brien')
```

This parses successfully, executes successfully, and silently drops the `%27` from the search pattern at decode time. The query becomes `contains(OrderNotes,'OBrien')`, which matches zero rows. No error, no warning. The user concludes there are no O'Brien orders in the system. There are 80.

Note the asymmetry: `&` does not need to be URL-encoded inside a single-quoted string literal — the server treats it as a literal `&` correctly. So defensive `%26` encoding is harmless but unnecessary, while defensive `%27` encoding is actively wrong.

Quoted date literals on text-stored fields

Some FileMaker schemas store dates as text, in YYYY-MM-DD format. In our seed schema, `Orders.NextBillDateText` is exactly this. To find the orders whose next bill falls in January 2026, the correct OData filter is:

```
/fmi/odata/v4/OData_Demo/Orders?$filter=cast(NextBillDateText,Edm.Date) ge 2026-01-01 and  
cast(NextBillDateText,Edm.Date) lt 2026-02-01
```

This returns 268 rows, which is the actual answer. Now write what looks like the same query but with quoted date literals — the form a developer trained on SQL will write almost reflexively:

```
/fmi/odata/v4/OData_Demo/Orders?$filter=cast(NextBillDateText,Edm.Date) ge '2026-01-01' and  
cast(NextBillDateText,Edm.Date) lt '2026-02-01'
```

The single-quoted form succeeds, returns rows, and the rows it returns are wrong. Worse, the count is plausible — it's not zero, it's not the full table, it's some non-empty number that looks like it could be a January's worth of bills.

For real Date and Timestamp fields this trap doesn't exist in the same way — both quoted and unquoted forms are accepted equivalently. But for text-stored date fields, the unquoted form is the only correct one. The discipline rule I now apply: unquoted date literals on date filters, every time, regardless of underlying field type.

\$count=true&\$top=0 always returns zero

The cross-OData habit for "just give me the count, don't pay for any rows" is to set `$top=0`:

```
/fmi/odata/v4/OData_Demo/Orders?$filter=OrderStatus eq 'Cancelled'&$count=true&$top=0  
→ { "@count": 0, "value": [] }
```

That zero is wrong. There are 405 cancelled orders. `$count=true` combined with `$top=0` returns `@count: 0` regardless of the filter. Two workarounds. The first: set `$top=1` instead:

```
/fmi/odata/v4/OData_Demo/Orders?$filter=OrderStatus eq 'Cancelled'&$count=true&$top=1  
→ { "@count": 405, "value": [...1 row...] }
```

The second is the path-based count form, which returns the unfiltered table total as a bare integer and doesn't respect `$filter`:

```
/fmi/odata/v4/OData_Demo/Orders/$count  
→ 6000
```

A related issue: `$count=true` on `$apply` queries returns `@count: 1` regardless of how many groups are produced. To count groups, count the response array client-side.

Misspelled query options silently dropped

Type a query option name slightly wrong — `$filter` instead of `$filter`, `$slect` instead of `$select` — and the server silently drops the entire option. The query executes against an empty filter and returns the full table:

```
/fmi/odata/v4/0Data_Demo/Orders?$filter=OrderStatus eq 'Cancelled'&$top=5  
→ 5 random orders, filter completely ignored
```

There's no warning, no `400 Bad Request`, no log entry visible to the client. The unrecognized option just disappears. The same failure mode applies to parameters without the `$` prefix, and to duplicate query options — `$top=2&$top=5` silently uses the first value.

Empty-string equality is a parse error

Testing whether a field is the empty string seems like it should be `field eq ''`. On FileMaker 22, that returns `-1002 "syntax error in URL at: ''"`. The empty-string literal isn't parseable as a value.

Three cases you might want to distinguish for a text field:

- The field has no value at all: `field eq null`
- The field has a value, but it's the empty string: `length(field) eq 0`
- The field has a non-empty value: `field ne null and length(field) gt 0`

These are three distinct states in FileMaker. `null` and "empty string" aren't the same thing on the storage side, and they don't filter the same way.

Booleans stored as 'true'/'false' text

In our seed schema, `Products.Discontinued` is text-stored as the literal strings `'true'` and `'false'`. There are 71 discontinued products and 379 active ones. The natural-feeling filter:

```
/fmi/odata/v4/0Data_Demo/Products?$filter=Discontinued eq true
```

Unquoted, that returns `8309 "field 'true' does not exist"`. The bareword `true` is parsed as a field name. So you reach for the cast:

```
/fmi/odata/v4/0Data_Demo/Products?$filter=cast(Discontinued,Edm.Boolean) eq 'true'
```

That returns `8309 "An expression contains incompatible data types"`. The cast to `Edm.Boolean` doesn't work for text-stored values. The form that does work is plain string equality:

```
/fmi/odata/v4/0Data_Demo/Products?$filter=Discontinued eq 'true'  
→ 71 rows
```

Treat the field as what it actually is — a text field whose values happen to be the strings `'true'` and `'false'` — and the cast goes away. The lesson generalizes: reach for `cast()` when the server actively asks for it. Don't reach for it preemptively.

The sanity test

What ties all of these together is that the server's "this query is wrong" signal isn't reliable. Some wrong queries are loud, but a non-trivial fraction are silent. The discipline I've landed on, and that I now apply to every non-trivial query:

"Before running a query, write down what answer you expect — at least to an order of magnitude. After running it, reconcile."

For an aggregate, the rule is more specific: if you're computing parts of a whole — totals by region, totals by category, totals by status — sum the parts and check them against the unfiltered total. If they don't reconcile, the difference points at a silent drop. The most likely causes are the ones above: a `ne` excluding nulls, a lambda predicate dropped, an option name typo'd, a quoted date literal falling back to string comparison.

For a list query, the rule is: have an expected order of magnitude. If you asked for "orders in January 2026" and got 6,000 rows back, the filter didn't apply. The most useful number to know in advance is the table total — `tablename/$count` — because every other answer is bounded by it.

FileMaker 22 has a genuinely good OData surface. Server-side aggregation, computed fields, cross-table joins, and a meaningful slice of the canonical OData function set all work, and they work in combinations. Compared to the 21.x and earlier versions — which had `$filter`, `$select`, `$orderby`, and not much else — it's a substantial step forward. The framing of this final section is that the OData functionality is vast, and so are the possibilities for silent failures. The way to respond is to write everything down, and to keep adding to it whenever a new behavior surfaces.

Download Resources

Every example in this guide was run against the following resources. Download them to follow along, reproduce any query, or use as a starting point for your own OData project.



Postman Collection

Every URL in this guide, organized by section, ready to run against your own server. You'll need to set the `hostname`, `filename`, `username`, and `password` variables before running.

[OData_Demo.postman_collection.json](#)



FileMaker Database

The `OData_Demo.fmp12` file pre-populated with the seed data. Host it on FileMaker Server 22 and every query in this guide will work against it directly.

[OData_Demo.fmp12](#)



Seed Data CSVs

Six CSV files — Customers, Products, Orders, OrderLines, Vendors, VendorPayments — plus import notes. Use these to re-populate the database or build your own schema from scratch.

[Seed data folder →](#)



This Guide (PDF)

A print-ready PDF of this guide for offline reference, annotation, or sharing with your team.

[odata-guide.pdf](#)

If you find a behavior not covered here — a function that works on your server but isn't in the verified-working list, a silent-failure mode that isn't in this catalog, a surprise on a different server version — reach out. We'd rather find out from a peer than from an agent silently producing the wrong answer six months from now.