Aggregate

Aggregate()

Parameters

- [pipeline] «Array» aggregation pipeline as an array of objects
- [model] «Model» the model to use with this aggregate.

Aggregate constructor used for building aggregation pipelines. Do not instantiate this class directly, use Model.aggregate() instead.

Example:

```
const aggregate = Model.aggregate([
  { $project: { a: 1, b: 1 } },
  { $skip: 5 }
]);

Model.
  aggregate([{ $match: { age: { $gte: 21 }}}]).
  unwind('tags').
  exec(callback);
```

Note:

- The documents returned are plain javascript objects, not mongoose documents (since any shape of document can be returned).
- Mongoose does **not** cast pipeline stages. The below will **not** work unless `_id` is a string in the database

```
new Aggregate([{ $match: { _id: '00000000000000000000000a' } }]);
// Do this instead to cast to an ObjectId
new Aggregate([{ $match: { _id: mongoose.Types.ObjectId('00000000000000000000000a') } }]);
```

Aggregate.prototype.Symbol.asyncIterator()

Returns an asyncIterator for use with [`for/await/of` loops](https://thecodebarbarian.com/getting-started-with-async-iterators-in-node-js You do not need to call this function explicitly, the JavaScript runtime will call it for you.

Example

```
const agg = Model.aggregate([{ $match: { age: { $gte: 25 } } }]);
for await (const doc of agg) {
  console.log(doc.name);
}
```

Node.js 10.x supports async iterators natively without any flags. You can enable async iterators in Node.js 8.x using the `--harmony_async_iteration` flag.

Note: This function is not set if `Symbol.asyncIterator` is undefined. If `Symbol.asyncIterator` is undefined, that means your Node.js version does not support async iterators.

Aggregate.prototype.addFields()

Parameters

- arg «Object» field specification

Returns:

- «Aggregate»

Appends a new $addFields operator to this aggregate pipeline. Requires MongoDB v3.4+ to work

Examples:

```
// adding new fields based on existing fields
aggregate.addFields({
    newField: '$b.nested'
  , plusTen: { $add: ['$val', 10]}
  , sub: {
       name: '$a'
    }
})

// etc
aggregate.addFields({ salary_k: { $divide: [ "$salary", 1000 ] } });
```

Aggregate.prototype.allowDiskUse()

Parameters

- value «Boolean» Should tell server it can use hard drive to store data during aggregation.
- [tags] «Array» optional tags for this query

Sets the allowDiskUse option for the aggregation query (ignored for < 2.6.0)

Example:

```
await Model.aggregate([{ $match: { foo: 'bar' } }]).allowDiskUse(true);
```

Aggregate.prototype.append()

Parameters

- ops «Object» operator(s) to append

Returns:

- «Aggregate»

Appends new operators to this aggregate pipeline

Examples:

```
aggregate.append({ $project: { field: 1 }}, { $limit: 2 });

// or pass an array
const pipeline = [{ $match: { daw: 'Logic Audio X' }} ];
aggregate.append(pipeline);
```

Aggregate.prototype.catch()

Parameters

- [reject] «Function»

Returns:

- «Promise»

Executes the query returning a `Promise` which will be resolved with either the doc(s) or rejected with the error. Like `.then()`, but only takes a rejection handler.

Aggregate.prototype.collation()

Parameters

- collation «Object» options

Returns:

- «Aggregate» this

Adds a collation

Example:

```
const res = await Model.aggregate(pipeline).collation({ locale: 'en_US', strength: 1 });
```

Aggregate.prototype.count()

Parameters

- the «String» name of the count field

Returns:

- «Aggregate»

Appends a new $count operator to this aggregate pipeline.

Examples:

```
aggregate.count("userCount");
```

Aggregate.prototype.cursor()

Parameters

- options «Object»
- options.batchSize «Number» set the cursor batch size
    - [options.useMongooseAggCursor] «Boolean» use experimental mongoose-specific aggregation cursor (for `eachAsync()` and other query cursor semantics)

Returns:

- «AggregationCursor» cursor representing this aggregation

Sets the `cursor` option and executes this aggregation, returning an aggregation cursor. Cursors are useful if you want to process the results of the aggregation one-at-a-time because the aggregation result is too big to fit into memory.

Example:

```
const cursor = Model.aggregate(..).cursor({ batchSize: 1000 });
cursor.eachAsync(function(doc, i) {
  // use doc
});
```

Aggregate.prototype.exec()

Parameters

- [callback] «Function»

Returns:

- «Promise»

Executes the aggregate pipeline on the currently bound Model.

Example:

```
aggregate.exec(callback);

// Because a promise is returned, the `callback` is optional.
const promise = aggregate.exec();
promise.then(..);
```

Aggregate.prototype.explain()

Parameters

- callback «Function»

Returns:

- «Promise»

Execute the aggregation with explain

Example:

```
Model.aggregate(..).explain(callback)
```

Aggregate.prototype.facet()

Parameters

- facet «Object» options

Returns:

- «Aggregate» this

Combines multiple aggregation pipelines.

Example:

```
const res = await Model.aggregate().facet({
  books: [{ groupBy: '$author' }],
  price: [{ $bucketAuto: { groupBy: '$price', buckets: 2 } }]
```

});

```
// Output: { books: [...], price: [{...}, {...}] }
```

Aggregate.prototype.graphLookup()

Parameters

- options «Object» to $graphLookup as described in the above link

Returns:

- «Aggregate»

Appends new custom $graphLookup operator(s) to this aggregate pipeline, performing a recursive search on a collection.

Note that graphLookup can only consume at most 100MB of memory, and does not allow disk use even if `{ allowDiskUse: true }` is specified.

Examples:

```
// Suppose we have a collection of courses, where a document might look like `{ _id: 0, name: 'Ca
 aggregate.graphLookup({ from: 'courses', startWith: '$prerequisite', connectFromField: 'prerequis
```

Aggregate.prototype.group()

Parameters

- arg «Object» $group operator contents

Returns:

- «Aggregate»

Appends a new custom $group operator to this aggregate pipeline.

Examples:

```
aggregate.group({ _id: "$department" });
```

Aggregate.prototype.hint()

Parameters

- value «Object|String» a hint object or the index name

Sets the hint option for the aggregation query (ignored for < 3.6.0)

Example:

```
Model.aggregate(..).hint({ qty: 1, category: 1 }).exec(callback)
```

Aggregate.prototype.limit()

Parameters

- num «Number» maximum number of records to pass to the next stage

Returns:

- «Aggregate»

Appends a new $limit operator to this aggregate pipeline.

Examples:

```
aggregate.limit(10);
```

Aggregate.prototype.lookup()

Parameters

- options «Object» to $lookup as described in the above link

Returns:

- «Aggregate*» @api public

Appends new custom $lookup operator to this aggregate pipeline.

Examples:

```
aggregate.lookup({ from: 'users', localField: 'userId', foreignField: '_id', as: 'users' });
```

Aggregate.prototype.match()

Parameters

- arg «Object» $match operator contents

Returns:

- «Aggregate»

Appends a new custom $match operator to this aggregate pipeline.

Examples:

```
aggregate.match({ department: { $in: [ "sales", "engineering" ] } });
```

Aggregate.prototype.model()

Parameters

- [model] «Model» set the model associated with this aggregate.

Returns:

- «Model»

Get/set the model that this aggregation will execute on.

Example:

```
const aggregate = MyModel.aggregate([{ $match: { answer: 42 } }]);
aggregate.model() === MyModel; // true

// Change the model. There's rarely any reason to do this.
aggregate.model(SomeOtherModel);
aggregate.model() === SomeOtherModel; // true
```

Aggregate.prototype.near()

Parameters

- arg «Object»

Returns:

- «Aggregate»

Appends a new $geoNear operator to this aggregate pipeline.

NOTE:

MUST be used as the first operator in the pipeline.

Examples:

```
aggregate.near({
  near: [40.724, -73.997],
  distanceField: "dist.calculated", // required
  maxDistance: 0.008,
  query: { type: "public" },
  includeLocs: "dist.location",
  uniqueDocs: true,
  num: 5
});
```

Aggregate.prototype.option()

Parameters

- options «Object» keys to merge into current options
- number «[options.maxTimeMS]» limits the time this aggregation will run, see MongoDB docs on `maxTimeMS`
- boolean «[options.allowDiskUse]» if true, the MongoDB server will use the hard drive to store data during this aggregation
- object «[options.collation]» see `Aggregate.prototype.collation()`
- ClientSession «[options.session]» see `Aggregate.prototype.session()`

Returns:

- «Aggregate» this

Lets you set arbitrary options, for middleware or plugins.

Example:

```
const agg = Model.aggregate(..).option({ allowDiskUse: true }); // Set the `allowDiskUse` option
agg.options; // `{ allowDiskUse: true }`
```

Aggregate.prototype.options

Type:

- «property»

Contains options passed down to the aggregate command.

Supported options are

- `readPreference`
- `cursor`
- `explain`
- `allowDiskUse`
- `maxTimeMS`
- `bypassDocumentValidation`
- `raw`
- `promoteLongs`
- `promoteValues`
- `promoteBuffers`
- `collation`
- `comment`
- `session`

Aggregate.prototype.pipeline()

Returns:

- «Array»

Returns the current pipeline

Example:

```
MyModel.aggregate().match({ test: 1 }).pipeline(); // [{ $match: { test: 1 } }]
```

Aggregate.prototype.project()

Parameters

- arg «Object|String» field specification

Returns:

- «Aggregate»

Appends a new $project operator to this aggregate pipeline.

Mongoose query selection syntax is also supported.

Examples:

```
// include a, include b, exclude _id
aggregate.project("a b -_id");

// or you may use object notation, useful when
// you have keys already prefixed with a "-"
aggregate.project({a: 1, b: 1, _id: 0});

// reshaping documents
aggregate.project({
    newField: '$b.nested'
  , plusTen: { $add: ['$val', 10]}
  , sub: {
       name: '$a'
    }
})

// etc
aggregate.project({ salary_k: { $divide: [ "$salary", 1000 ] } });
```

Aggregate.prototype.read()

Parameters

- pref «String» one of the listed preference options or their aliases
- [tags] «Array» optional tags for this query

Returns:

- «Aggregate» this

Sets the readPreference option for the aggregation query.

Example:

```
await Model.aggregate(pipeline).read('primaryPreferred');
```

Aggregate.prototype.readConcern()

Parameters

- level «String» one of the listed read concern level or their aliases

Returns:

- «Aggregate» this

Sets the readConcern level for the aggregation query.

Example:

```
await Model.aggregate(pipeline).readConcern('majority');
```

Aggregate.prototype.redact()

Parameters

- expression «Object» redact options or conditional expression
- [thenExpr] «String|Object» true case for the condition
- [elseExpr] «String|Object» false case for the condition

Returns:

- «Aggregate» this

Appends a new $redact operator to this aggregate pipeline.

If 3 arguments are supplied, Mongoose will wrap them with if-then-else of $cond operator respectively If `thenExpr` or `elseExpr` is string, make sure it starts with $$, like `$$DESCEND`, `$$PRUNE` or `$$KEEP`.

Example:

```
await Model.aggregate(pipeline).redact({
  $cond: {
    if: { $eq: [ '$level', 5 ] },
    then: '$$PRUNE',
    else: '$$DESCEND'
  }
});

// $redact often comes with $cond operator, you can also use the following syntax provided by mong
await Model.aggregate(pipeline).redact({ $eq: [ '$level', 5 ] }, '$$PRUNE', '$$DESCEND');
```

Aggregate.prototype.replaceRoot()

Parameters

- the «String|Object» field or document which will become the new root document

Returns:

- «Aggregate»

Appends a new $replaceRoot operator to this aggregate pipeline.

Note that the `$replaceRoot` operator requires field strings to start with '$'. If you are passing in a string Mongoose will prepend '$' if the specified field doesn't start '$'. If you are passing in an object the strings in your expression will not be altered.

Examples:

```
aggregate.replaceRoot("user");

aggregate.replaceRoot({ x: { $concat: ['$this', '$that'] } });
```

Aggregate.prototype.sample()

Parameters

- size «Number» number of random documents to pick

Returns:

- «Aggregate»

Appends new custom $sample operator to this aggregate pipeline.

Examples:

```
aggregate.sample(3); // Add a pipeline that picks 3 random documents
```

Aggregate.prototype.search()

Parameters

- $search «Object» options

Returns:

- «Aggregate» this

Helper for Atlas Text Search's `$search` stage.

Example:

```
const res = await Model.aggregate().
 search({
   text: {
     query: 'baseball',
     path: 'plot'
   }
 });

// Output: [{ plot: '...', title: '...' }]
```

Aggregate.prototype.session()

Parameters

- session «ClientSession»

Sets the session for this aggregation. Useful for transactions.

Example:

```
const session = await Model.startSession();
await Model.aggregate(..).session(session);
```

Aggregate.prototype.skip()

Parameters

- num «Number» number of records to skip before next stage

Returns:

- «Aggregate»

Appends a new $skip operator to this aggregate pipeline.

Examples:

```
aggregate.skip(10);
```

Aggregate.prototype.sort()

Parameters

- arg «Object|String»

Returns:

- «Aggregate» this

Appends a new $sort operator to this aggregate pipeline.

If an object is passed, values allowed are `asc`, `desc`, `ascending`, `descending`, `1`, and `-1`.

If a string is passed, it must be a space delimited list of path names. The sort order of each path is ascending unless the path name is prefixed with `-` which will be treated as descending.

Examples:

```
// these are equivalent
aggregate.sort({ field: 'asc', test: -1 });
aggregate.sort('field -test');
```

Aggregate.prototype.sortByCount()

Parameters

- arg «Object|String»

Returns:

- «Aggregate» this

Appends a new $sortByCount operator to this aggregate pipeline. Accepts either a string field name or a pipeline object.

Note that the `$sortByCount` operator requires the new root to start with '$'. Mongoose will prepend '$' if the specified field name doesn't start with '$'.

Examples:

```
aggregate.sortByCount('users');
aggregate.sortByCount({ $mergeObjects: [ "$employee", "$business" ] })
```

Aggregate.prototype.then()

Parameters

- [resolve] «Function» successCallback
- [reject] «Function» errorCallback

Returns:

- «Promise»

Provides promise for aggregate.

Example:

```
Model.aggregate(..).then(successCallback, errorCallback);
```

Aggregate.prototype.unwind()

Parameters

- fields «String|Object» the field(s) to unwind, either as field names or as objects with options. If passing a string, prefixing the field name with '$' is optional. If passing an object, `path` must start with '$'.

Returns:

- «Aggregate»

Appends new custom $unwind operator(s) to this aggregate pipeline.

Note that the `$unwind` operator requires the path name to start with '$'. Mongoose will prepend '$' if the specified field doesn't start '$'.

Examples:

```
aggregate.unwind("tags");
aggregate.unwind("a", "b", "c");
aggregate.unwind({ path: '$tags', preserveNullAndEmptyArrays: true });
```