Design: Array updates

Author: Tess Avitabile
Last Updated: 2017-06-22

Summary & Motivation

Users who have documents with complicated array structures have a difficult time updating or manipulating them. At best, the current array update capabilities are clumsy, leading to extremely awkward operations or ambiguous semantics. At worst, certain updates are outright impossible without performing the updates client-side and issuing a save-style update, including the ability to update multiple elements of an array (the "poly-array update"). There is a very strong interest in improved array update capabilities from the community, with many complaints that these features are essential for anyone with a schema that utilizes arrays.

Behavioral Description

Add new path syntax to all update modifiers:

```
{$<updateModifier>: {<path>.$[<id>].<path>: ...}}
```

Add new option to `update` command. The elements of `arrayFilters` are filters on the array elements the update should apply to:

```
{arrayFilters: [<filter>, ...]}
```

`arrayFilters` is supported for the write command only.

Examples

- Update all documents in array:
  Input: `{a: [{b: 0}, {b: 1}]}`
  Output: `{a: [{b: 2}, {b: 2}]}`

- Update all matching documents in array:
  Input: `{a: [{b: 0}, {b: 1}]}`
  Output: `{a: [{b: 2}, {b: 1}]}`
  ```
  db.coll.update({}, {$set: {“a.$[i].b”: 2}},
                 {arrayFilters: [{“i.b”: 0}]})
  ```

- Update all matching scalars in array:
  Input: `{a: [0, 1]}`
  Output: `{a: [2, 1]}`
  ```
  db.coll.update({}, {$set: {“a.$[i]”: 2}}, {arrayFilters: [{i: 0}]})
  ```

- Update all matching scalars in array of arrays:
  Input: `{a: [[0, 1], [0, 1]]}`
  Output: `{a: [[2, 1], [2, 1]]}`
  ```
  db.coll.update({}, {$set: {"a.$[].$[j]": 2}},
                 {arrayFilters: [{j: 0}]})
  ```

- Update all matching documents in nested array:
  Input: `{a: [{b: 0, c: [{d: 0}, {d: 1}]}, {b: 1, c: [{d: 0}, {d: 1}]}]}`
  Output: `{a: [{b: 0, c: [{d: 2}, {d: 1}]}, {b: 1, c: [{d: 0}, {d: 1}]}]}`
  ```
  db.coll.update({}, {$set: {"a.$[i].c.$[j].d": 2}},
                 {arrayFilters: [{"i.b": 0}, {"j.d": 0}]})
  ```

- Update all scalars in array matching a logical predicate:
  Input: `{a: [0, 1, 3]}`
  Output: `{a: [2, 1, 2]}`
  ```
  db.coll.update({}, {$set: {"a.$[i]": 2}},
                 {arrayFilters: [{$or: [{i: 0}, {i: 3}]}]})
  ```

Error cases

- Provide an `<id>` with no array filter:
  ```
  db.coll.update({}, {$set: {"a.$[i]": 0}})
  ```

- Use an `<id>` at the same position as a `$`, integer, or field name:
  ```
  db.coll.update({a: 0}, {$set: {"a.$[i]": 0, "a.$": 0}},
                 {arrayFilters: [{i: 0}]})

  db.coll.update({}, {$set: {"a.$[i]": 0, "a.0": 0}},
                 {arrayFilters: [{i: 0}]})

  db.coll.update({}, {$set: {"a.$[i]": 0, "a.b": 0}},
                 {arrayFilters: [{i: 0}]})
  ```

- Provide multiples filters for the same `<id>`:
  ```
  db.coll.update({}, {$set: {"a.$[i]": 0}},
                 {arrayFilters: [{i: 0}, {i: 1}]})
  ```

- Include a top-level field name in `arrayFilters` that is not an `<id>` in the update parameter:
  ```
  db.coll.update({}, {$set: {a: 0}}, {arrayFilters: [{j: 0}]})
  ```

- Include multiple top-level field names in a single `arrayFilters` filter:
  ```
  db.coll.update({}, {$set: {"a.$[i].b.$[j]": 0}},
  ```

```
                {arrayFilters: [{$or: [{“i.c”: 0}, {j: 1}]}]})
```

- Include an implicit array traversal in a path in an update modifier (this is already an error):
```
db.coll.insert({a: [{b: 0]})
db.coll.update({}, {$set: {“a.b”: 1}})
```

- <id> contains special characters or does not begin with a lowercase letter:
```
db.coll.update({}, {$set: {“a.$[$i]”: 1}}, {arrayFilters: [{“$i”: 0}]})
db.coll.update({}, {$set: {“a.$[I]”: 1}}, {arrayFilters: [{I: 0}]})
db.coll.update({}, {$set: {“a.$[i.j]”: 1}},
                {arrayFilters: [{“i.j”: 0}]})
```

Notes

- There is no mechanism to update only the first matching array element.
- There is no mechanism to update one array based on the index of a matching element in another array.
- The filters in `arrayFilters` will not be used for query planning.

Extensions

We are not planning to include options such as `limit` or `filterRelativeTo` in this project, but these options could be added using an additional `update` parameter:

```
arrayFilterOptions: [{id: <string>, limit: <int>,
      filterRelativeTo: "element" | {root: <path>} | {sibling: <path>}}, ...]
```

Design Alternatives

- **Add an update stage to aggregation.**
  We we would like to eventually move updates to the aggregation system, but this project was rejected for 3.6 due to time needed to achieve performance comparable to the update system.
- **Introduce an array update modifier.**
  These proposals were rejected due to treatment of scalars.
  - **$arrayUpdate**
    ```
    {$arrayUpdate: {
      <path>: {
        $updates: {
          <mod>: <document>,
          ...
        },
        $scalarName: <string>,  // optional
        $filter: <document>  // optional
      }
    }
    ```
  - **$recurse**

```
{$recurse: [{path:
  [{subpath: <string>,
    scalarName: <string>,  // optional
    filter: <document>}, ...],
 update: <document>}, ...]}
```

- **Allow nested positional $ operator, fix ambiguous positional $ updates, create syntax for positional $ poly updates.**
  This was rejected because we would like to move away from having query predicates fill out `MatchDetails` and we are interested in eventually deprecating the positional $ operator in the MongoDB update language. This is because a predicate is purely a function that takes a single document as input and returns a boolean result; the idea that a predicate can report "how it matched the document" is nonsensical and shouldn't be encoded in the MongoDB update language as part of the path traversal for an update modifier.
- **Allow custom Javascript function for updating a document.**
  This was rejected because of the large engineering effort that would be required to complete an implementation of our own minimal JS interpreter.
- **Add writable views/virtual collections.**
  Although desirable in its own right, this was rejected because it only indirectly addresses the array update use cases which we're trying to fix.

Detailed Design

Parse arrayFilters

Add a method to `ParsedUpdate` that takes in the `arrayFilters` as a `BSONArray` and parses them to `ArrayFilters`, which contain a field name and a `MatchExpression`. It should check that:

- Each array filter has a single top-level field name that starts with a lowercase letter and contains no special characters.
- No two array filters have the same top-level field name.

The parsed `arrayFilters` should be passed to `UpdateDriver::parse()`.

Parse update document

`UpdateDriver::parse()` should parse the update document to a tree of `UpdateNodes`:

```
class UpdateObjectNode : UpdateNode {
    // Map from field name (including "0") to UpdateNode.
    unordered_map<std::string, std::unique_ptr<UpdateNode>> _children;
    std::unique_ptr<UpdateNode> _positionalChild;
}

class UpdateArrayNode : UpdateNode {
    // Map from array filter to UpdateNode.
    std::map<ArrayFilter*, std::unique_ptr<UpdateNode>> _children;
}
```

```
class UpdateLeafNode : UpdateNode {
    // No children.
}
```

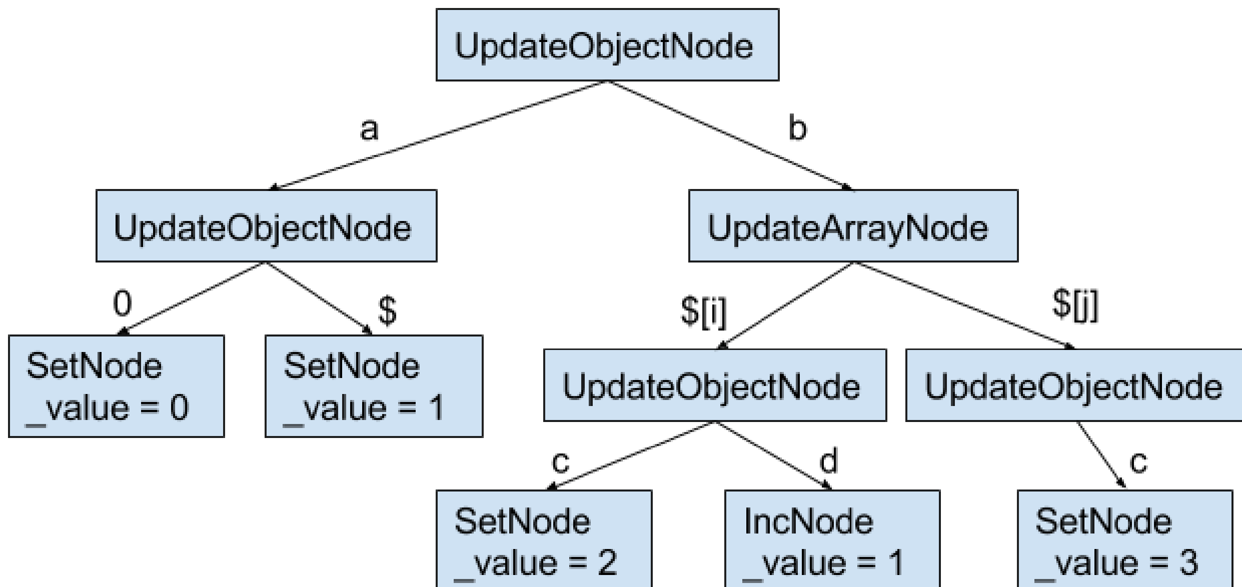
We fail to parse if we ever attempt to make two nodes in the same position. The root must be an `UpdateObjectNode`.

While parsing, we should check that:

- Each `<id>` has an array filter.
- Each array filter is used by at least one `<id>`.

Example

Update: `{$set: {“a.0”: 0, “a.$”: 1, “b.$[i].c”: 2, “b.$[j].c”: 3},`
`        $inc: {“b.$[i].d”: 1}}`



Execute the update

Each `UpdateNode` implements `apply(mutablebson::Element element, FieldRef* pathToCreate, StringData matchedField, ...)`:

`UpdateObjectNode` iterates through each of its children, recursively calling `apply()` on each child. If `pathToCreate` is empty and the child’s field exists in the current element, it passes that field as `element`. Otherwise, it appends the field to the end of `pathToCreate` and passes the entire current element as `element`. If the field is “$”, it is replaced with `matchedField`. If there was already a child `matchedField`, the children must be cloned and merged, and `apply()` called on the result. The children are merged by combining `UpdateObjectNodes` and `UpdateArrayNodes` along the same path, and erroring with

`ErrorCodes::ConflictingUpdateOperators` if we try to combine two nodes of different types or two `UpdateLeafNodes`. Before moving on to the next child, it checks whether the fields along `pathToCreate` were created, and if so, it resets `element` to the end of `pathToCreate` and resets `pathToCreate` to “”.

The `UpdateLeafNodes` create the fields along `pathToCreate` in `element`, then update the element at the end of `pathToCreate`. They require a list of immutable fields to ensure that they are not updating an immutable field. They must validate the updated element. Note that the `UpdateLeafNodes` need to be responsible for creating the fields along `pathToCreate` because they may require specific behavior. For example, `UnsetNode` does not create the fields along `pathToCreate`. `SetNode` will ignore errors caused by “blocking elements” if the update is from replication, e.g. `{$set: {“a.b”: 0}}` on `{a: 0}` will be ignored. All other `UpdateLeafNodes` will error if attempting to add a field to an element of the wrong type.

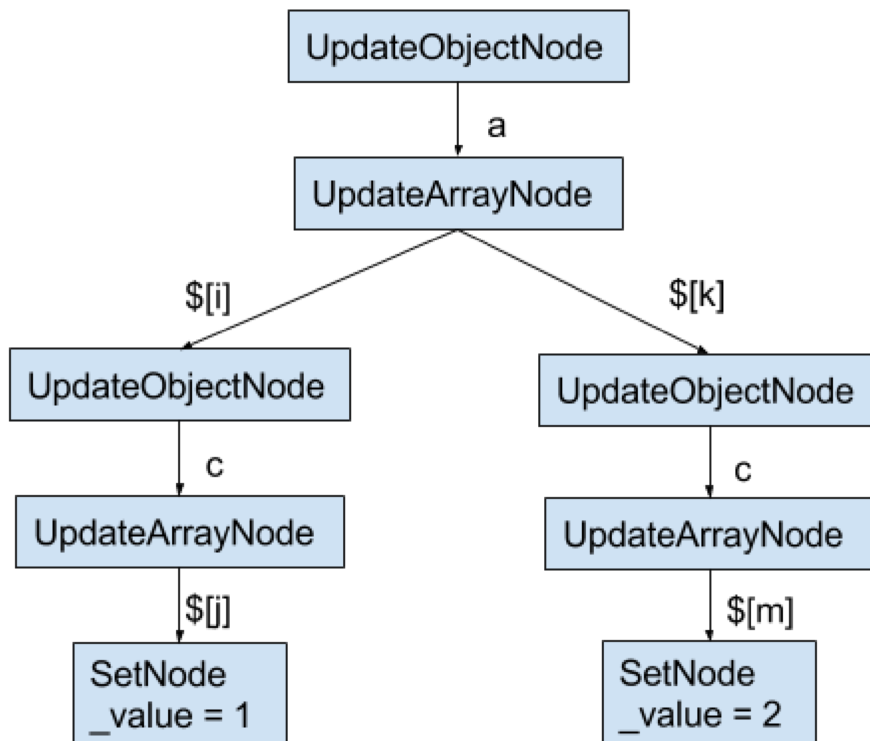
`UpdateArrayNode` requires that `pathToCreate` be empty and `element` be an array (i.e. the array existed in the document prior to update). For each element of the array, it uses the `arrayFilters` to determine the set of children to apply to the element. If only one child matches, we call `apply()` on that child. If multiple children match, it clones and merges the children, and calls `apply()` on the result. The merged set of children should be stored and reused for other array elements and other documents.

Example

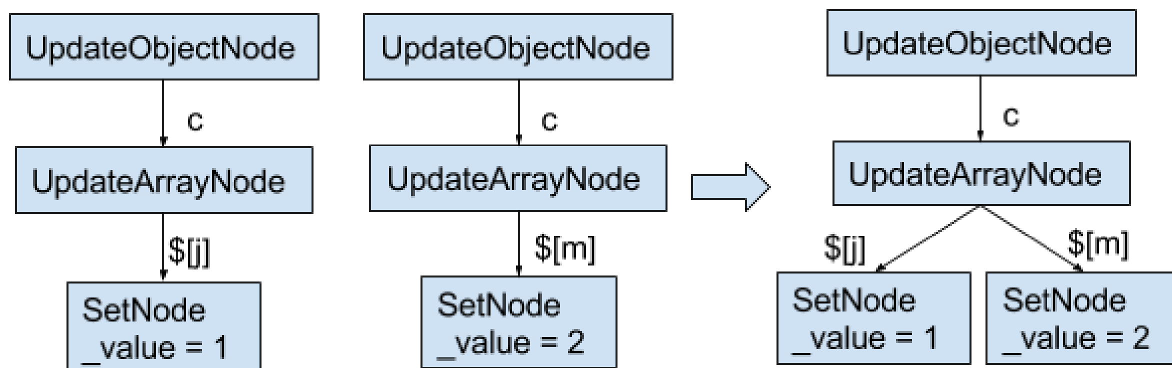
(please forgive the pseudocode)

Document: `{a: [{b: 0, c: [0, 1]}]}`
Update: `{$set: {“a.$[i].c.$[j]”: 1, “a.$[k].c.$[m]”: 2}}`
Array filters: `[{“i.b”: {$gte: 0}}, {j: 0}, {“k.b”: {$lte: 0}}, {m: 1}]`

Initially we have the tree:



`UpdateDriver::update()` calls `UpdateObjectNode::apply({a: [{b: 0, c: [0, 1]}]})`, which calls `UpdateArrayNode::apply([{b: 0, c: [0, 1]}])`. This iterates through each element of the array. It finds that `{"i.b": {$gte: 0}}` and `{"k.b": {$lte: 0}}` both apply to the first element, so it clones and merges those children:



It then calls `UpdateObjectNode::apply({b: 0, c: [0, 1]})`, which calls `ArrayUpdateNode::apply([0, 1])`. This iterates through each element of the array. It finds that only `{j: 0}` applies to the first element, so it calls `SetNode::apply(0)`. It finds that only {m: 1} applies to the second element, so it calls `SetNode::apply(1)`.

Determining whether an array element matches a filter

We must efficiently determine whether an array element matches a filter, e.g. `{$or: [{i: 0}, {"i.a": 1}]}`. To do so, we define a class `BSONElementViewMatchableDocument`, which

inherits from `MatchableDocument`. `BSONElementViewMatchableDocument` is similar to `BSONMatchableDocument`, except that it holds a `BSONElement` rather than a `BSONObj`, and it creates its `BSONElementIterator` after removing the first field in `path`. `BSONElementIterator` must be modified to take a `BSONObj` or `BSONElement`.

We can then use the array element to construct a `BSONElementViewMatchableDocument`, which can be passed to `MatchExpression::matches()`. The result is that `MatchExpression::matches()` will match the array filter to the array element, and each path traversal in the array element will skip the initial “`<id>`”.

Update indexes

The `UpdateDriver` has an `UpdateIndexData` containing key patterns for all indexes. It only wants to know whether any index is affected. `UpdateNode::apply()` should additionally take the path taken to reach `element` and the `UpdateIndexData`, and return a boolean indicating whether any indexes are affected. The `UpdateLeafNodes` are responsible for determining whether indexes are affected, and the internal `UpdateNodes` just OR the responses of their children.

Replication

Oplog entries

Array updates will put the entire array in the oplog entry if multiple elements are modified, or put the modified (nested) element in the oplog entry if a single (nested) element is modified. No changes are required for oplog application. Note that positional `$` updates only put the modified element in the oplog entry.

Examples

A single array element was matched and modified.
Document: `{b: [0, 1]}`
Update: `{$set: {“b.$[i]”: 2}}`
Array filters: `[{i: 0}]`
Oplog entry update: `{$set: {“b.0”: 2}}`

A single array element was matched and modified as an unset.
Document: `{b: [0, 1]}`
Update: `{$unset: {“b.$[i]”: true}}`
Array filters: `[{i: 0}]`
Oplog entry update: `{$unset: {“b.0”: true}}`

A single array element was matched, but was not modified.
Document: `{b: [0, 1]}`
Update: `{$set: {“b.$[i]”: 0}}`
Array filters: `[{i: 0}]`
Oplog entry update: none

Multiple array elements were modified.
Document: `{a: [0, 0]}`
Update: `{$set: {“a.$[i]”: 2}}`
Array filters: `[{i: 0}]`
Oplog entry update: `{$set: {a: [2, 2]}}`

Multiple array elements were modified as an unset.
Document: `{a: [0, 0]}`
Update: `{$unset: {“a.$[i]”: true}}`
Array filters: `[{i: 0}]`
Oplog entry update: `{$set: {a: [null, null]}}`

Multiple array elements were matched, but none were modified.
Document: `{a: [0, 0]}`
Update: `{$set: {“a.$[i]”: 0}}`
Array filters: `[{i: 0}]`
Oplog entry update: none

Multiple array elements were matched, and one was modified.
Document: `{a: [0, 1]}`
Update: `{$set: {“a.$[i]”: 0}}`
Array filters: `[{i: {$gte: 0}}]`
Oplog entry update: `{$set: {“a.1”: 0}}`

A single nested array element was matched and modified.
Document: `{a: [{b: 0, c: [0, 1]}, {b: 1, c: [0, 1]}]}`
Update: `{$set: {“a.$[i].c.$[j]”: 2}}`
Array filters: `[{“i.b”: 0}, {j: 0}]`
Oplog entry update: `{$set: {“a.0.c.0”: 2}}`

Multiple nested array elements were modified.
Document: `{a: [{b: 0, c: [0, 1]}, {b: 0, c: [0, 1]}]}`
Update: `{$set: {“a.$[i].c.$[j]”: 2}}`
Array filters: `[{“i.b”: 0}, {j: 0}]`
Oplog entry update: `{$set: {a: [{b: 0, c: [2, 1]}, {b: 0, c: [2, 1]}]}}`

Multiple nested array elements were matched, but none were modified.
Document: `{a: [{b: 0, c: [0, 1]}, {b: 0, c: [0, 1]}]}`
Update: `{$set: {“a.$[i].c.$[j]”: 0}}`
Array filters: `[{“i.b”: 0}, {j: 0}]`
Oplog entry update: none

Multiple nested array elements were matched, and one was modified. The entire modified array element is included in the oplog entry, though it would be preferable to just include the nested element.
Document: `{a: [{b: 0, c: [1, -1]}, {b: 0, c: [0, -1]}]}`
Update: `{$set: {"a.$[i].c.$[j]": 0}}`
Array filters: `[{"i.b": 0}, {j: {$gte: 0}}]`
Oplog entry update: `{$set: {"a.0": {b: 0, c: [0, -1]}}}`

To implement this, `UpdateNode::apply()` should additionally take the path taken to reach `element` and an optional `LogBuilder`. Each `UpdateLeafNode` records itself in the `LogBuilder`, if it was not a no-op. `UpdateObjectNode` simply passes the `LogBuilder` on to its children. `UpdateArrayNode` first applies the array filters to determine how many elements are matched. Then if only one element was matched, it passes the `LogBuilder` on to that child when it calls `apply()`. Otherwise, it does not pass the `LogBuilder` on to its children when it calls `apply()`. Each child should return whether the operation was a no-op. If multiple children were modified, it records the entire array as a `$set`. If one element was modified, it records that element as a $set. If no elements were modified, it does not produce an oplog entry. This is extensible to creating a knob that allows users to set the maximum number of individual elements that should be individually recorded in the oplog entry.

Sharding

No changes are required for sharding, because the update document is only used for targeting in the case of whole document replacement.

Upgrade/Downgrade Design

What changes will be made to support mixed-version operation, rolling upgrade and downgrade? Think about: On-disk format changes, communication pattern/protocol changes, operation semantics changes, what happens if a new operation is sent to an old node?

`featureCompatibilityVersion` will be used to toggle between the old and the new update implementation. This is needed because the new implementation adds new fields in alphabetical order, whereas the old implementation adds new fields in the order they were specified in the update expression. Then if you have a primary running the old implementation, and a secondary running the new implementation, the fields will end up in different orders on the different nodes. Array filters are only allowed using the new implementation (i.e. when the `featureCompatibilityVersion` is 3.6).

Note that if we send the new path syntax to a 3.4 mongod, we get an error:

```
> db.coll.insert({a: [1]})
WriteResult({ "nInserted" : 1 })
> db.coll.update({}, {$set: {"a.$[i]": 1}})
WriteResult({
      "nMatched" : 0,
```

```
        "nUpserted" : 0,
        "nModified" : 0,
        "writeError" : {
              "code" : 16837,
              "errmsg" : "cannot use the part (a of a.$[i]) to traverse the
element ({a: [ 1.0 ]})"
        }
})
> db.coll.drop()
true
> db.coll.insert({})
WriteResult({ "nInserted" : 1 })
> db.coll.update({}, {$set: {"a.$[i]": 1}})
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 0,
        "nModified" : 0,
        "writeError" : {
              "code" : 52,
              "errmsg" : "The dollar ($) prefixed field '$[i]' in 'a.$[i]' is
not valid for storage."
        }
})
> db.coll.drop()
true
> db.coll.insert({a: 1})
WriteResult({ "nInserted" : 1 })
> db.coll.update({}, {$set: {"a.$[i]": 1}})
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 0,
        "nModified" : 0,
        "writeError" : {
              "code" : 16837,
              "errmsg" : "cannot use the part (a of a.$[i]) to traverse the
element ({a: 1.0})"
        }
})
```

If we send the option `arrayFilters` to a 3.4 mongod, we get an error:

```
> db.runCommand({update: "coll", updates: [{q: {}, u: {$set: {a: 1}},
arrayFilters: []}]})
{
```

```
	"operationTime" : Timestamp(0, 0),
	"ok" : 0,
	"errmsg" : "Unrecognized field in update operation: arrayFilters",
	"code" : 9,
	"codeName" : "FailedToParse"
}
```

Thus if the `featureCompatibilityVersion` is 3.4, the update will fail.

Note that the shell silently does not pass through unknown update options. Then on old versions of the shell connected to a 3.6 version of the server, `db.c.update({}, {$set: {“a.$[i]”: 1}}, {arrayFilters: [{i: 0}]})` will report an error, since the server will receive no array filter for “`i`”. However, `db.c.update({}, {$set: {“a.$[]”: 1}})` will succeed and update every element of “`a`”.

Test Plan

The purpose of developing a test plan is to mitigate the risk of the code changes being made. It's important to focus testing efforts on high-value scenarios and test cases because the set of tests that could be performed is infinite.

Functional Testing

One way to classify the kinds of correctness test cases that need to be written is to consider the impact of the code changes on different areas of the system. Think about: chunk migrations, initial sync, data replication, replication rollback, query execution/yielding, index builds. Please email the Test Infrastructure team if you'd like to discuss the capabilities of our existing testing infrastructure (e.g. resmoke.py, jstestfuzz, concurrency suite) or have additional requirements for it as part of this project.

Much of the implementation is unit testable. In particular, we should unit test parsing of the initial update and merging of two `UpdateNodes`:

- “`.0`”, “`.$`”, and “`.<string>`” parse as `UpdateObjectNodes`.
- “`.$[<id>]`” and “`.$[]`” parse as `UpdateArrayNodes`.
- Merging two `UpdateObjectNodes` succeeds and children with the same field name are merged.
- Merging two `UpdateArrayNodes` succeeds and children with the same `ArrayFilter` are merged.
- Merging two `UpdateLeafNodes` fails.
- Merging two `UpdateNodes` of different types fails.

We should add integration tests to the core suite for all the examples and error cases mentioned in the Behavioral Description. We should also be sure to test the following nested array update cases:

- “`a.$[i].b.$[k].c`” and “`a.$[j].b.$[k].d`” are not a conflict, even if `i` and `j` are not disjoint.
- “`a.$[i].b.$[k].c`” and “`a.$[j].b.$[k].c`” are not a conflict if `i` and `j` are disjoint.
- “`a.$[i].b.$[k].c`” and “`a.$[j].b.$[k].c`” are a conflict if `i` and `j` are not disjoint.

- “`a.$[i].b.$[k].c`” and “`a.$[j].b.$[m].c`” are not a conflict if `k` and `m` are disjoint for each element of a matching `i` and `j`.
- “`a.$[i].b.$[k].c`” and “`a.$[j].b.$[m].c`” are a conflict if `k` and `m` intersect for some element of a matching `i` and `j`.

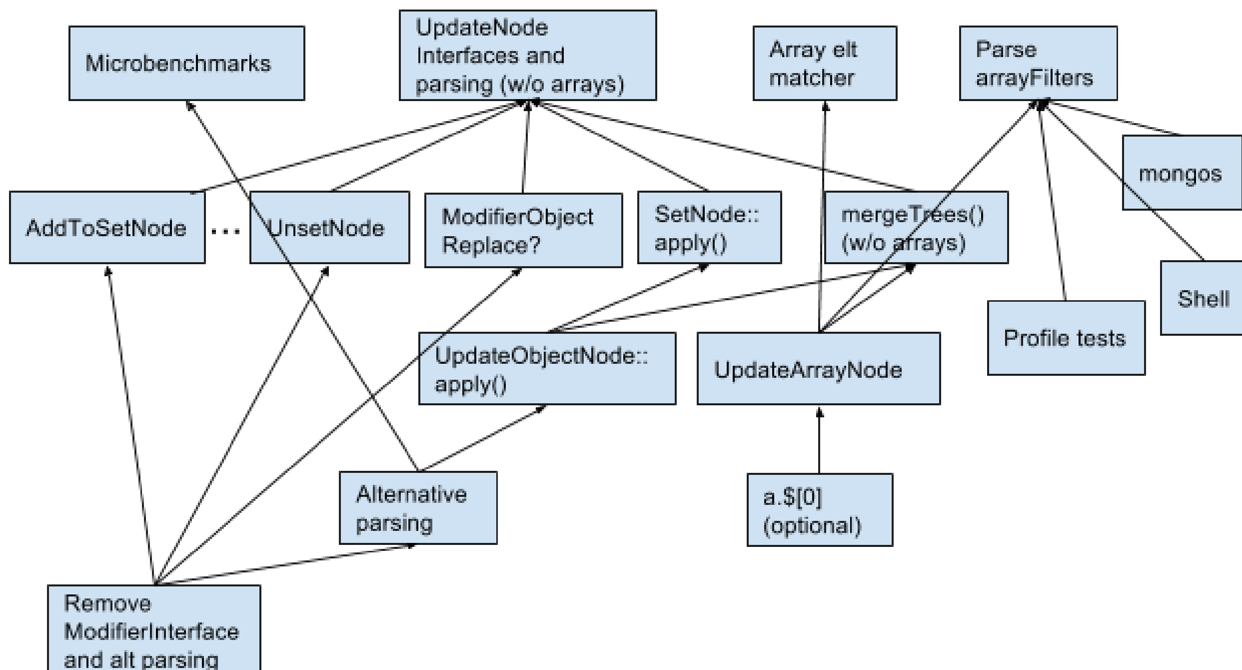
We should do multiversion testing for mixed version replication. A 3.4 primary should error when it receives array updates. A 3.6 primary should succeed when it receives array updates, and the updates should correctly replicate to a 3.4 secondary.

Performance Testing

For updating single-element arrays, we should not be slower than positional $ updates. Microbenchmarks for positional $updates and nested document updates should be added before this work.

Implementation Plan/Schedule

Remember, for complex projects upgrade/downgrade can take 1-2 engineers 2 months.



Security Implications

N/A

External Dependencies

N/A

Downstream Changes

Cloud Manager

1. Any changes to configuration options?
    a. No
2. Any changes to the oplog schema or the applyOps command?
    a. No
3. Any changes that affect the format of data stored on disk?
    a. No

Changes to log lines for individual updates/deletes

For update (command and legacy):

- Remove fields: query, update, and collation.
- Add field: command: {q: <query>, u: <update>, upsert: <boolean>, multi: <boolean>, collation: <collation>, arrayFilters: <filters>}.

For delete (command and legacy):

- Remove fields: query and collation.
- Add field: command: {q: <query>, limit: <integer>, collation: <collation>}.

For all operations (command and legacy):

- Rename query field to command.

Changes to profiler entries

For update (command and legacy):

- Remove fields: query, updateobj, and collation.
- Add field: command: {q: <query>, u: <update>, upsert: <boolean>, multi: <boolean>, collation: <collation>, arrayFilters: <filters>}.

For delete (command and legacy):

- Remove fields: query and collation.
- Add field: command: {q: <query>, limit: <integer>, collation: <collation>}.

For all operations (command and legacy):

- Rename query field to command.

Changes to currentOp

For update (command and legacy):

- Remove fields: query and collation.
- Add field: command: {q: <query>, u: <update>, upsert: <boolean>, multi: <boolean>, collation: <collation>, arrayFilters: <filters>}.

For delete (command and legacy):

- Remove fields: query and collation.
- Add field: command: {q: <query>, limit: <integer>, collation: <collation>}.

For all operations (command and legacy):

- Rename `query/insert` field to `command`.

Drivers

1. Any new authentication mechanisms?
    a. No
2. Any changes at all to the public API of the server?[1]
    a. Yes, a new path syntax in update modifiers, and a new option to the update command.
3. Any changes to the BSON spec?
    a. No
4. Any changes to the extended JSON spec?
    a. No
5. Any wire protocol changes?
    a. No

Compass

1. Any changes to $sample or aggregation defaults?
    a. No
2. Any changes to explain output (all three verbosity levels)?
    a. No
3. Any changes to document validation rules?
    a. No
4. Any changes to the user privileges document returned by *connectionStatus* and *usersInfo* commands with *{showPrivileges:true}*?
    a. No

---

[1] For example, are you changing the types of any values returned in a command response? Another example, are you adding any new aggregation pipeline stages, accumulators, or expressions?