# Realtime Metric Filters Design Document

## Summary

Realtime metrics , driven by the nearline tier, is the forcing factor to create a comprehensive filter-builder component, however, should be made to be generally applicable to Commit data anywhere in the Commit and admin apps. The defined data structure and desired desired flexibility of the controls present a technical design challenge. The dataset on which to filter is dynamic, varying by customer, resulting in a implementation that must consider broad types of use-cases, data types and complexity. In order to satisfy these requirements, the component must also be dynamic, grow based on a user's needs and ensure the resulting data matches user expectations.

## Scope

While realtime metrics are the main driving product need, the design and implementation need to be flexible enough to be used throughout the Commit app to provide a seamless user experience. This may include filtering pipeline deals as well as within areas in the Commit admin app. This document focuses on only the design and implementation to support the use case of realtime metrics.

- **Data structure**
  The Commit backend team is handling the design and architecture of the realtime metrics service and thus control the shape of the data delivered to the client-side app. This scope and document covers the client side adaption to this data model.

- **UX and design**
  Given the complexity and requirement to adhere to and operate within the constraints of the data delivered to the UI, the initial concepts, flow, and structure of the filter controls are handed to the UI engineer responsible for delivering the completed component. Design concepts and iterations will undergo frequent review and critique with Commit's UX team to ensure a cohesive feel and adherence to existing design formalisms.

- **Client-side rendering**
  The filter component will be rendered within the Commit React app, with data delivered via our GRPC API. The filter feature will be comprised of multiple separate and independent components, within a context responsible for saving client-side changes as well as syncing with the backend service.

## Requirements

- **Use of boolean logic**

  In order to define filtering logic capable of providing accurate and relevant information to the customer, multiple filters must be applied and aggregated using boolean logic of AND or OR operators.

  **OR Operator**

  Performs the filter combination in an inclusive way, where records are included if only one of the criteria is satisfied. As more criteria are added to the query, the more expansive the result set.

**AND Operator**
Results are exclusive, where the records must match all of the criteria, resulting in a narrower result set. Filters applied without the explicit use of a boolean operator risk confusion and unexpected results because the user may not understand how the logic is applied.

It is of important note that the AND/OR operators be used in a mutually exclusive way. Two filters may be only be combined with one or the other but not both. When three or more filters are combined, they *must* all use the same operator. The reason for this is due to the order of operations when executing the query.

For example, when combining abstract filters A, B and C using AND and OR operators, it is clear there is room for interpretation between the service executing the query and the user's intention.

```
A or B and C
```

*Figure 1*

Even in this simple example, the logic may be evaluated in two different ways, demonstrated using grouping characters *[ ]* where the innermost brackets are evaluated first, working outward.

```
[ A or B] and [ C ]  or alternatively [A] or [ B and C ]
```

*Figure 2*

It is not clear what the user intended and without the use of grouping, the service cannot reliably deliver the expected results. Therefore, client-side logic must be applied to ensure a mixed-modality query is not executed by the backend service. This may be enforced using a combination of error states, hiding/removal of options, or UI actions being made on behalf of the user.

- **Support for multiple/various data types**

  The data on which fillers will be applied may vary in type, including:

  - *String*
  - *Integer*
  - *Floating point number*
  - *Boolean*
  - *Date*
  - *Timestamp*
  - *Raw (base64/binary data)*
  - *Array( of types defined )*
  - *Object ( n-number of key/value pairs )*

  The data-type is defined by the backend service and both the client and service must have agreement on the options of possible types. Unknown types inherently cannot be compared. If additional types are desired, changes to both the service and client will be required.

- **Data comparators and relevant UI control surfaces**

  While the combination of filter rules are limited to that of AND and OR, data of a specified type may be compared with a limited set of comparator operators. Here again, both the client and backend service must have prior agreement on supported operators. The following comparators should be supported:

```
EQUAL_TO
NOT_EQUAL_TO
```

```
LESS_THAN
GREATER_THAN
GREATER_THAN_EQUAL_TO
LESS_THAN_EQUAL_TO
BETWEEN
NOT_BETWEEN
IN
NOT_IN
REGEX
NOT_REGEX
```

It is important to note that not all comparators can be used with all data types listed above. For example boolean data can not be compared using the LESS/GREATER THAN operators. Even if the server-side logic is able to perform the operation, it would be an unpredictable outcome to the user. Because of this, the data types must be mapped to the possible comparators.

```
const UNSPECIFIED = {
    key: FieldType.FIELD_TYPE_UNSPECIFIED,
    comparisonOps: []
  }

  const STRING = {
    key: FieldType.FIELD_TYPE_STRING,
    comparisonOps: [
      operators.EQUAL_TO,
      operators.NOT_EQUAL_TO,
      operators.IN,
      operators.NOT_IN,
      operators.REGEX,
      operators.NOT_REGEX
    ]
  }

  const INT = {
    key: FieldType.FIELD_TYPE_INT,
    comparisonOps: [
      operators.EQUAL_TO,
      operators.NOT_EQUAL_TO,
      operators.LESS_THAN,
      operators.GREATER_THAN,
      operators.GREATER_THAN_EQUAL_TO,
      operators.LESS_THAN_EQUAL_TO,
      operators.BETWEEN,
      operators.NOT_BETWEEN
    ]
  }

  const FLOAT = {
    key: FieldType.FIELD_TYPE_FLOAT,
    comparisonOps: [
      operators.EQUAL_TO,
      operators.NOT_EQUAL_TO,
      operators.LESS_THAN,
      operators.GREATER_THAN,
      operators.GREATER_THAN_EQUAL_TO,
      operators.LESS_THAN_EQUAL_TO,
      operators.BETWEEN,
      operators.NOT_BETWEEN
    ]
  }

  const BOOL = {
    key: FieldType.FIELD_TYPE_BOOL,
    options: [{
      label: 'True',
      value: 'True'
    },
    {
      label: 'False',
      value: 'False'
    }],
    comparisonOps: [
      operators.EQUAL_TO,
      operators.NOT_EQUAL_TO
    ]
  }

  const DATE = {
    key: FieldType.FIELD_TYPE_DATE,
    comparisonOps: [
      operators.EQUAL_TO,
      operators.NOT_EQUAL_TO,
      operators.LESS_THAN,
      operators.GREATER_THAN,
      operators.GREATER_THAN_EQUAL_TO,
      operators.LESS_THAN_EQUAL_TO,
      operators.BETWEEN,
      operators.NOT_BETWEEN
    ]
  }

  const TIMESTAMP = {
    key: FieldType.FIELD_TYPE_TIMESTAMP,
    comparisonOps: [
      operators.EQUAL_TO,
      operators.NOT_EQUAL_TO,
      operators.LESS_THAN,
      operators.GREATER_THAN,
      operators.GREATER_THAN_EQUAL_TO,
      operators.LESS_THAN_EQUAL_TO,
      operators.BETWEEN,
      operators.NOT_BETWEEN
    ]
  }

  const RAW = {
    key: FieldType.FIELD_TYPE_RAW,
    comparisonOps: []
  }

  const ARRAY = {
    key: FieldType.FIELD_TYPE_ARRAY,
    comparisonOps: [
      operators.IN,
      operators.NOT_IN,
    ]
  }

  const OBJECT = {
    key: FieldType.FIELD_TYPE_OBJECT,
    comparisonOps: []
  }
```

Each data type must be able to be displayed in a UI form control. Generally, the type of data dictates the form control, however some types may be used with multiple controls.

The input types are defined:

- *INPUT* - A single value input control
- *MULTI_INPUT - An input capable to accepting multiple user-defined values*
- *SELECT - Displays defined options, only one selectable*
- *MULTI_SELECT - Displays defined options, multiple selections*
- *RANGE - Two singular input types defining a starting and end point of the data type*

Comparator operators dictate the available input controls for a given data type as some combinations are not logical. For example a MULTI_INPUT would not be applicable when using a LESS/GREATER THAN comparator.

Finally data types can optionally have multiple format types. Possible format types are as listed:

```
currency
int
percent
email
plaintextarea
richtextarea
select
text
url
date
timestamp
float
location
```

The format type is defined during the data-mapping stage in the admin app. The format types can be grouped by input types, and is used when the UI control is ultimately rendered. *Note the location type is defined by the service however won’t be implemented on the client at this time.

- **Be generic to allow for other/future use-cases**

  The filter control is required to configure the realtime metrics, however should be developed in a flexible way to be implemented in other areas of the commit app.

- **UX requirements**
  - Support simple to very complex logic
    Most often, only a few filters may need to be applied to generate the desired result however a user may need to apply many filters using a combination of logical operators to achieve the desired result. The control must allow for both the simple use-case without

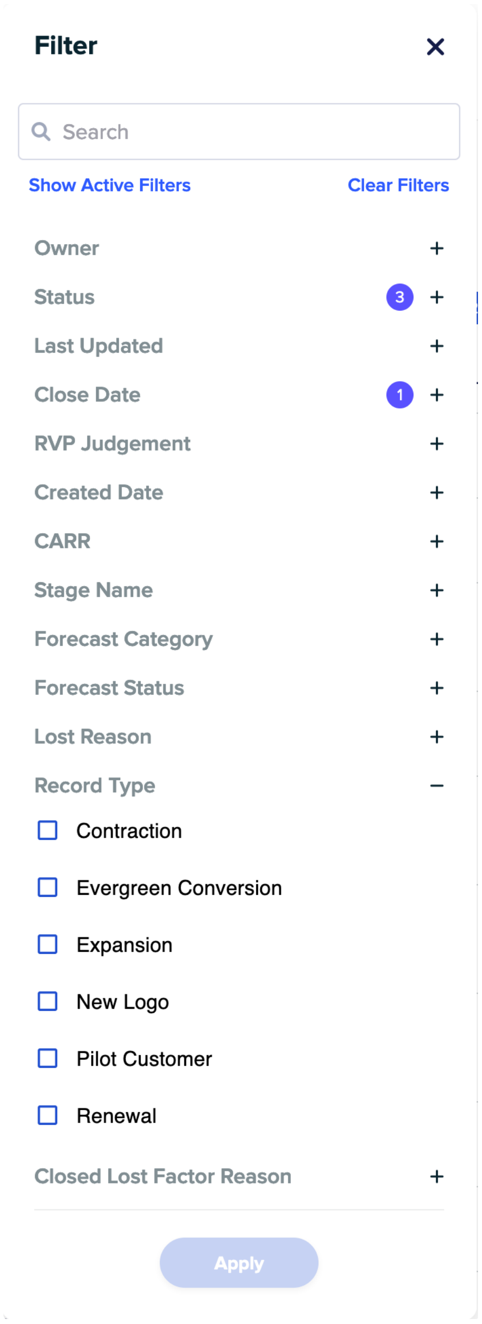
being intimidating while supporting a power-user to create the complex.

- Compact
  By design, the number/size of filters is not limited and can become very complex. The design needs to allow for one to many filters and filter groups to be applied while still remaining readable.

- Constrain user input/apply client-side logic to prevent impossible filter combinations
  As described previously, a user could combine filters in an illogical way that would allow for either ambiguity or guaranteed null results. In order to provide a good experience, the components must enforce certain rules by providing relevant feedback/errors, preforming actions on behalf of the user and hiding/showing options in a given situation. These forcing measures should be done in a manner that is consistent with the rest of the Commit app.

## Ideation

The current opportunity filters use an all-in approach, where all available filters are presented then the user can pick and choose what options to toggle, as depicted below.


Filter

Search

Show Active Filters
Clear Filters

Owner +

Status 3 +

Last Updated +

Close Date 1 +

RVP Judgement +

Created Date +

CARR +

Stage Name +

Forecast Category +

Forecast Status +

Lost Reason +

Record Type −

- [ ] Contraction
- [ ] Evergreen Conversion
- [ ] Expansion
- [ ] New Logo
- [ ] Pilot Customer
- [ ] Renewal

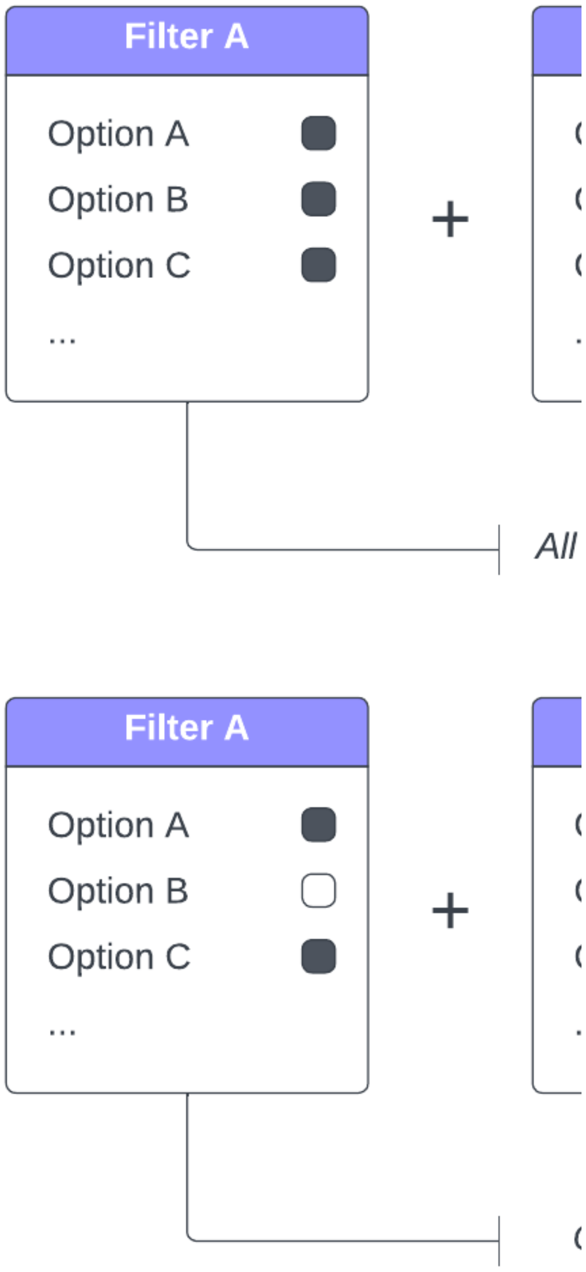
Closed Lost Factor Reason +

Apply

This is a common pattern, typically when the content has a finite set of criteria on which to filter.  For example many e-commerce sites utilize this type of filtering, where all options are listed and selecting one reduces the main result set. This is not ideal for a Commit customers as their data is driven by CRM which does not limit the number and type of data fields that can be defined. To query and render all the available fields with all the available  options would be taxing on the server/client and overwhelming to the user.

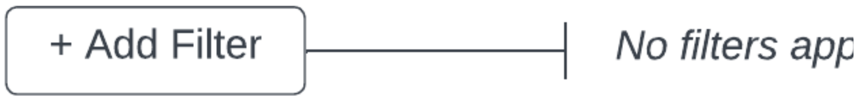
**Subtractive Filtering Pattern**

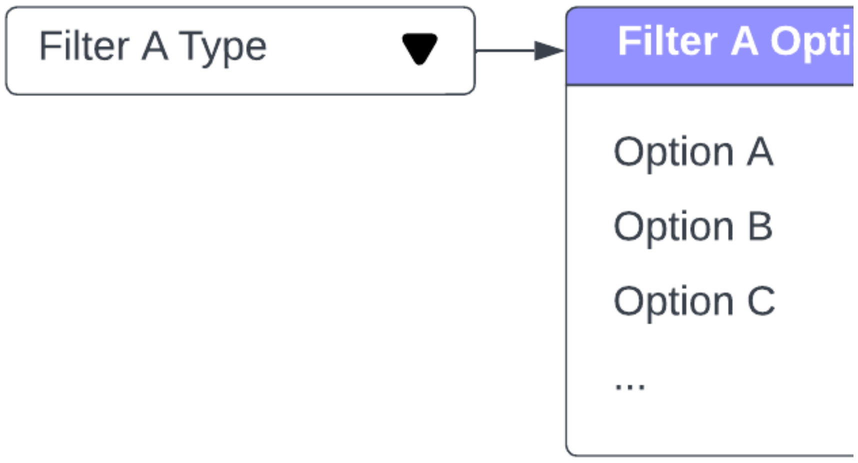
## Subtractive Filtering



An alternative approach is to start with no parameters and add in only what is relevant. This approach ensures a clean and simple initial step. It also reduces the initial load and render time by reducing the amount of data that is necessary to render the controls. This “additive” pattern is less common among consumer and e-commerce sites, however more common in digital applications that interface with large data-sets, such as geographic, weather, documents and financial.

**Additive Filtering Pattern**





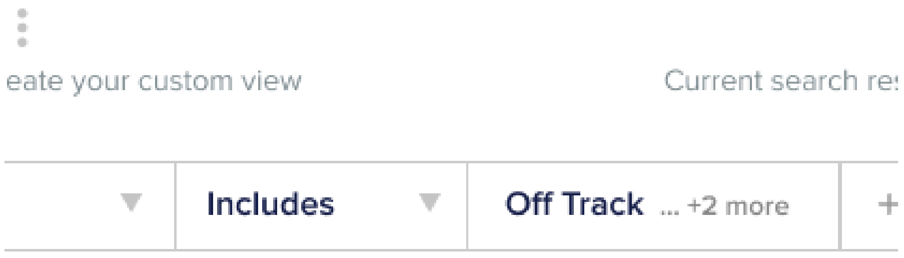
When looking at systems that apply this type of filtering, some common patterns emerge. First, filters are arranged in rows, made up of several fields. The first field is the field selector, where all the available properties or attributes of the data are listed. Next is a comparison operator, relevant to the selected field. Finally, there is a value input, which again varies by field type and comparison operator. Users can optionally add additional filter criteria, but how this is accomplished differs by implementation. Here again is the design challenge of how multiple filters are combined. While some offer only a single “track” of logic, using either AND or OR logic, others offer more complexity by grouping this logic. This allows for more granular and precise logic and is the type that is implemented in Commit.

The initial concept explores a graphical grouping system, where a filter condition is created then can be moved/dragged around to create groups, demonstrated in the following flow:

eate your custom view

Current search re:

:er

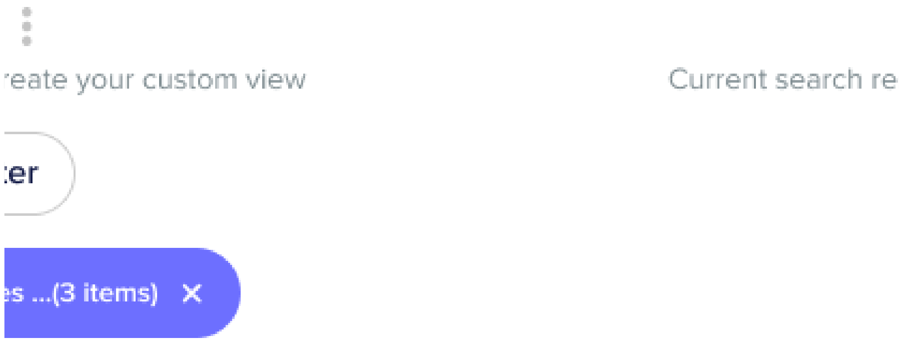

eate your custom view

Current search re:

Includes

Off Track ... +2 more

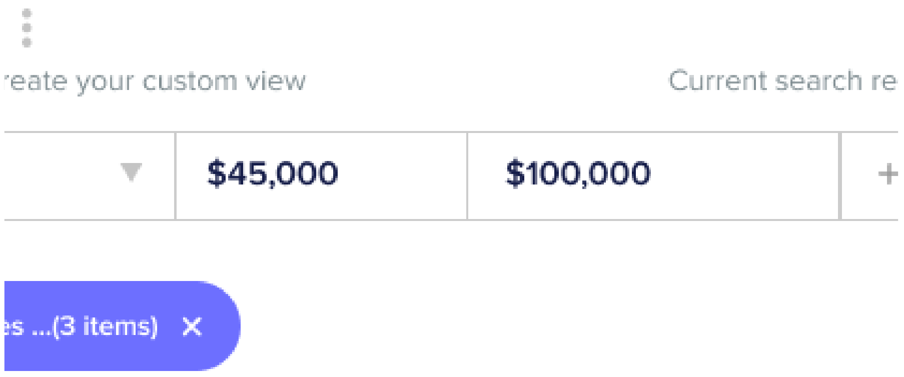
+


reate your custom view

Current search re

:er

es ...(3 items) ✕


reate your custom view

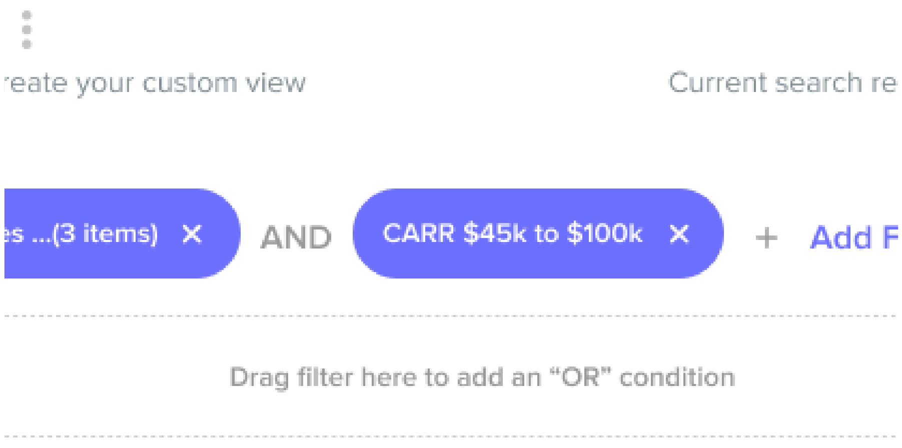
Current search re

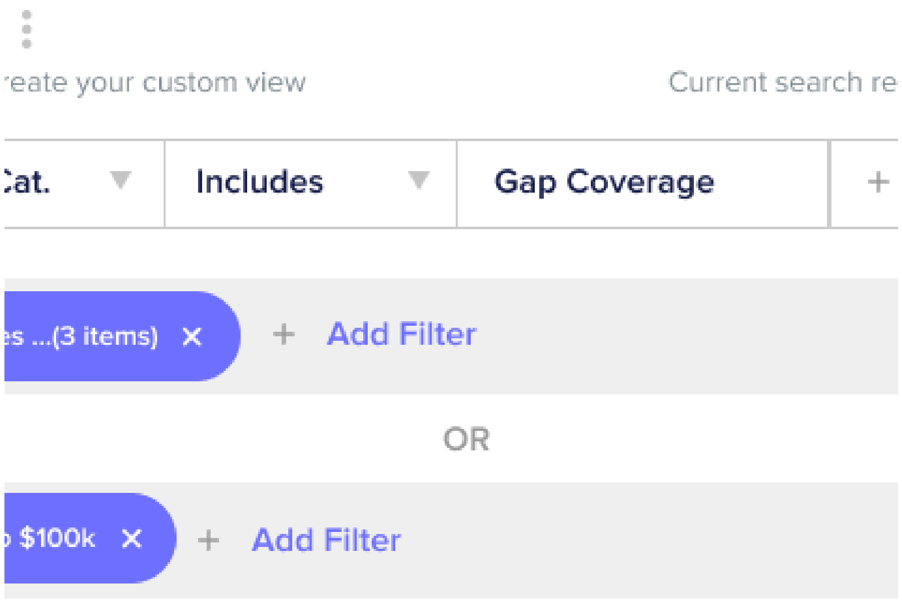
$45,000

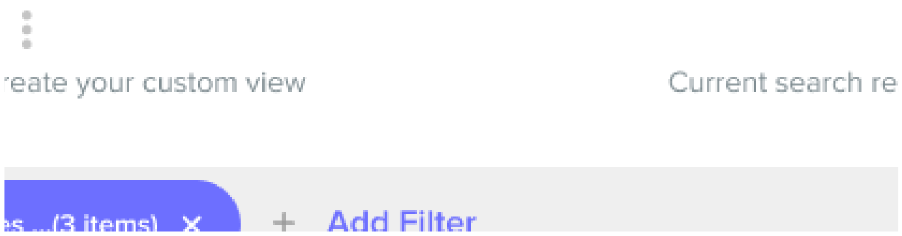
$100,000

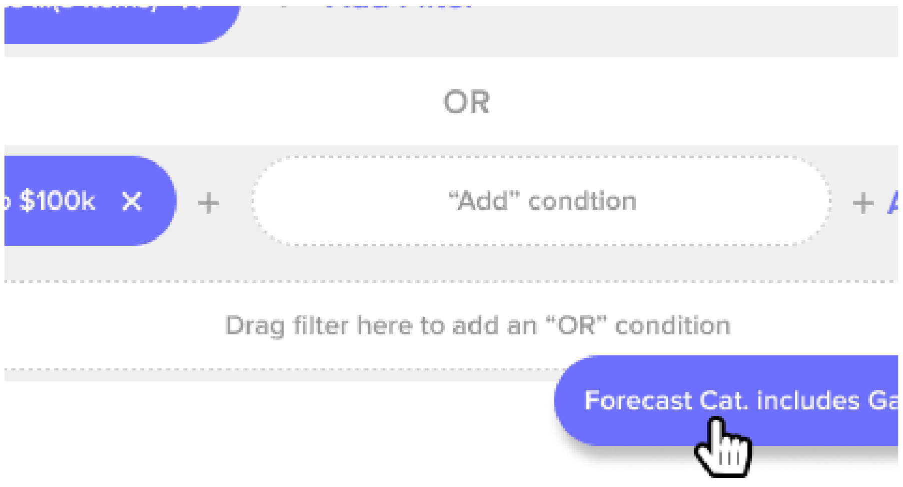
+

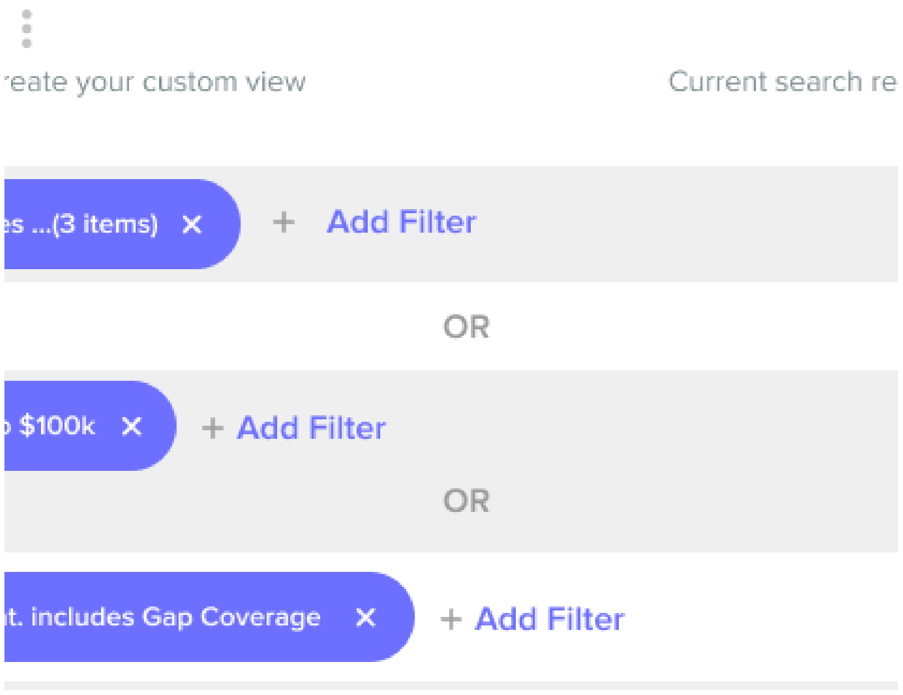
es ...(3 items) ✕


eate your custom view
Current search re
es ...(3 items)
AND
CARR $45k to $100k
Add F
Drag filter here to add an "OR" condition


eate your custom view
Current search re
Cat.
Includes
Gap Coverage
es ...(3 items)
Add Filter
OR
$100k
Add Filter


eate your custom view
Current search re
es ...(3 items)
Add Filter


OR
$100k ✕
+
"Add" condtion
+
Drag filter here to add an "OR" condition
Forecast Cat. includes Ga

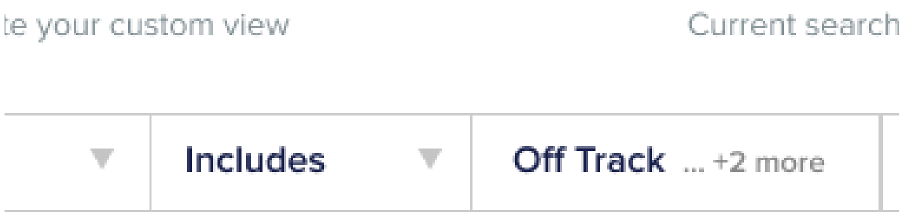


This design does a good job at minimizing the complexity, albeit at the expense of readability. The consolidated filter condition removes too much information, requiring the user to click into the item to understand the full definition of the filter. Additionally, the portability of the filter conditions does not offer much in the way of clearly defining boolean grouping logic.

The next design iteration solves for the filter definition being too portable and opaque by leaving the inputs exposed and functional.

te your custom view

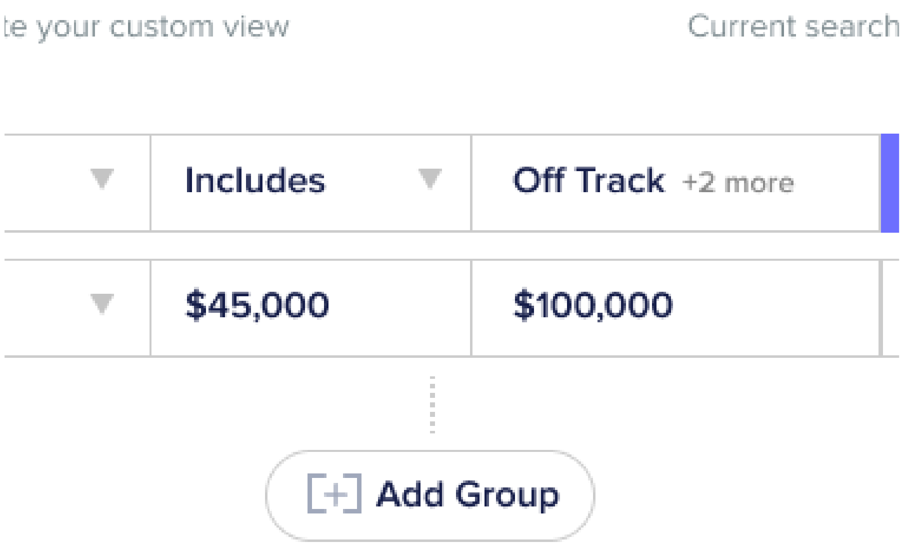
Current search



te your custom view

Current search

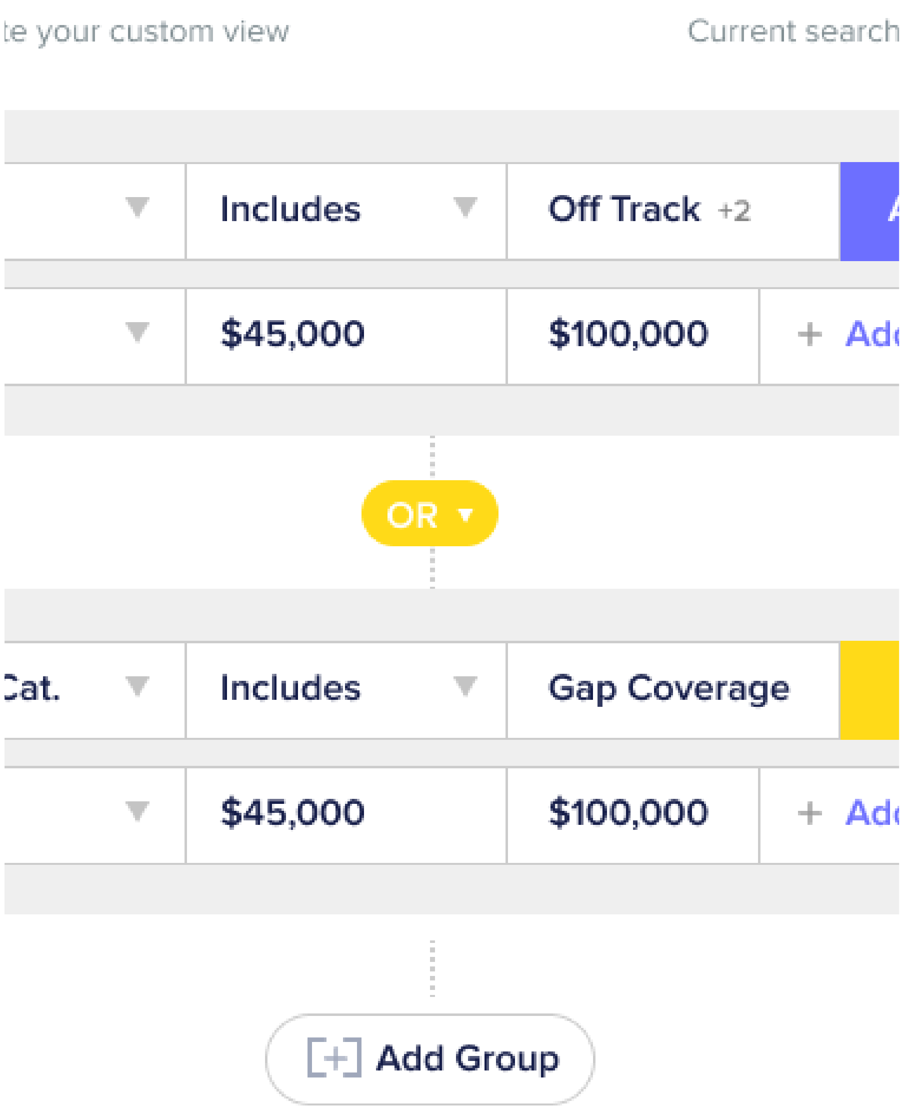
| | | |
|---|---|---|
| ▼ | Includes ▼ | Off Track ... +2 more |



te your custom view

Current search

| | | |
|---|---|---|
| ▼ | Includes ▼ | Off Track +2 more |
| ▼ | $45,000 | $100,000 |


te your custom view

Current search

Includes

Off Track +2

$45,000

$100,000

+ Add

OR

Cat.

Includes

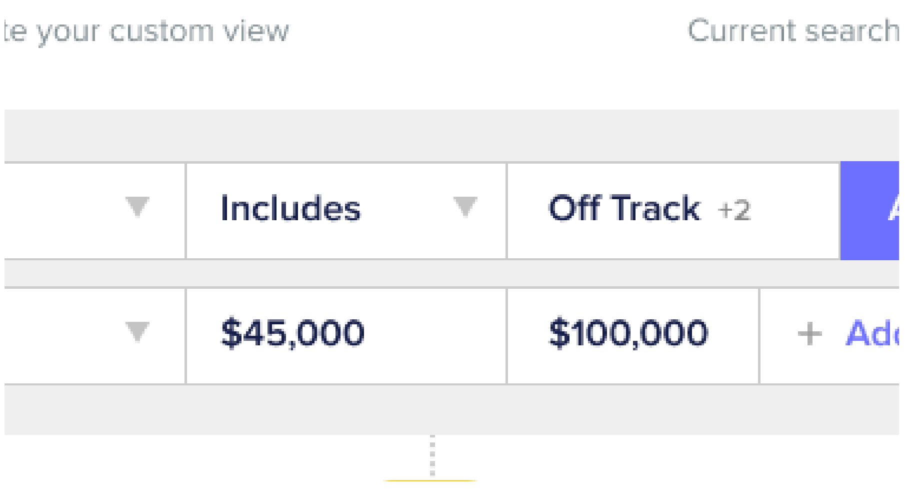
Gap Coverage

$45,000

$100,000

+ Add

Add Group


te your custom view

Current search

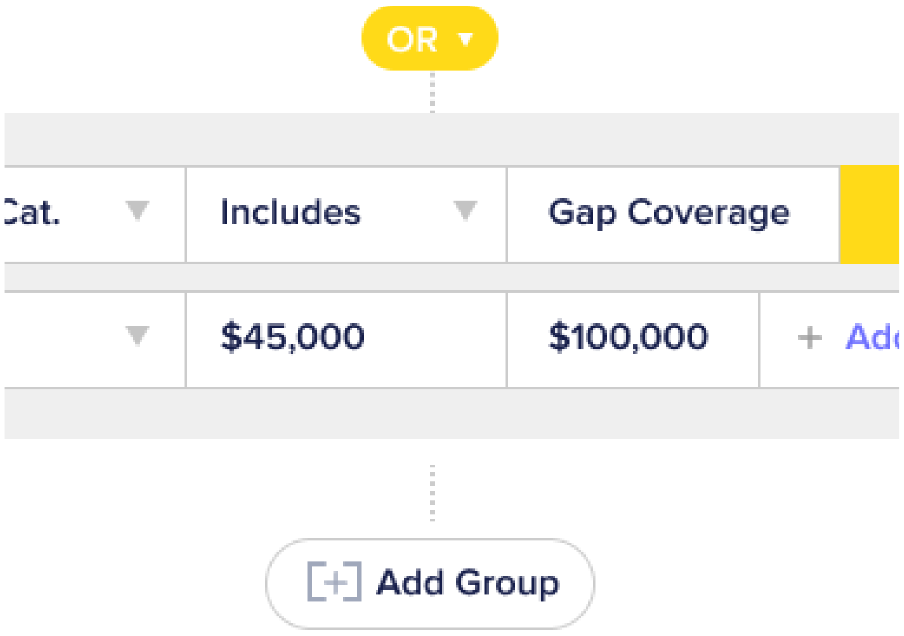
Includes

Off Track +2

$45,000

$100,000

+ Add



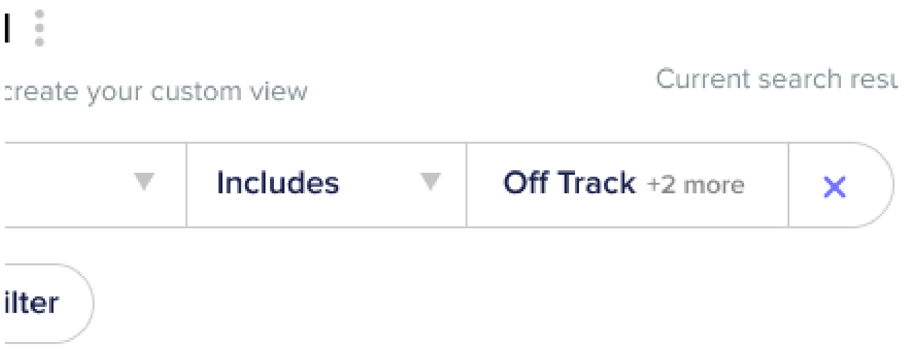
Additionally in this design, grouping logic is much more clear. Filters are created in a column form, and continues down until a new grouping is created. This is where the client-side enforcement of structure takes place. Items *within* the group use the same combining operator, and do so *exclusively.* Changing the AND or OR within a group changes them all to match. That change has no effect on the combining operator *among* groups or within filter conditions of other groups. It is important to note that in similar fashion to combining filter conditions in a group using the same operator, *groups* must now also use the same operator to combine with one another.

This design is clear and functional allowing flexibility and clarity, however, limits the amount of complexity a user can define. Filters are confined to group, located a static depth.

While this would likely serve the needs of most users, the server-side model allows for infinite nesting of groups of filter definitions, which is realized in the final design iteration.

reate your custom view | Current search resu

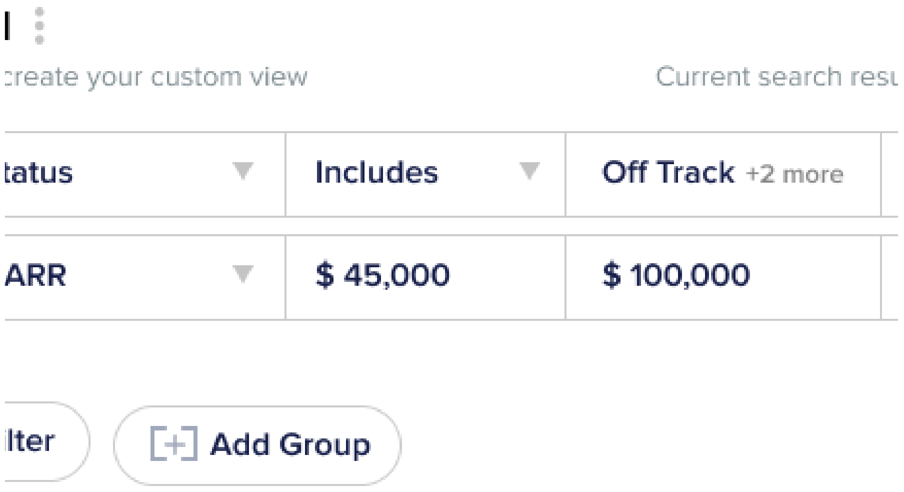
lter



reate your custom view | Current search resu

| | | | |
|---|---|---|---|
| | Includes | Off Track +2 more | × |

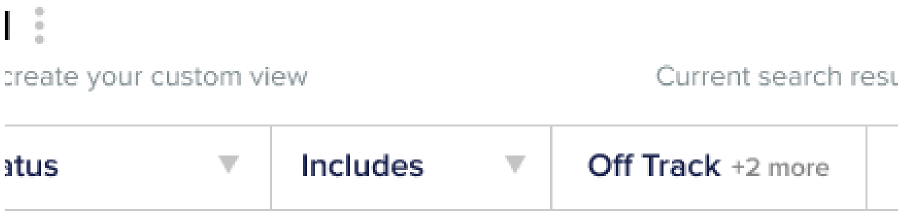
ilter



reate your custom view | Current search resu

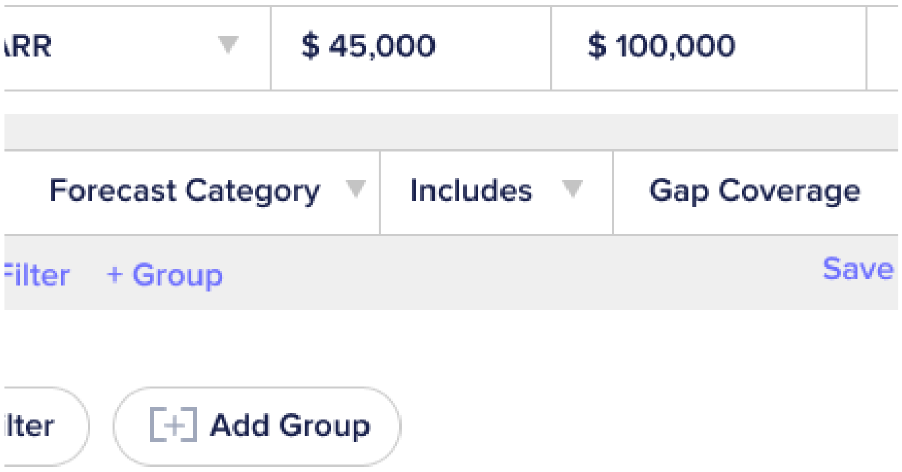
| | | |
|---|---|---|
| tatus | Includes | Off Track +2 more |
| ARR | $ 45,000 | $ 100,000 |

lter | Add Group



reate your custom view | Current search resu

| | | |
|---|---|---|
| atus | Includes | Off Track +2 more |

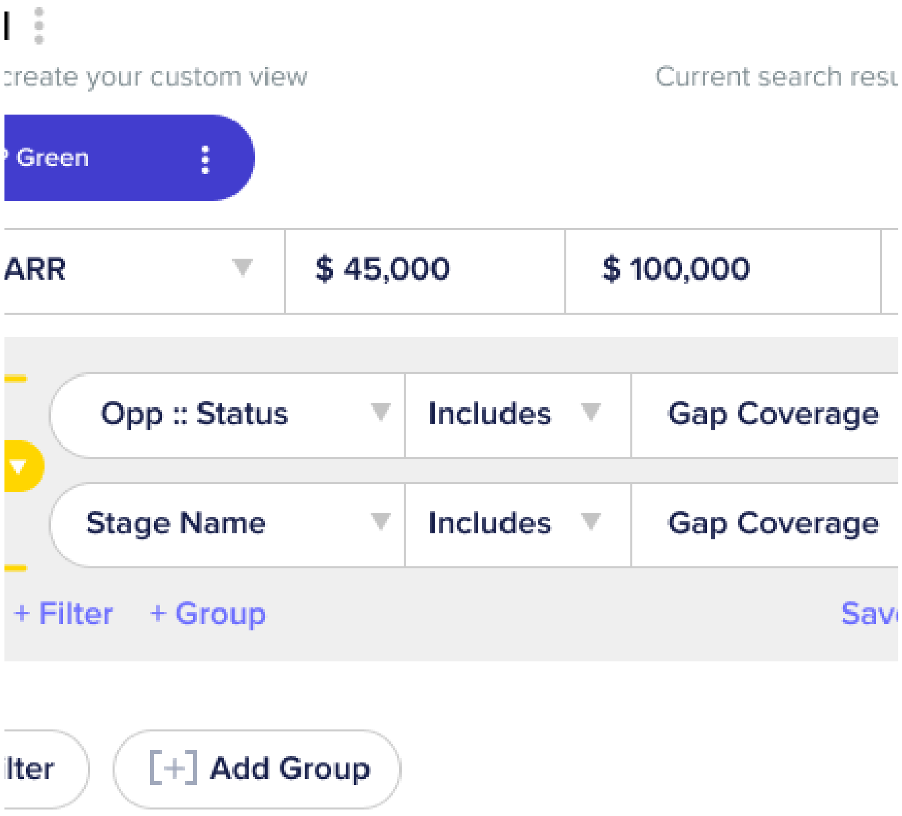


ARR ▼ | $ 45,000 | $ 100,000

Forecast Category ▼ | Includes ▼ | Gap Coverage

Filter + Group Save

lter [+] Add Group



I

create your custom view

Current search resu

Green

ARR ▼ | $ 45,000 | $ 100,000

Opp :: Status ▼ | Includes ▼ | Gap Coverage

Stage Name ▼ | Includes ▼ | Gap Coverage

+ Filter + Group Sav

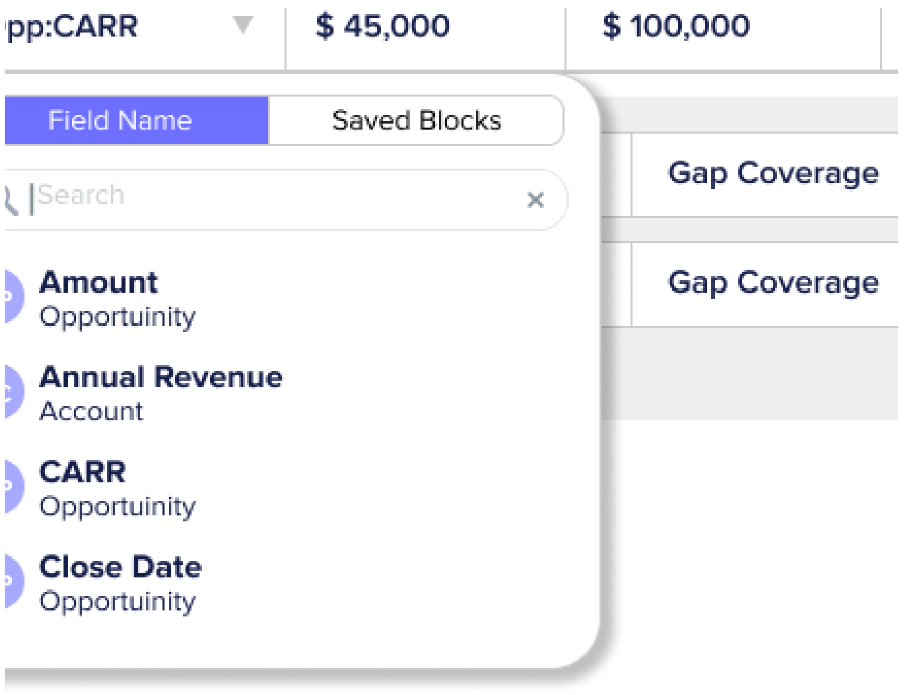
lter [+] Add Group

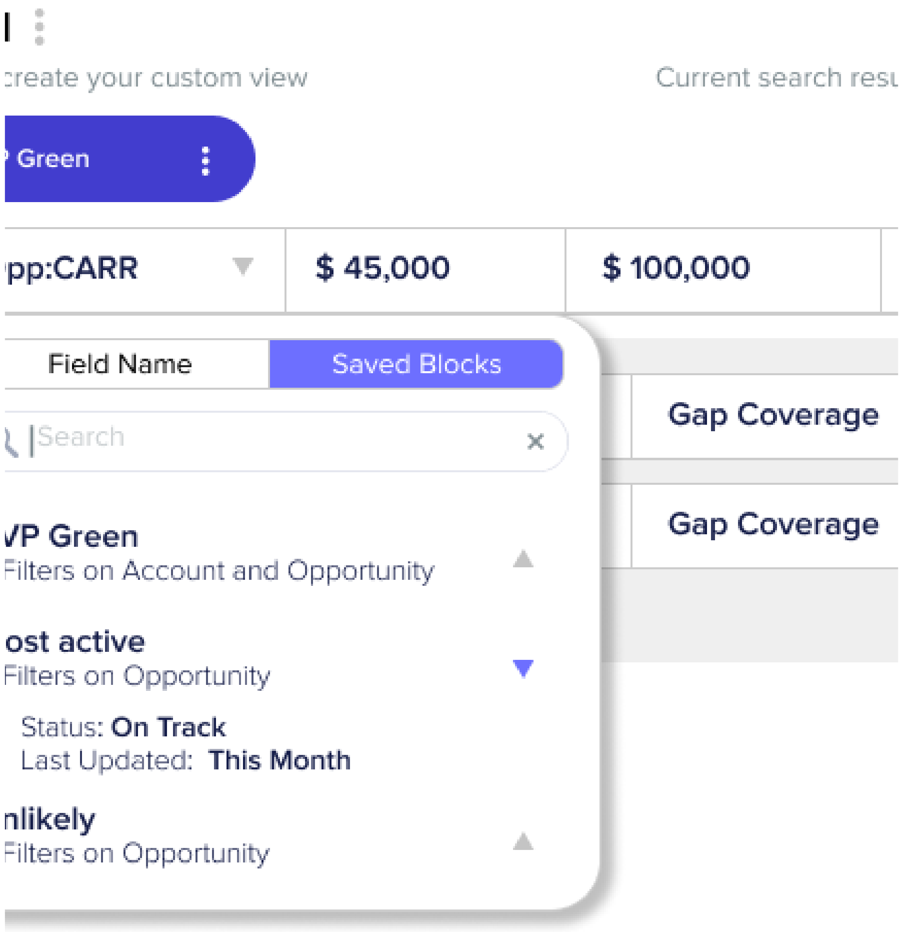
I

create your custom view

Current search resu

Green





This approach extends the previous design by allowing for the creation of groups within groups. Indeed, the data structure can quickly become complex, but only as much as the user desires. The graphical group structure does well to demonstrate boolean relationships while serving as combiner user input.

It is worth noting that this design also explores a possible feature to save filter configurations into “blocks” that can be reused in the future or by other teams. This serves to consolidate visual noise while offering a convenience to power users, however lives outside of the scope and functional requirements of this feature. This design will ultimately become the design that will be implemented, with refinements from the UX team.

## Implementation

This design and feature presents some implementation challenges, due to the recursive data structure and therefore requires “generative” UI. This complexity is in addition to the aforementioned client-side enforcement of data integrity as well as the potentially large amount of data required to render the filter options, i.e. listing all the member names in an organization. These complications are described in detail below.

### Data model and providers

Initially, filters will be used as a way to define the data displayed by a realtime metric however can be used in conjunction of any other higher-level model. The data structure is defined by the backend and therefore out of scope of this document, however the resulting model is broken down for the client to consume and described here using protos models with non-relevant properties omitted.

The highest order defines `UserQueryParameters`, which generally describes a container for the metric. The relevant type here is the `QueryParameters` object, yielding an array of `QueryParameter`. Each `UserQueryParameter` is managed by an instance of a `FilterEditorProvider` in the React app.

```
message UserQueryParameters {
  string id = 1;
  string tenant_id = 2;
  string group_id = 3;
  string user_id = 4;
  string query_key = 5;
  string name = 6;
  string description = 7;
  QueryParameters parameters = 8;
  ...
}

message QueryParameters {
  repeated QueryParameter parameters = 1;
}

message QueryParameter {
  ParameterType type = 4;
  ParameterOptions options = 5;
  FilterParameter filter = 6;
  ValueParameter value = 10;
  ResolvedQueryParameter resolved_query = 11;
  ...
}
```

The QueryParameter object defines what has been referred to as a filter definition in this document. There are several types defined in the enum, however, the filter, value and resolvedQuery types are what the client uses and are relevant to this document.

```
enum ParameterType {
  PARAMETER_TYPE_FILTER = 1;
  PARAMETER_TYPE_VALUE = 5;
  PARAMETER_TYPE_RESOLVED_QUERY = 7;
  ...
}

message ParameterOptions {
  repeated SourceOptions sources = 1;
}

message SourceOptions {
  string source = 1;
  repeated FieldOptions fields = 2;
}

message FieldOptions {
  string field = 1;
  string label = 2;
  FieldType field_type = 3;
  string format = 4;
  repeated LabelValue values = 5;
  loupe.common.mapping.FieldReference field_reference = 6;
}
```

The `QueryParameter` may hold SourceOptions. Sources are the container of all available `FileldOptions` that a user may select to create a filter definition. For example, field options may include options such as Owner, Close Date, or Stage Name. It is important to note that at this point, the `FieldOptionData` has not yet been loaded in the initial api request. This is due again to the potential huge amount of data being delivered in a single request. Before a filter definition can be created, all available `SourceOptions` must be loaded, in a separate api request. This logic is handled by the `FilterSourceOptionsProvider`. This ensures that all source data is available to all components within it, including the `FilterEditorProvider`.

```
<FilterSourceOptionsProvider>
  <RealtimeFilterEditorProvider>
    ...
  </RealtimeFilterEditorProvider>
</FilterSourceOptionsProvider>
```

These providers contain all the logic and data required to render, create and edit filter definitions.
With the parameter definitions and data loaded, `FilterEditor` houses the initial set of components for managing filters. The filter editor component simmply renders a list of `QueryParameters`, as described above, and contained in the `ParameterRow` component.

The `ParameterRow` component simply evaluates the parameter's type `[FILTER | RESOLVED_QUERY | VALUE]` and renders the relevant subtype component. While these types are unique, they are generally made up of the same elements, with resolved query and value types being less complex than that of filter. These are broadly row[s] of key/value pairs and their implementation is inherently covered with the proceeding description of the filter type parameter.

The fiiter editor is generally structured thus far as:

```
<FilterSourceOptionsProvider>
  <RealtimeFilterEditorProvider>
    <FilterEditor>
      {
        <ParameterRow>
          <FilterParameter | ValueParameter | ResolvedQueryParameter>
```

```
            ...
          <FilterParameter | ValueParameter | ResolvedQueryParameter>
        </ParameterRow>
        ...parameterRows
      }
  </RealtimeFilterEditorProvider>
</FilterSourceOptionsProvider>
```

Returning to the data model, the `QueryParameter` describes a `FilterParameter` and its types:

```
message FilterParameter { ;nl 7
  FilterWrapper filters = 1;
}

message FilterWrapper {
  Filter filter = 1;
  repeated FilterWrapper nested = 2;
  FilterCombiner combiner = 3;
}

enum FilterCombiner {
  AND = 0;
  OR = 1;
}

message Filter {
  string source = 1;
  string field = 2;
  FieldType field_type = 3;
  ComparisonOp comparison_op = 4;
  repeated string values = 5;
  QueryDateRangeType query_date_range_type = 6;
}
```

As defined, a `FilterParameter` defines a `FilterWrapper`, which in turn defines a `Filter`, `FilterCombiner`, and/or another object of its own type. The `FilterWrapper` has only 3 properties but houses all the filter definitions as well as describes the high-level logic for how filter definitions can be combined as described below.

Starting with an empty wrapper, a filter type may be added but, is done so by first creating a new wrapper, then adding the filter. The reason for this is by determined by the data structure. The `FilterWrapper` property is an array. Items in the array are compared by the combiner value at that level. Setting a default combiner value ensures this is enforced.

After adding a new filter, the data structure is that of:

```
FilterWrapper
{
  nested: [
    {
      filter: filterA
    }
  ],
  combiner: 'AND',
  filter: null
}
```

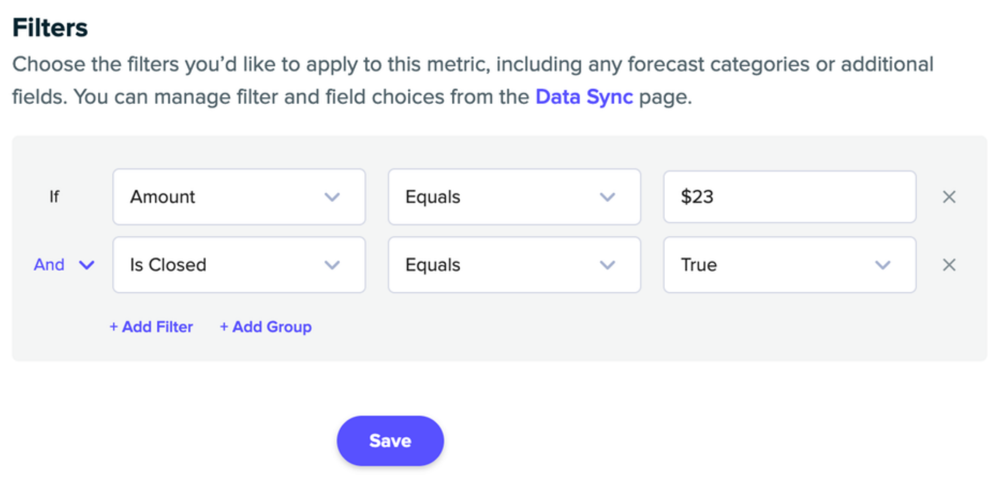
To add an additional filter is simply adding another `FilterWrapper` with a defined `Filter` to the existing nested property. In this manner, the user is able to create n-number of filters either reducing the result set with the AND combiner or expanding the result with the OR combiner.

```
FilterWrapper
{
  nested: [
    {
      filter: filterA
    },
    {
      filter: filterB
    }
  ]
  combiner: 'AND',
  filter: null
}
```

The component structure that has formed as a result of adding thees two filters look like this:

```
<FilterSourceOptionsProvider>
  <RealtimeFilterEditorProvider>
    <FilterEditor>
      {
        <ParameterRow>
          <FilterParameter>
            <FilterWrapper>
              <ConditionRow (filter) />
            </FilterWrapper>
            <FilterWrapper>
              <Combiner />
              <ConditionRow (filter) />
            </FilterWrapper>
          <FilterParameter>
        </ParameterRow>
        ...parameterRows
      }
  </RealtimeFilterEditorProvider>
</FilterSourceOptionsProvider>
```

Of note in this structure is that there is only one Combiner component for the two conditions. Adding additional filters at to the wrapper would have no effect to this. The UI for these components should look like the following:


Filters
Choose the filters you'd like to apply to this metric, including any forecast categories or additional fields. You can manage filter and field choices from the Data Sync page.
If
Amount
Equals
$23
And
Is Closed
Equals
True
+ Add Filter
+ Add Group
Save

Thus far the filters are applied in the same group, on the same level. In order to be able to nest filters to apply complex logic using AND and OR, the user needs to be able to create groups. To do so, no modifications needs to be made to the data structure, only in the logic applied when creating a group and when rendering the ui. Continuing with the working example, when selecting the "Add Group" option, the resulting data model would look like this:

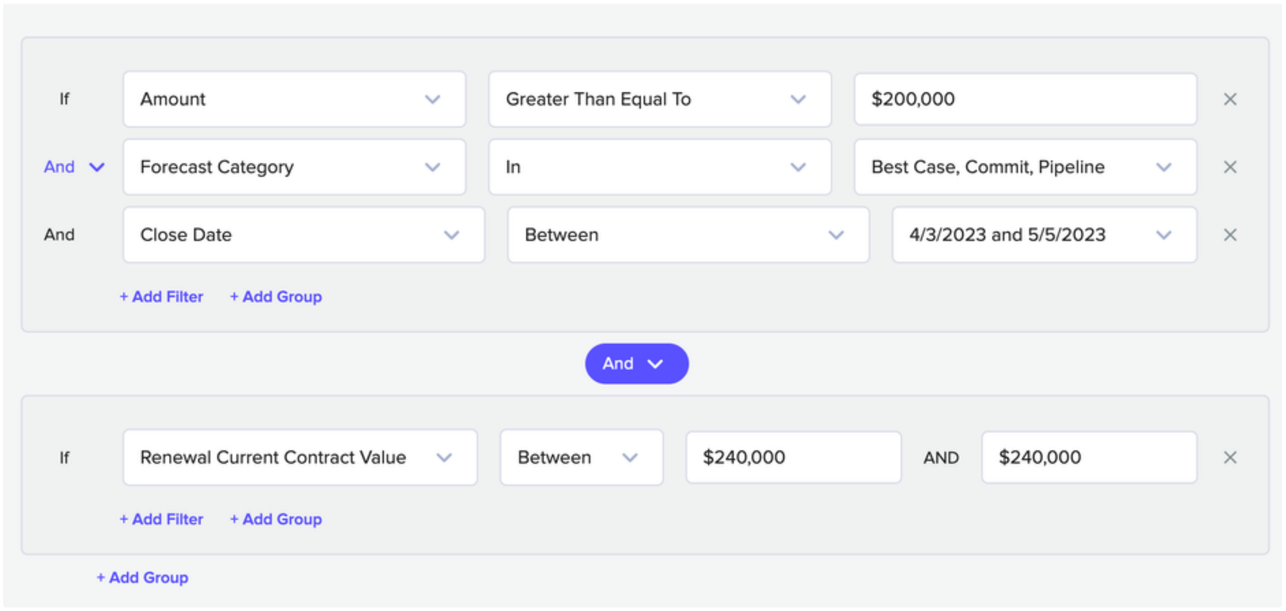
```
FilterWrapper
{
  nested: [
    {
      nested: [
        {
          filter: filterA
        },
        {
          filter: filterB
        }
      ]
    },
    {
      nested: [
        filter: filterC
      ]
      combiner: 'AND',
      filter: null
    }
  ]
  combiner: 'OR',
  filter: null
}
```

This is the final bit of logic that allows for the formation of any query combination. When the new group is added, the existing contents of that `FilterWrapper` are removed and re-inserted as a child member of a new `FilterWrapper` object. At the same time, another `FilterWrapper` is created and inserted as the second child, with a new empty `Filter` object. This effectively creates depth as well as allows for combinations of AND and OR logic to be created in and among groups of filter definitions. The final component structure demonstrates the complexity of this this logic:

```
<FilterSourceOptionsProvider>
  <RealtimeFilterEditorProvider>
    <FilterEditor>
```

```
      {
        <ParameterRow>
          <FilterParameter>
            <FilterWrapper>
              <FilterWrapper>
                <ConditionRow (filter) />
              </FilterWrapper>
              <FilterWrapper>
                <Combiner />
                <ConditionRow (filter) />
              </FilterWrapper>
              <FilterWrapper>
                <ConditionRow (filter) />
              </FilterWrapper>
            </FilterWrapper>
            <FilterWrapper>
              <Combiner />
              <ConditionRow (filter) />
            </FilterWrapper>
          <FilterParameter>
        </ParameterRow>
        ...parameterRows
      }
  </RealtimeFilterEditorProvider>
</FilterSourceOptionsProvider>
```

While the UI rendered from this structure appears as such:


If | Amount | Greater Than Equal To | $200,000
And | Forecast Category | In | Best Case, Commit, Pipeline
And | Close Date | Between | 4/3/2023 and 5/5/2023

And

If | Renewal Current Contract Value | Between | $240,000 AND $240,000

**Filter options, comparators and value types**

The other main design consideration, aside from grouped and nested logic is that of dynamically rendered filter definition options. The options discussed in the requirements section of this document generally cover the field types, comparison operators and field inputs, however this section describes the logic and rendering handled by the client.

The available filterable fields and datatype are determined by the customer using the data-mapping feature in Commit admin. The resulting object delivered to the UI to be rendered looks similar to these examples:

```
{
  "field": "nextStep",
  "label": "Next Step",
  "fieldType": 1,
  "format": "richtextarea",
  "valuesList": []
}

{
  "field": "forecastCategoryName",
  "label": "Forecast Category",
  "fieldType": 1,
  "format": "select",
  "valuesList": [
    {
      "label": "Best Case",
      "value": "\"Best Case\""
    },
    {
      "label": "Closed",
      "value": "\"Closed\""
    },
    {
      "label": "Commit",
      "value": "\"Commit\""
    },
    {
      "label": "Omitted",
      "value": "\"Omitted\""
    },
    {
      "label": "Pipeline",
      "value": "\"Pipeline\""
    }
  ]
}
```

The objects to be rendered contain properties commonly used when creating forms, like a name, label and options. The objects also assign a format field, to ensure the correct input validation is applied. The filter options also include a FieldType property as described previously and referenced as enums as follows:

```
enum FieldType {
  FIELD_TYPE_UNSPECIFIED = 0;
  FIELD_TYPE_STRING = 1;
  FIELD_TYPE_INT = 2;
  FIELD_TYPE_FLOAT = 3;
  FIELD_TYPE_BOOL = 4;
  FIELD_TYPE_DATE = 5;
  FIELD_TYPE_TIMESTAMP = 6;
  FIELD_TYPE_RAW = 7;
  FIELD_TYPE_ARRAY = 8;
  FIELD_TYPE_OBJECT = 9;
}
```

This property allows the UI to render a relevant list of comparison operators. Without this property, the user could potentially select a comparator that is illogical to the type. For example, using a BETWEEN comparison on a boolean field type. To control for this a hook is created, statically defining available comparison operators for a given type. This is where the UI must have prior knowledge of the types that may be received. Aside from the hook, the main component for dictating what gets rendered in the UI is handled by the `ConditionRow` component.

The `ConditionRow` ensures input sequence by ensuring that a field type is selected first, followed by a comparison operator, and finally a valid input. Again the latter two inputs are dictated by the initial field selected. If the user selects a different field, the selection process must start again to present only relevant inputs.

The `ConditionRow` component is generally comprised of the following components:

```
<ConditionRow>
  <OperatorSelector />
  <FieldSelect fieldType={[...fieldTypes]} />
  <FieldSelect fieldOperators={[...relevantOperators]} />
  <FieldInput
    type
    format
    options />
</ConditionRow>
```

The `OperatorSelector`, is the comparator among `FieldWrappers` as discussed previously, and is only rendered once in a group. The following two `FieldSelects` display single-select options for field types and comparison operators respectively. The final `FieldInput` component does not render a field directly but is an abstract component representing the rendered user input(s).

The fields ultimately rendered are the result of a helper object that takes in a `FieldType` and operator and returns components to render. The components that can be rendered are:

> FieldSelect
> FieldMultiSelect
> FieldInput
> FieldMultiInput
> FieldInputRange
> FieldDateRange
> FieldDate
> FieldEmpty

The types are shallow wrappers around the HTML elements of similar name, however the "range" components render two fields representing the min and max values of the selected field. The wrappers are each responsible for ensuring that the entered input conforms to the format type defined by the `FieldType`. This is most relevant with the `FieldInput` type, which could be formatted to a string, integer, float or pick list, with special formatting for types of currency, percent and others.

## Future considerations

The final component satisfies the current requirements but does offer several opportunities for improvement in the areas of usability, scalability and visibility.

### Usability

The dynamic/generative nature of this component functions generally well for the initial 4-5 levels of depth, however can become less readable going deeper depending on the filter name and type. Additionally, while a majority of our users interact with Outreach on traditional screen sizes, mobile usage should be a consideration moving forward.

Currency type is already taken into account, based on Tenant settings, however localization has not. The amount of static text is minimal, broader implications of usage and context may affect how our growing segment of international customers interact with this component.

**Scalability**

The types currently supported reflect the need and definitions our current customers, however new data types may be needed with new logos or potentially with the adoption of a new CRM provider. To facilitate this, both client and server-side contracts need to be updated to be agreement. This is a manual process at the moment but could be dynamic by refactoring the client to rely on type-comparator-input/format mappings from a service endpoint, rather than from static, client-side configurations.

In terms of data size, more can be done to reduce the amount of data needed to render the components by implementing more lazy loading instances, as well as using logical grouping of options based on data type and comparators. This may become a more pressing issue as Outreach continues to attract larger/enterprise customers that come with deeper orgs and more mapped CRM fields.

**Visibility**

Quantifying data in various contexts to provide insight into the expected result-set may assist the user to make more informed design decisions when constructing filters. For example, it may be of benefit to display the total count of options for a multi-select data type, or show the minimum/maximum value of all deals in pipeline. Additionally, “preflighting” the query to display the actual result as the user interacts with the component would ensure the user gets the expected result. This could even be displayed in context of a group or single filter to highlight the additive/subtractive nature of combining logic with AND/OR operators.