Otimização de pipeline de agregação

Nesta página

As operações de pipeline de agregação têm uma fase de otimização que tenta remodelar o pipeline para melhorar o desempenho.

Para ver como o otimizador transforma um determinado pipeline de agregação, inclua a `explain`opção no `db.collection.aggregate()`método

As otimizações estão sujeitas a alterações entre os releases.

Otimização de projeção

O pipeline de agregação pode determinar se requer apenas um subconjunto dos campos nos documentos para obter os resultados. Nesse caso, o pipeline usará apenas os campos obrigatórios, reduzindo a quantidade de dados que passam pelo pipeline.

Otimização de sequência de pipeline

(`$project`ou `$unset`ou `$addFields`ou `$set`) + `$match`Otimização de sequência

Para um pipeline de agregação que contém um estágio de projeção (`$project`ou `$unset`ou `$addFields`ou `$set`) seguido por um `$match`estágio, o MongoDB move quaisquer filtros no `$match`estágio que não exijam valores calculados no estágio de projeção para um novo `$match`estágio antes da projeção.

Se um pipeline de agregação contiver várias projeções e / ou `$match`estágios, o MongoDB executará essa otimização para cada `$match`estágio, movendo cada `$match`filtro antes de todos os estágios de projeção dos quais o filtro não depende.

Considere um pipeline dos seguintes estágios:

```
{ $addFields: {
    maxTime: { $max: "$times" },
    minTime: { $min: "$times" }
} },
{ $project: {
    _id: 1, name: 1, times: 1, maxTime: 1, minTime: 1,
    avgTime: { $avg: ["$maxTime", "$minTime"] }
} },
{ $match: {
    name: "Joe Schmoe",
    maxTime: { $lt: 20 },
    minTime: { $gt: 5 },
    avgTime: { $gt: 7 }
} }
```

The optimizer breaks up the `$match` stage into four individual filters, one for each key in the `$match` query document. The optimizer then moves each filter before as many projection stages as possible, creating new `$match` stages as needed. Given this example, the optimizer produces the following *optimized* pipeline:

```
{ $match: { name: "Joe Schmoe" } },
{ $addFields: {
    maxTime: { $max: "$times" },
    minTime: { $min: "$times" }
} },
{ $match: { maxTime: { $lt: 20 }, minTime: { $gt: 5 } } },
{ $project: {
    _id: 1, name: 1, times: 1, maxTime: 1, minTime: 1,
    avgTime: { $avg: ["$maxTime", "$minTime"] }
} },
{ $match: { avgTime: { $gt: 7 } } }
```

The `$match` filter `{ avgTime: { $gt: 7 } }` depends on the `$project` stage to compute the `avgTime` field. The `$project` stage is the last projection stage in this pipeline, so the `$match` filter on `avgTime` could not be moved.

The maxTime and minTime fields are computed in the $project stage. The optimizer created a new $matcl before the $project stage.

The $match filter { name: "Joe Schmoe" } does not use any values computed in either the $project or $addFields stages so it was moved to a new $match stage before both of the projection stages.

> NOTE:
>
> After optimization, the filter { name: "Joe Schmoe" } is in a $match stage at the beginning of the pipeline. This has the added benefit of allowing the aggregation to use an index on the name field when initially querying the collection. See Pipeline Operators and Indexes for more information.

$sort + $match Sequence Optimization

When you have a sequence with $sort followed by a $match, the $match moves before the $sort to minimize the number of objects to sort. For example, if the pipeline consists of the following stages:

```
{ $sort: { age : -1 } },
{ $match: { status: 'A' } }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $match: { status: 'A' } },
{ $sort: { age : -1 } }
```

$redact + $match Sequence Optimization

When possible, when the pipeline has the $redact stage immediately followed by the $match stage, the aggregation can sometimes add a portion of the $match stage before the $redact stage. If the added $match stage is at the start of a pipeline, the aggregation can use an index as well as query the collection to limit the number of documents that enter the pipeline. See Pipeline Operators and Indexes for more information.

For example, if the pipeline consists of the following stages:

```
{ $redact: { $cond: { if: { $eq: [ "$level"
{ $match: { year: 2014, category: { $ne: "Z
```

The optimizer can add the same `$match` stage before the `$redact` stage:

```
{ $match: { year: 2014 } },
{ $redact: { $cond: { if: { $eq: [ "$level", 5 ] }, then: "$$PRUNE", else: "$$DESCEND"
{ $match: { year: 2014, category: { $ne: "Z" } } }
```

`$project`/`$unset` + `$skip` Sequence Optimization

New in version 3.2.

When you have a sequence with `$project` or `$unset` followed by `$skip`, the `$skip` moves before `$project`. For example, if the pipeline consists of the following stages:

```
{ $sort: { age : -1 } },
{ $project: { status: 1, name: 1 } },
{ $skip: 5 }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $sort: { age : -1 } },
{ $skip: 5 },
{ $project: { status: 1, name: 1 } }
```

Pipeline Coalescence Optimization

When possible, the optimization phase coalesces a pipeline stage into its predecessor. Generally, coalescence occurs *after* any sequence reordering optimization.

`$sort` + `$limit` Coalescence

Changed in version 4.0.

When a `$sort` precedes a `$limit`, the optimizer can intervening stages modify the number of documents (e.g the `$limit` into the `$sort` if there are pipeline stages that change the number of documents between the `$sort` and `$limit` stages..

For example, if the pipeline consists of the following stages:

```
{ $sort : { age : -1 } },
{ $project : { age : 1, status : 1, name : 1 } },
{ $limit: 5 }
```

During the optimization phase, the optimizer coalesces the sequence to the following:

```
{
    "$sort" : {
       "sortKey" : {
          "age" : -1
       },
       "limit" : NumberLong(5)
    }
},
{ "$project" : {
         "age" : 1,
         "status" : 1,
         "name" : 1
  }
}
```

This allows the sort operation to only maintain the top `n` results as it progresses, where `n` is the specified limit, and MongoDB only needs to store `n` items in memory [1]. See $sort Operator and Memory for more information.

SEQUENCE OPTIMIZATION WITH $SKIP:

If there is a `$skip` stage between the `$sort` and `$limit` stages, MongoDB will coalesce the `$limit` into the `$sort` stage and increase the `$limit` value by the `$skip` amount. See $sort + $skip + $limit Sequence for an example.

[1] The optimization will still apply when allowDiskUse is t

`$limit` + `$limit` Coalescence

When a `$limit` immediately follows another `$limit`, the two stages can coalesce into a single `$limit` where the limit amount is the *smaller* of the two initial limit amounts. For example, a pipeline contains the following sequence:

```
{ $limit: 100 },
{ $limit: 10 }
```

Then the second `$limit` stage can coalesce into the first `$limit` stage and result in a single `$limit` stage where the limit amount `10` is the minimum of the two initial limits `100` and `10`.

```
{ $limit: 10 }
```

`$skip` + `$skip` Coalescence

When a `$skip` immediately follows another `$skip`, the two stages can coalesce into a single `$skip` where the skip amount is the *sum* of the two initial skip amounts. For example, a pipeline contains the following sequence:

```
{ $skip: 5 },
{ $skip: 2 }
```

Then the second `$skip` stage can coalesce into the first `$skip` stage and result in a single `$skip` stage where the skip amount `7` is the sum of the two initial limits `5` and `2`.

```
{ $skip: 7 }
```

`$match` + `$match` Coalescence

When a `$match` immediately follows another `$match`, the two stages can coalesce into a single `$match` combining the conditions with an `$and`. For example, a pipeline contains the following sequence:

```
{ $match: { year: 2014 } },
{ $match: { status: "A" } }
```

Then the second `$match` stage can coalesce into the first `$match` stage and result in a single `$match` stage

```
{ $match: { $and: [ { "year" : 2014 }, { "status" : "A" } ] } }
```

`$lookup` + `$unwind` Coalescence

New in version 3.2.

When a `$unwind` immediately follows another `$lookup`, and the `$unwind` operates on the `as` field of the `$lookup`, the optimizer can coalesce the `$unwind` into the `$lookup` stage. This avoids creating large intermediate documents.

For example, a pipeline contains the following sequence:

```
{
  $lookup: {
    from: "otherCollection",
    as: "resultingArray",
    localField: "x",
    foreignField: "y"
  }
},
{ $unwind: "$resultingArray"}
```

The optimizer can coalesce the `$unwind` stage into the `$lookup` stage. If you run the aggregation with `explain` option, the `explain` output shows the coalesced stage:

```
{
  $lookup: {
    from: "otherCollection",
    as: "resultingArray",
    localField: "x",
    foreignField: "y",
    unwinding: { preserveNullAndEmptyArrays: false }
  }
}
```

Example

`$sort` + `$skip` + `$limit` Sequence

A pipeline contains a sequence of `$sort` followed by a `$skip` followed by a `$limit`:

```
{ $sort: { age : -1 } },
{ $skip: 10 },
{ $limit: 5 }
```

The optimizer performs $sort + $limit Coalescence to transforms the sequence to the following:

```
{
   "$sort" : {
      "sortKey" : {
         "age" : -1
      },
      "limit" : NumberLong(15)
   }
},
{
   "$skip" : NumberLong(10)
}
```

O MongoDB aumenta a `$limit`quantidade com a reordenação.

VEJA TAMBÉM:

`explain` opção no `db.collection.aggregate`