

2019 数据库系统实现技术

期末作业



学院：计算机科学与工程学院

专业：计算机科学与技术

班级：计算机 1606 班

姓名：戚子强

学号：20164625

一、MongoDB 分析

1.1 MongoDB 概述

MongoDB, 官方网站: <https://www.mongodb.com/>, GitHub 仓库地址: <https://github.com/mongodb/mongo/>。MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写。旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。MongoDB 是一个介于关系数据库和非关系数据库之间的产品, 是非关系数据库当中功能最丰富, 最像关系数据库的。它支持的数据结构非常松散, 是类似 json 的 bson 格式, 因此可以存储比较复杂的数据类型。Mongo 最大的特点是它支持的查询语言非常强大, 其语法有点类似于面向对象的查询语言, 几乎可以实现类似关系数据库单表查询的绝大部分功能, 而且还支持对数据建立索引。

它的特点是高性能、易部署、易使用, 存储数据非常方便。主要功能特性有:

- 面向集合存储, 易存储对象类型的数据;
- 模式自由;
- 支持动态查询;
- 支持完全索引, 包含内部对象;
- 支持查询;
- 支持复制和故障恢复;
- 使用高效的二进制数据存储, 包括大型对象 (如视频等);
- 自动处理碎片, 以支持云计算层次的扩展性;
- 支持 Golang, RUBY, PYTHON, JAVA, C++, PHP, C# 等多种语言;
- 文件存储格式为 BSON (一种 JSON 的扩展);
- 可通过网络访问, 等等。

其中, 所谓“面向集合” (Collection-Oriented), 意思是数据被分组存储在数据集中, 被称为一个集合 (Collection)。每个集合在数据库中都有一个唯一的标识名, 并且可以包含无限数目的文档。集合的概念类似关系型数据库 (RDBMS) 里的表 (table), 不同的是它不需要定义任何模式 (schema)。

Nytro MegaRAID 技术中的闪存高速缓存算法, 能够快速识别数据库内大数据集中的热数据, 提供一致的性能改进。模式自由 (Schema-Free), 意味着对于存储在 MongoDB 数据库中的文件, 我们不需要知道它的任何结构定义。如果需要的话, 你完全可以把不同结构的文件存储在同一个数据库里。存储在集合中的文档, 被存储为键-值对的形式。键用于唯一标识一个文档, 为字符串类型, 而值则可以是各种复杂的文件类型。我们称这种存储形式为 BSON (Binary Serialized Document Format)。MongoDB 已经在多个站点部署, 其主要场景如下:

(1) 网站实时数据处理。它非常适合实时的插入、更新与查询, 并具备网站实时数据存储所需的复制及高度伸缩性。

(2) 缓存。由于性能很高, 它适合作为信息基础设施的缓存层。在系统重启之后, 由它搭建的持久化缓存层可以避免下层的数据源过载。

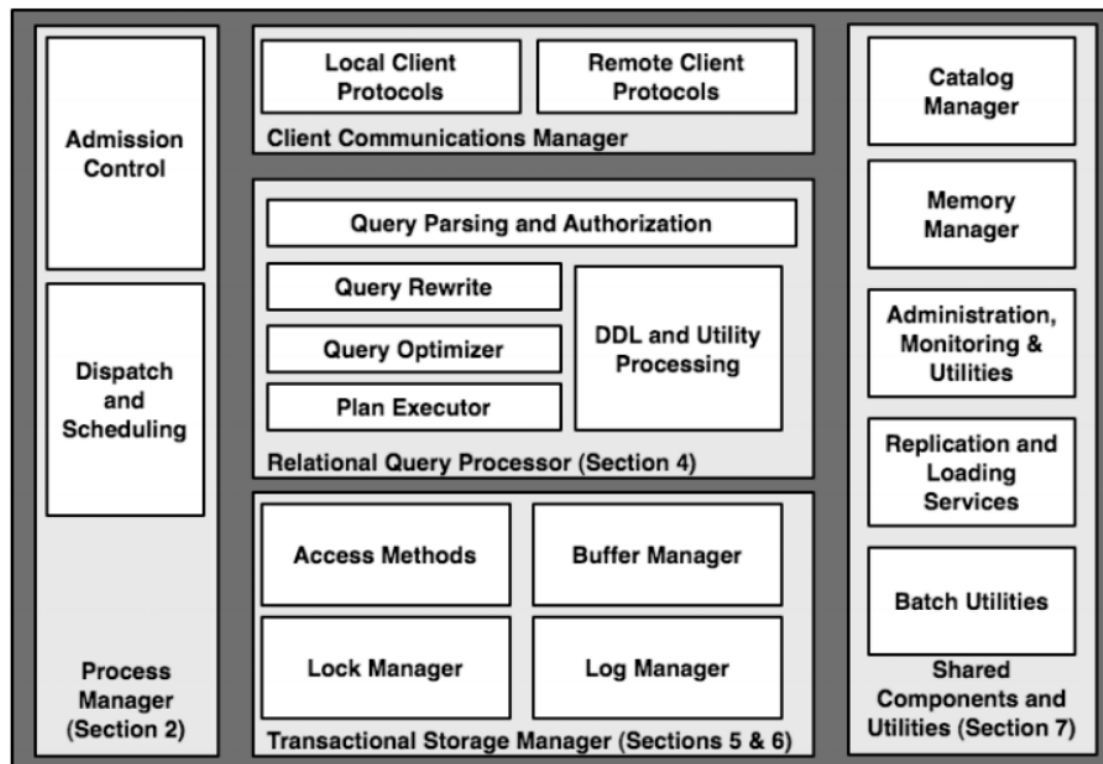
(3) 高伸缩性的场景。非常适合由数十或数百台服务器组成的数据库，它的路线图中已经包含对 MapReduce 引擎的内置支持。

不适用的场景如下：

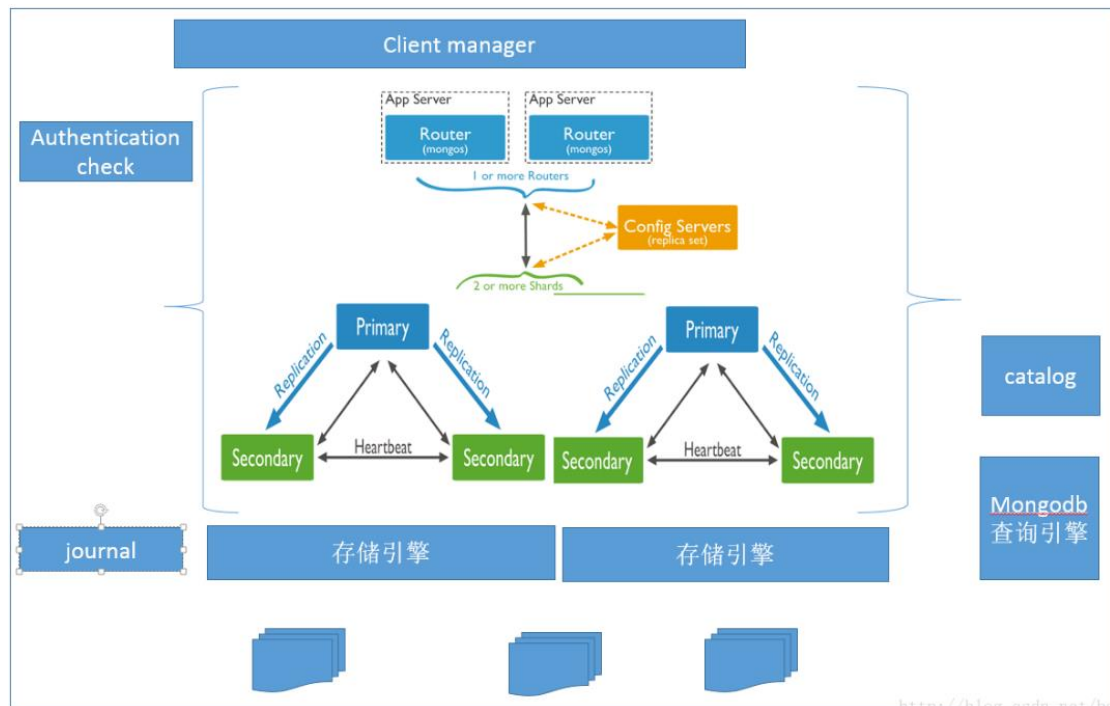
- (1) 要求高度事务性的系统。
- (2) 传统的商业智能应用。
- (3) 复杂的跨文档（表）级联查询。

1.2 MongoDB 系统体系结构

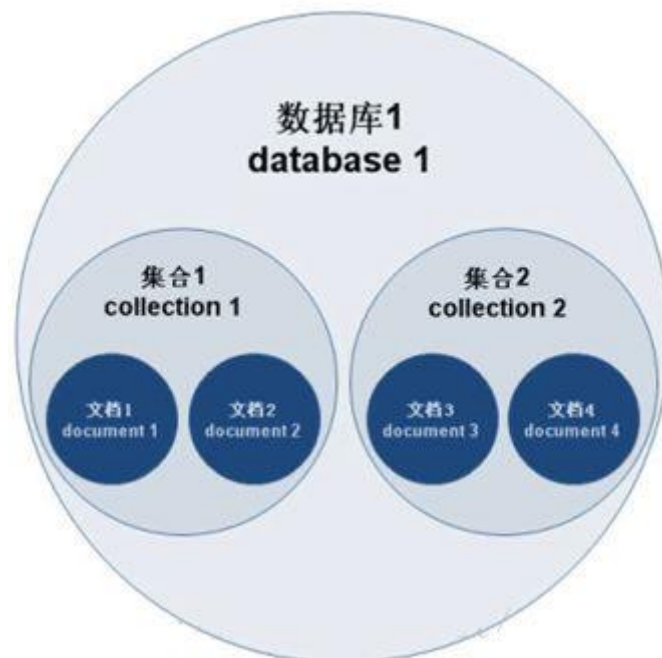
MongoDB 的整体架构跟其他的关系型的数据库很类似，都是有一些关键的模块组成，作为一个简单的对比， 如下是一个经典的关系型数据库的结构图：



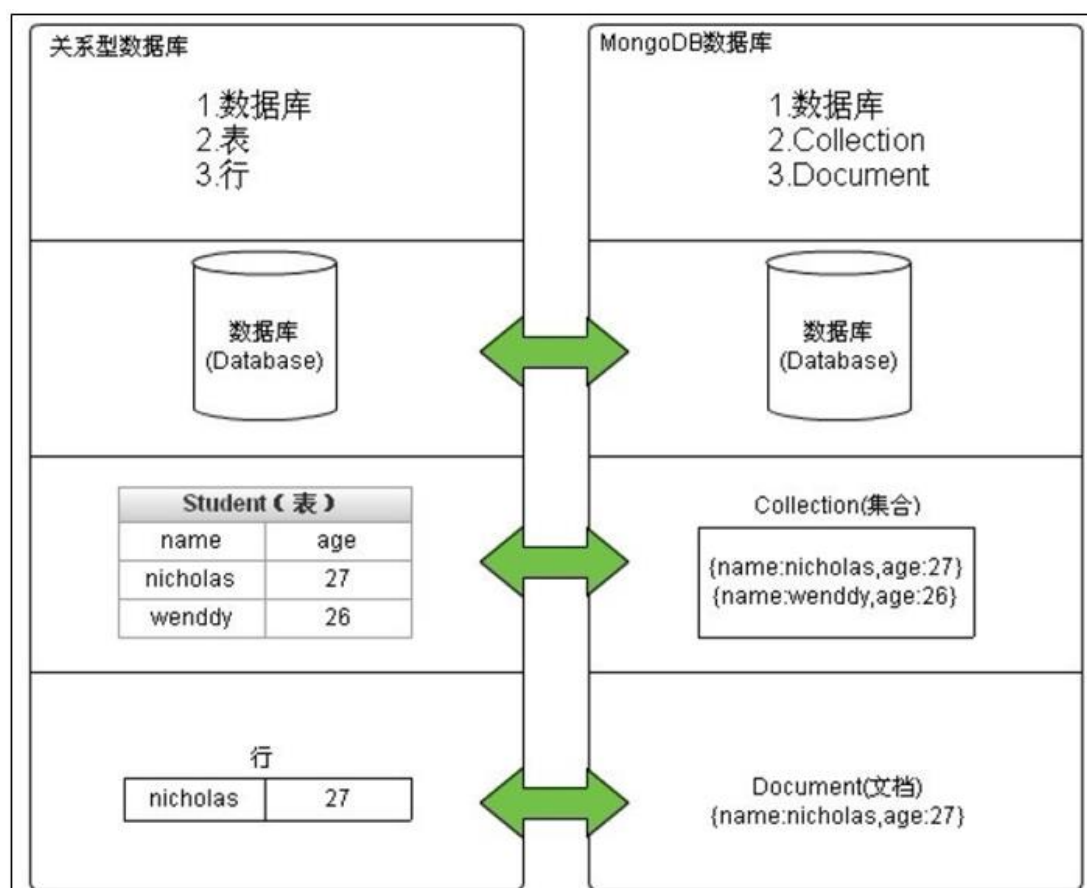
如下是 MongoDB 的一个类似的结构图：



MongoDB 的逻辑结构是一种层次结构。主要由：文档(document)、集合(collection)、数据库(database)这三部分组成的。逻辑结构是面向用户的，用户使用 MongoDB 开发应用程序使用的就是逻辑结构。文档(document)、集合(collection)、数据库(database)的层次结构如下图：



数据库中的对应关系，及存储形式的说明：



- 一个 mongod 实例中允许创建多个数据库。
- 一个数据库中允许创建多个集合（集合相当于关系型数据库的表）。
- 一个集合则是由若干个文档构成（文档相当于关系型数据库的行，是 MongoDB 中数据的基本单元）。

MongoDB 与 SQL 的结构对比如下：

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the _id field.
aggregation (e.g. group by)	aggregation pipeline See the SQL to Aggregation Mapping Chart.

在 MongoDB 的源代码中，有如下结构：

```
root@wiki:[/opt/mongodb/mongodb-linux-x86_64-3.6.2/bin]11
total 244188
-rwxr-xr-x 1 mongodb mongodb 5487061 Jan 11 00:07 bsondump
-rwxr-xr-x 1 mongodb mongodb 5792 Jan 11 00:22 install_compass
-rwxr-xr-x 1 mongodb mongodb 33070144 Jan 11 00:22 mongo
-rwxr-xr-x 1 mongodb mongodb 58404032 Jan 11 00:22 mongod
-rwxr-xr-x 1 mongodb mongodb 8667585 Jan 11 00:07 mongodump
-rwxr-xr-x 1 mongodb mongodb 6620731 Jan 11 00:07 mongoexport
-rwxr-xr-x 1 mongodb mongodb 6471255 Jan 11 00:07 mongofiles
-rwxr-xr-x 1 mongodb mongodb 6771267 Jan 11 00:07 mongoimport
-rwxr-xr-x 1 mongodb mongodb 57928912 Jan 11 00:22 mongoperf
-rwxr-xr-x 1 mongodb mongodb 9780339 Jan 11 00:07 mongoreplay
-rwxr-xr-x 1 mongodb mongodb 10082759 Jan 11 00:07 mongorestore
-rwxr-xr-x 1 mongodb mongodb 33456944 Jan 11 00:22 mongos
-rwxr-xr-x 1 mongodb mongodb 6833184 Jan 11 00:07 mongostat
-rwxr-xr-x 1 mongodb mongodb 6441493 Jan 11 00:07 mongotop
```

其含义分别为：

- 数据库服务（mongod）
- 分片集群部署中，数据和查询的路由服务（mongos）
- shell 客户端（mongo）
- 导入导出工具（mongoimport / mongoexport）
- 备份恢复工具（mongodump / mongorestore）
- 拉取并重放 oplog 的工具（mongoexportlog）
- 监控工具（mongostat、mongotop、mongosniff）
- GridFS 的命令行操作工具（mongofiles）
- 性能测试工具（mongoperf，暂时只能测 I/O）
- 查看 bson 文件的工具（bsondump）

mongod 在不同的部署方案中（单机部署，副本集部署，分片集群部署），通过不同的配置，可以扮演多种不同的角色：在单机部署中扮演 数据库服务器（提供所有读写功能）在副本集部署中，通过配置，可以部署为 **primary** 节点（主服务器，负责写数据，也可以提供查询）、**secondary** 节点（从服务器，它从主节点复制数据，也可以提供查询）、以及 **arbiter** 节点（仲裁节点，不保存数据，主要用于参与选举投票）。在分片集群中，除了在每个分片中扮演上述角色外，还扮演着配置服务器的角色（存储有分片集群的所有元数据信息，mongos 的数据路由分发等都要依赖于它）。在一台服务器上，可以启动多个 mongod 服务。但在实际生产部署中，通常还是建议一台服务器部署一个 mongod 实例，这样不仅减少资源竞争，而且服务器故障也不会同时影响到多个服务。

1.3 MongoDB 源码分析

MongoDB 的源码根目录如下所示：

48,868 commits

28 branches

763 releases

393 contributors

View license

C++ 72.5%

JavaScript 20.1%

Python 4.8%

Go 2.2%

Shell 0.1%

C 0.1%

Other 0.2%

Branch: master

New pull request

Create new file

Upload files

Find file

Clone or download

zakhark and evergreen

SERVER-42846 add mongodap and mongoreplay man pages to rpm specs

Latest commit 1c4b963 11 hours ago

buildscripts	SERVER-42748 Support using stored procedures (system.js) in map/reduce	13 hours ago
debian	SERVER-42846 Update manpages	16 hours ago
distsrc	SERVER-26906 Add constexpr function to convert Durations	5 months ago
docs	SERVER-41757 Fix issues with the building documentation	4 months ago
etc	SERVER-43797 Add new burn_in_tests_multiversion_gen task	18 hours ago
jstests	Revert "SERVER-44042 Enable reconstruct_prepared_transactions_initial...	13 hours ago
pytests	SERVER-42615 Run chkdisk command on Windows after each powercycle loop.	3 months ago
rpm	SERVER-42846 add mongodap and mongoreplay man pages to rpm specs	11 hours ago
site_scons	SERVER-43730 Small build system speed improvements	16 days ago
src	SERVER-42748 Support using stored procedures (system.js) in map/reduce	13 hours ago

.clang-format	SERVER-41771 Use `clang-format-7.0.1` in our clang-format helper script	3 months ago
.eslintignore	SERVER-31390 Use a templating language to generate error_codes.{h,cpp...	2 years ago
.eslintrc.yml	SERVER-23728 Enable the no-unused-expressions ESLint rule	3 years ago
.gdbinit	SERVER-28668 Add mongo_printers.py to .gdbinit	3 years ago
.gitattributes	SERVER-29877 Mount /data on EBS volume in AWS EC2 instance	2 years ago
.gitignore	SERVER-43160 Updating rpm, deb, signing, and MSI packaging for mongok...	last month
.lldbinit	SERVER-41168 Add lldb comment to .lldbinit	5 months ago
.pydocstyle	SERVER-40559 Python linters do not run after upgrade to Python 3	7 months ago
.pylintrc	SERVER-40559 Python linters do not run after upgrade to Python 3	7 months ago
.style.yapf	SERVER-23312 Format Python files with yapf	2 years ago
APACHE-2.0.txt	Add the Apache 2 license, add licensing info to README. MINOR	10 years ago
CONTRIBUTING.rst	SERVER-29767 Update CONTRIBUTING.rst to point to github wiki	2 years ago
LICENSE-Community.txt	SERVER-37651 Fix typo in license files	last year
README	SERVER-41833 Update README license text	4 months ago
README.third_party.md	SERVER-44082 Update third party inclusions file	9 days ago
SConstruct	SERVER-43936 Implement simpler, quicker python based C++ linter	5 days ago
mypy.ini	SERVER-32295 Support Python 3	7 months ago


```

collection-1-1695719484656079182.wt index-11-66333636898585760.wt index-9-5321798130068747782.wt
root@wiki: [/opt/mongodb/data]ll
total 327180
-rw----- 1 root root 4096 Jun 6 18:49 collection-0--1695719484656079182.wt
-rw----- 1 mongodb mongodb 32768 Jun 6 18:49 collection-0-5321798130068747782.wt
-rw----- 1 mongodb mongodb 4096 Jun 6 18:49 collection-0-6479534320575166107.wt
-rw----- 1 root root 4096 Jun 6 18:49 collection-0--66333636898585760.wt
-rw----- 1 root root 32768 Jun 6 18:49 collection-0-9218527314995862213.wt
-rw----- 1 root root 258048 Jun 6 18:49 collection-10--66333636898585760.wt
-rw----- 1 root root 4096 Jun 6 18:49 collection-1--1695719484656079182.wt
-rw----- 1 mongodb mongodb 36864 Jun 6 18:50 collection-2-5321798130068747782.wt
-rw----- 1 root root 4096 Jun 6 18:49 collection-2--66333636898585760.wt
-rw----- 1 mongodb mongodb 231907328 Jun 6 18:49 collection-3-6479534320575166107.wt
-rw----- 1 root root 16384 Jun 6 18:49 collection-4--66333636898585760.wt
-rw----- 1 root root 4096 Jun 6 18:49 collection-8--66333636898585760.wt
drwx----- 2 mongodb mongodb 4096 Jun 7 11:42 diagnostic.data
-rw----- 1 root root 98304 Apr 20 16:42 index-11--66333636898585760.wt
-rw----- 1 mongodb mongodb 32768 Jun 6 18:49 index-1-5321798130068747782.wt
-rw----- 1 mongodb mongodb 4096 Apr 19 11:37 index-1-6479534320575166107.wt
-rw----- 1 root root 4096 Apr 20 16:42 index-1--66333636898585760.wt
-rw----- 1 root root 32768 Jun 6 18:48 index-1-9218527314995862213.wt
-rw----- 1 mongodb mongodb 4096 Jun 6 18:50 index-2-6479534320575166107.wt
-rw----- 1 root root 32768 Jun 6 18:50 index-2-9218527314995862213.wt
-rw----- 1 mongodb mongodb 36864 Jun 6 18:50 index-3-5321798130068747782.wt
-rw----- 1 root root 4096 Apr 20 16:42 index-3--66333636898585760.wt
-rw----- 1 mongodb mongodb 102240256 Apr 19 11:37 index-4-6479534320575166107.wt
-rw----- 1 root root 16384 Apr 20 16:42 index-5--66333636898585760.wt
-rw----- 1 root root 4096 Apr 20 16:42 index-9--66333636898585760.wt
drwx----- 2 mongodb mongodb 107 Jun 6 18:50 journal
-rw----- 1 mongodb mongodb 36864 Jun 6 18:49 _mdb_catalog.wt
-rw----- 1 mongodb mongodb 6 Jun 6 18:49 mongod.lock
-rw----- 1 mongodb mongodb 36864 Jun 6 18:50 sizeStorer.wt
-rw----- 1 mongodb mongodb 114 Apr 17 10:51 storage.bson
-rw----- 1 mongodb mongodb 45 Apr 17 10:51 WiredTiger
-rw----- 1 root root 4096 Jun 6 18:49 WiredTigerLAS.wt
-rw----- 1 mongodb mongodb 21 Apr 17 10:51 WiredTiger.lock
-rw----- 1 root root 1049 Jun 6 18:50 WiredTiger.turtle
-rw----- 1 mongodb mongodb 102400 Jun 6 18:50 WiredTiger.wt
root@wiki: [/opt/mongodb/data]

```

MongoDB 的数据库文件主要有 3 种：journal 日志文件、namespace 表名文件、data 数据及索引文件。MongoDB 的日志文件只是用来在系统出现宕机时候恢复尚未来得及同步到硬盘的内存数据。日志文件会存放在一个分开的目录下面。启动时候 MongoDB 会自动预先创建 3 个每个为 1G 的日志文件（初始为空）。用来存储整个数据库的集合以及索引的名字。这个文件不大，默认 16M，可以存储 24000 个集合或者索引名以及那些集合和索引在数据文件中得具体位置。通过这个文件 MongoDB 可以知道从哪里去开始寻找或插入集合的数据或者索引数据。这个值可以通过参数调整至 2G。MongoDB 的数据以及索引都存放在一个或者多个 MongoDB 数据文件里。第一个数据文件会以“数据库名.0”命名，如 my-db.0。这个文件默认大小是 64M，在接近用完这个 64M 之前，MongoDB 会提前生成下一个数据文件如 my-db.1。数据文件的大小会 2 倍递增。第二个数据文件的大小为 128M，第三个为 256M。一直到了 2G 以后就会停止，一直按这个 2G 这个大小增加新的文件。

在每一个数据文件内，MongoDB 把所存储的 BSON 文档的数据和 B 树索引组织到逻辑容器“Extent”里面。如下图所示（my-db.1 和 my-db.2 是数据库的两个数据文件）：一个文件可以有多个 Extent；每一个 Extent 只会包含一个集合的数据或者索引；同一个集合的数据或索引可以分布在多个 Extent 内。这几个 Extent 也可以分步于多个文件内；同一个 Extent 不会又有数据又有索引。在每个 Extent 里面存放有多个“Record”，每一个记录里包含一个记录头以及 MongoDB 的 BSON 文档，以及一些额外的 padding 空间。Padding 是 MongoDB

在插入记录时额外分配一些未用空间，这样将来文档变大的时候不至于需要把文档迁移到别处。记录头以整个记录的大小开始，包括该记录自己的位置以及前一个记录和后一个记录的位置。可以想象成一个 Double Linked List。

1.4 MongoDB 的一个查询过程

由于查询部分流程比较长，将分成 mongo 端的请求，mongod 端的数据库的加载，mongod query 的选取，mongod 文档的匹配与数据的响应几部分来分析。首先进入 mongo 的查询请求部分。mongo 的查询请求部分归纳起来很简单就是将请求分装成一个 Message 结构，然后将其发送到服务端,等待服务端的相应数据，取得数据最后显示结果。下面来看具体流程分析：

首先来到 mongo/shell/dbshell.cpp:

```
if ( ! wascmd ) {
    try {
        if ( scope->exec( code.c_str() , "(shell)" , false , true , false ) )//执行相应的javascript代码
            scope->exec( "shellPrintHelper( __lastres__ );" , "(shell2)" , true , true , false );
    }
    catch ( std::exception& e ) {
        cout << "error:" << e.what() << endl;
    }
}
```

下面进入 javascript 代码，其在 mongo/shell/collection.js，这里因为只设置了 query，所以其它选项都是空的，this.getQueryOptions()目前只有一个 SlaveOK 的 option，在 replset 模式下是不能查询 secondary 服务器的，需要调用 rs.SlaveOK()之后才能对 secondary 进行查询，其执行 SlaveOK 后每次查询时都会添加一个

```
DBCollection.prototype.find = function (query, fields, limit, skip, batchSize, options) {
    return new DBQuery( this._mongo , this._db , this ,
                        this._fullName , this._messageObject( query ) , fields , limit , skip , batchSize , options ||
                        this.getQueryOptions() );
}
```

new DBQuery 对象的创建发生在：

```

JSBool dbquery_constructor( JSContext *cx, JSObject *obj, uintN argc, jsval *argv, jsval *rval ) {
    try {
        smuassert( cx , "DDQuery needs at least 4 args" , argc >= 4 );
        Convertor c(cx);
        c.setProperty( obj , "_mongo" , argv[0] );
        c.setProperty( obj , "_db" , argv[1] );
        c.setProperty( obj , "_collection" , argv[2] );
        c.setProperty( obj , "_ns" , argv[3] );

        if ( argc > 4 && JSVAL_IS_OBJECT( argv[4] ) )
            c.setProperty( obj , "_query" , argv[4] );
        else {
            JSObject * temp = JS_NewObject( cx , 0 , 0 , 0 );
            CHECKNEWOBJECT( temp, cx, "dbquery_constructor" );
            c.setProperty( obj , "_query" , OBJECT_TO_JSVAL( temp ) );
        }

        if ( argc > 5 && JSVAL_IS_OBJECT( argv[5] ) )
            c.setProperty( obj , "_fields" , argv[5] );
        else
            c.setProperty( obj , "_fields" , JSVAL_NULL );

        if ( argc > 6 && JSVAL_IS_NUMBER( argv[6] ) )
            c.setProperty( obj , "_limit" , argv[6] );
        else
            c.setProperty( obj , "_limit" , JSVAL_ZERO );

        if ( argc > 7 && JSVAL_IS_NUMBER( argv[7] ) )
            c.setProperty( obj , "_skip" , argv[7] );
        else
            c.setProperty( obj , "_skip" , JSVAL_ZERO );

        if ( argc > 8 && JSVAL_IS_NUMBER( argv[8] ) )
            c.setProperty( obj , "_batchSize" , argv[8] );
        else
            c.setProperty( obj , "_batchSize" , JSVAL_ZERO );

        if ( argc > 9 && JSVAL_IS_NUMBER( argv[9] ) )
            c.setProperty( obj , "_options" , argv[9] );
        else
            c.setProperty( obj , "_options" , JSVAL_ZERO );

        c.setProperty( obj , "_cursor" , JSVAL_NULL );
        c.setProperty( obj , "_numReturned" , JSVAL_ZERO );
        c.setProperty( obj , "_special" , JSVAL_FALSE );
    }
    catch ( const AssertionException& e ) {
        if ( ! JS_IsExceptionPending( cx ) ) {
            JS_ReportError( cx, e.what() );
        }
        return JS_FALSE;
    }
    catch ( const std::exception& e ) {
        log() << "unhandled exception: " << e.what() << ", throwing Fatal Assertion" << endl;
        fassertFailed( 16323 );
    }
    return JS_TRUE;
}

```

可以看到上面只有 DBQuery 对象的构建动作,并没有真正的查询请求,回到:

```

try {
    if ( scope->exec( code.c_str(), "(shell)", false, true, false ) )//执行相应的javascript代码
        scope->exec( "shellPrintHelper( __lastres__ );", "(shell2)", true, true, false );
}

```

继续分析 shellPrintHelper 函数，这里 __lastres__ 是什么，搜索整个 source insight 工程发现 mongo/scripting/engine_spidermonkey.cpp 中：

```

bool exec( const StringData& code, const string& name = "(anon)", bool printResult = false, bool reportError = true, bool
assertOnError = true, int timeoutMs = 0 ) {
    JSBool worked = JS_EvaluateScript( _context,
        _global,
        code.data(),
        code.size(),
        name.c_str(),
        1,
        &ret );

    if ( worked )
        _converter->setProperty( _global, "__lastres__", ret );
}

```

__lastres__ 就是上一条执行语句的结果也就是这里的 DBQuery 对象。继续分析 shellPrintHelper 函数(mongo/util/util.js)：

```

shellPrintHelper = function (x) {
    if (typeof (x) == "undefined") {
        // Make sure that we have a db var before we use it
        // TODO: This implicit calling of GLE can cause subtle, hard to track issues - remove?
        if ( __callLastError && typeof ( db ) != "undefined" && db.getMongo ) {
            __callLastError = false;
            // explicit w:1 so that replset getLastErrorDefaults aren't used here which would be bad.
            var err = db.getLastError(1);
            if (err != null) {
                print(err);
            }
        }
        return;
    }

    if (x == __magicNoPrint)
        return;

    if (x == null) {
        print("null");
        return;
    }

    if (typeof x != "object")
        return print(x);

    var p = x.shellPrint; //我们这里是DBQuery对象，所以执行到这里，来到了DBQuery.shellPrint函数
    if (typeof p == "function")
        return x.shellPrint();

    var p = x.tojson;
    if (typeof p == "function")
        print(x.tojson());
    else
        print(tojson(x));
}

```

通过上述代码可知：

```

DBQuery.prototype.shellPrint = function(){//(mongo/util/query.js)
    try {
        var start = new Date().getTime();
        var n = 0; //还有查询结果并且输出数目小于shellBatchSize,循环打印结果
        while ( this.hasNext() && n < DBQuery.shellBatchSize ){//这里shellBatchSize定义为20
            var s = this._prettyShell ? tojson( this.next() ) : tojson( this.next() , "" , true );
            print( s );//调用native函数native_print打印结果
            n++;
        }
        if (typeof _verboseShell !== 'undefined' && _verboseShell) {
            var time = new Date().getTime() - start;
            print("Fetched " + n + " record(s) in " + time + "ms");
        }
        if ( this.hasNext() ){
            print( "Type \"it\" for more" );
            __it__ = this;
        }
        else {
            __it__ = null;
        }
    }
    catch ( e ){
        print( e );
    }
}

```

继续看 hasNext 函数和 next 函数:

```

DBQuery.prototype.hasNext = function(){
    this._exec();

    if ( this._limit > 0 && this._cursorSeen >= this._limit )//超过了限制返回false, 将不会再输出结果
    |   return false;
    var o = this._cursor.hasNext();
    return o;
}

DBQuery.prototype.next = function(){
    this._exec();

    var o = this._cursor.hasNext();
    if ( o )
    |   this._cursorSeen++;
    else
    |   throw "error hasNext: " + o;

    var ret = this._cursor.next();
    if ( ret.$err && this._numReturned == 0 && ! this.hasNext() )
    |   throw "error: " + tojson( ret );

    this._numReturned++;
    return ret;
}

```

继续前进到 _exec 函数:

```

DBQuery.prototype._exec = function(){//到这里终于到了this._mongo.find
    if ( ! this._cursor ){
        assert.eq( 0 , this._numReturned );
        this._cursor = this._mongo.find( this._ns , this._query , this._fields , this._limit , this._skip , this._batchSize ,
        this._options );
        this._cursorSeen = 0;
    }
    return this._cursor;
}

```

到这里来到了 this._mongo.find,这里_mongo 是一个 Mongo 对象, 在上一篇文章中了解到 find 函数是本地函数 mongo_find.继续分析 mongo_find (mongo/scripting/sm_db.cpp), 这里删除了部分错误处理代码:

```

JSBool mongo_find(JSContext *cx, JSObject *obj, uintN argc, jsval *argv, jsval *rval) {
    shared_ptr< DBClientWithCommands > * connHolder = (shared_ptr< DBClientWithCommands >*)JS_GetPrivate( cx , obj );
    smuassert( cx , "no connection!", connHolder && connHolder->get() );
    DBClientWithCommands *conn = connHolder->get();
    Converter c( cx );
    string ns = c.toString( argv[0] );
    BSONObj q = c.toObject( argv[1] );
    BSONObj f = c.toObject( argv[2] );

    int nToReturn = (int) c.toNumber( argv[3] );
    int nToSkip = (int) c.toNumber( argv[4] );
    int batchSize = (int) c.toNumber( argv[5] );
    int options = (int)c.toNumber( argv[6] );//上面一篇文章我们分析到这里的conn其实是由ConnectionString::connect函数返回的，其返回的
    对象指针可能是:DBClientConnection对应Master,也就是只设置了一个地址，DBClientReplicaSet对应pair或者set模式,SyncClusterConnection对
    应sync模式。继续分析流程我们选择最简单的Master模式,只有一个地址的服务端
    auto_ptr<DBClientCursor> cursor = conn->query( ns , q , nToReturn , nToSkip , f.nFields() ? &f : 0 , options , batchSize );

    if ( ! cursor.get() ) {
        log() << "query failed : " << ns << " " << q << " to: " << conn->toString() << endl;
        JS_ReportError( cx , "error doing query: failed" );
        return JS_FALSE;
    }
    JSObject * mycursor = JS_NewObject( cx , &internal_cursor_class , 0 , 0 );
    CHECKNEWOBJECT( mycursor, cx, "internal_cursor_class" );
    verify( JS_SetPrivate( cx , mycursor , new CursorHolder( cursor, *connHolder ) ) );
    *rval = OBJECT_TO_JSVAL( mycursor );
    return JS_TRUE;
}

```

继续前进来到 DBClientConnection::query 函数，该函数只是简单调用了

DBClientBase::query 函数：

```

auto_ptr<DBClientCursor> DBClientBase::query(const string &ns, Query query, int nToReturn,
    int nToSkip, const BSONObj *fieldsToReturn, int queryOptions , int batchSize ) {
    auto_ptr<DBClientCursor> c( new DBClientCursor( this, //根据传入的参数创建一个DBClientCursor对象
        ns, query.obj, nToReturn, nToSkip,
        fieldsToReturn, queryOptions , batchSize ) );
    if ( c->init() )//创建Message并向服务端发送查询请求
        return c;
    return auto_ptr< DBClientCursor >( 0 );
}

```

_assembleInit 是创建一个 message 结构，若是第一次请求那么请求操作为 dbQuery，若不是则请求操作为 dbGetmore。

```

bool DBClientCursor::init() {
    Message toSend;
    _assembleInit( toSend );//构建将要发送的查询请求这是一个Message,具体来说Message负责发送数据,具体的数据是在
    MsgData中
    verify( _client );
    if ( !_client->call( toSend, *batch.m, false, &_originalHost ) ) { //实际的发送数据,同时这里发送了数据后会调用
    recv接收数据
        // log msg temp? //接收的数据同样是MsgData, 同样由Message来
        管理
        log() << "DBClientCursor::init call() failed" << endl;
        return false;
    }
    if ( batch.m->empty() ) {
        // log msg temp?
        log() << "DBClientCursor::init message from call() was empty" << endl;
        return false;
    }
    dataReceived();//根据上面的batch.m收到的数据得出查询是否成功成功则设置cursorId,下一次请求时operation就变动为
    dbGetmore了。
    return true; //查询错误则抛出异常
}

```

继续看 MsgData 的具体结构，回到 javascript 部分的打印请求：

```

DBQuery.prototype.hasNext = function(){
    this._exec();

    if ( this._limit > 0 && this._cursorSeen >= this._limit )
        return false;
    var o = this._cursor.hasNext();//这里的cursor对应于上面的C++对象cursor,其hasNext函数对应的native函数为
    internal_cursor_hasNext,
    return o; //这是在mongo启动时初始化javascript环境时绑定的
}

```

看到了 `internal_cursor_hasNext`:

```
JSBool internal_cursor_hasNext(JSContext *cx, JSObject *obj, uintN argc, jsval *argv, jsval *rval) {
    try {
        DBClientCursor *cursor = getCursor( cx, obj );
        *rval = cursor->more() ? JSVAL_TRUE : JSVAL_FALSE; //这里返回的就是是否还有数据,如果本地没有查询数据了,那么
        其会再构建一个                                     //dbGetmore的请求向服务器请求更多数据,还是没有则返回
    }
    false,表示没有数据了
    catch ( const AssertionException& e ) {
        if ( ! JS_IsExceptionPending( cx ) ) {
            JS_ReportError( cx, e.what() );
        }
        return JS_FALSE;
    }
    catch ( const std::exception& e ) {
        log() << "unhandled exception: " << e.what() << ", throwing Fatal Assertion" << endl;
        fassertFailed( 16290 );
    }
    return JS_TRUE;
}
```

最后 javascript 部分打印数据:

```
DBQuery.prototype.shellPrint = function(){
    try {
        var start = new Date().getTime();
        var n = 0;
        while ( this.hasNext() && n < DBQuery.shellBatchSize ) { //前面分析这里hasNext对应C++函数
            internal_cursor_hasNext,next对应
            var s = this._prettyShell ? tojson( this.next() ) : tojson( this.next() , "" , true ); //
            internal_cursor_next,怎么得到的数据不再分析
            print( s ); //调用native_print打印结果
            n++;
        }
    }
}
```

以上是 mongodb 自带的 mongo 客户端发起的查询请求以及结果的打印过程,令人感到麻烦的就是代码的执行流程在 javascript 和 C++中来回的切换,需要注意的就是对于 mongo 的查询请求的格式 `MsgData`。

1.5 MongoDB 优缺点评价

这里主要是对比 MySQL 来说明。

优点方面:

1. 不存在 sql 注入: MySQL 的是 sql 注入是一个很严重的缺点,虽然可以使用参数绑定和预处理以及特殊字符转义来处理。但是 MongoDB 根本不存在这个问题。不过 xss 攻击还是需要防范的。
2. 不需要提前创建表: 在 MySQL 中如果想要写入一条数据的话必须要先创建好一张表然后才能写入数据,比如:要在 user 表里写入 id=1,username='aaa',sex='女',age='20' 这条数据,那你就必须在 MySQL 数据库上提前建好一张 user 表,并且至少必须有 id,username,sex,age 这几个字段才能写入成功。但是 MongoDB 可以直接写入数据,不需要提前创建表,例子:
`db.user.insert({"id":1,"username":"aaa","sex":"女","age":20})`
3. 可以任意添加或减少字段: 用上一个例子,在 MySQL 中假设我要在上一条数据上再加一个身高 height 字段,那么我就必须再去 user 表上再去添加一个 height 字段才能写入 height 数据。但是在 MongoDB 中你可以直接写就行,而且假设你的第一条数据是 id,username,sex,age,那么你的第二条数据大可

以写成 id,username,age,height。

4. 字段数据格式自由：在 MySQL 中，如果 id 字段是数字的话你写一个字符串进去是会报错的，但是 MongoDB 不会。
5. 可以处理 json 结构：在 MongoDB 可以存储一个 json 对象，比如 字段 a 的值为{"a":11,"b":12,"c":"abc","d":[1,2,3]},你可以直接去读取或设置 a 字段的 b 值 a.b，读取 a 字段 d 数组的第二个值：a.d.1,可以去删除 a 字段的 a 数据 \$unset:{"a.a":1},就会变成：{"b":12,"c":"abc","d":[1,2,3]}
6. 可以使用 upsert 操作，即修改的数据不存在时直接插入。
({where},{ \$set:{data}},{"upsert":true})
7. 充分利用了计算机内存，所以查询和插入效率要远大于 MySQL。我自己测试（400w 随机数据）的时候只有在没有索引的情况下 MongoDB 的查询效率要远大于 MySQL，插入效率和 MySQL 差不多都是 8w 条左右 1 分钟。在有索引的时候 MySQL 的查询要速度要高于 MongoDB。

缺点方面：

1. MongoDB 是个 NoSQL 数据，所以关系能力薄弱，不能像 MySQL 一样使用 join, union 来进行联合查找，只能通过结合一些特殊语法来达到类似的结果。
2. 事务能力薄弱，虽然 MongoDB 里事务，但是好像只能针对单条语句（查了好多但是有些看不懂），不能像 MySQL 一样利用事务执行多条语句，可以根据情况来选着全部提交执行或者全部取消回滚。

二、OceanBase 数据库的分析

2.1 系统概述

在过去半个世纪左右的时间里，数据库系统从无到有，经历了层次数据库、网状数据库和关系数据库等几个发展阶段。时至今日，主要数据库系统基本都是关系数据库，并广泛应用于银行、信用卡交易、商品销售、航空与铁路运输、电信、电力系统、教育、医疗与健康、电子商务等领域，成为了信息社会最关键的基础设施之一。关系数据库系统也成为了最稳定可靠、最核心的系统之一。传统关系数据库本质上是单机系统，通常采用昂贵的高端服务器和高端存储，难以应对互联网应用的高可扩展、高性能、高可用和低成本挑战。阿里巴巴 OceanBase 团队研制了 OceanBase 开源 分布式无共享关系数据库，以“OceanBase 数据库 + 主流 PC 服务器”取代“商业数据库 + 高端服务器 + 高端存储”模式，很好地满足了互联网对关系数据库的需求。OceanBase 已经用于淘宝、天猫和支付宝的多个生产系统，极大地降低了软件和设备成本，良好的伸缩性和自动的故障恢复不仅很好地支撑了业务，而且大大地降低了运行维护的人力成本。

2.2 OceanBase 的国内主要市场情况

长期以来，数据库系统（DBMS）的市场规模在软件行业里一直是最大的。在 2017 年的总市场规模达到 368 亿美元，相对于 2016 年有超过 8% 的增长。然而在最近几年，传统的 5 家商业数据库厂商 Oracle、微软、IBM、SAP、Teradata 总市场份额却在持续下滑，从 2011 年的 91% 下降到 2016 年的 86.9%。与传统商业数据库的市场占有率持续下降相对应的是，AWS 透露在过去的两年中，有超过 64,000 个数据库从传统商业数据库（Oracle 等）迁移到 AWS 的云数据库服务。而这个数字在 2017 年 4 月份的 AWS 峰会上还是 23,000。虽然 AWS 没有透露迁移的数据库规模有多大，属于核心业务还是边缘业务，但是从数量和增速上看还是十分惊人的。

OceanBase 是由蚂蚁金服、阿里巴巴完全自主研发的金融级分布式关系数据库，始创于 2010 年。OceanBase 具有数据强一致、高可用、高性能、在线扩展、高度兼容 SQL 标准和主流关系数据库、低成本等特点。OceanBase 至今已成功应用于支付宝全部核心业务：交易、支付、会员、账务等系统以及阿里巴巴淘宝（天猫）收藏夹、P4P 广告报表等业务。除在蚂蚁金服和阿里巴巴业务系统中获广泛应用外，从 2017 年开始，OceanBase 开始服务外部客户，客户包括南京银行、浙商银行、印度 Paytm、人保健康险。经过一年多的积累，我们决定将 OceanBase 开向下一站的时候，选择的是先满足金融行业，尤其是专有云的场景需求。

在市场化方面，OceanBase 将会以蚂蚁金服为始发站，开向下一站金融行业。首先，金融业更关注整个系统的可靠性、扩展性，OceanBase 最重要的几个核心特性与金融行业的需求高度契合。其次，目前 OceanBase 具备的应用场景优势，是在蚂蚁内部积累的核心金融业务场景的经验，映射到金融行业显然是最容易被客户接受的，目前已经实施的几个银行客户案例也证明了这一逻辑。第三，

OceanBase 会在合适的时间选择上公有云。相对来说公有云的客户众多，对产品的广度要求远远高于深度，这并非我们当前最擅长的领域，团队需要继续成长，如果现在直接上公有云，会拖慢 OceanBase 团队在核心技术创新方面的脚步。对于蚂蚁金服来说，不是只有 OceanBase 在考虑怎么去做商业化的问题，而是整个蚂蚁金服都在加速将内部技术创新赋能给金融行业。在这个大背景下，OceanBase 跟着集团军一起作战显然是更好的选择。从客户角度看，通常也不会只关注一个单独的数据库产品，而不去考虑整体架构、研发效能、服务治理，不去考虑金融场景、产品营运、甚至是组织文化的创新。这时蚂蚁金服的整体解决方案价值就格外重要，整个金融业正在逐步形成一个共识——金融科技是未来，分布式架构是未来。一眨眼，OceanBase 也到了七年之痒的时间点，需要一些变化。对蚂蚁金服来说，OceanBase 的商业化进程非常重要，需要通过更好地理解外部客户的业务场景、具体需求，帮助客户的新业务能稳定运行，让更多的业务能够跑在 OceanBase 上，逐步建立起品牌和声誉。让蚂蚁金服的金融科技输出战略因为有了 OceanBase 而不同。当然，一切的前提是把产品和生态做好，才有能力越过 OceanBase 的下一个山丘。

2.3 性能对比分析

2019 年 10 月 2 日，数据库领域最权威的国际机构国际事务处理性能委员会（TPC，Transaction Processing Performance Council）在官网发表了最新的 TPC-C 基准测试结果。蚂蚁金服自主研发的金融级分布式关系数据库 OceanBase 以两倍于 Oracle（甲骨文）的成绩，打破数据库基准性能测试的世界记录，成为全球数据库演进史的重要里程碑。

官方提供的测试数据如下：

Sponsor	System	Performance (TpmC)	Price/TpmC	System Availability	Date Submitted	DB Software Name
	Alibaba Cloud Elastic Compute Service Cluster	60,880,800	6.25 CNY	10/2/2019	10/1/2019	OceanBase v2.2 Enterprise Edition with Partitioning, Horizontal Scalab
	SPARC SuperCluster with T3-4 Servers	30,249,688	1.01 USD	6/1/2011	12/2/2010	Oracle Database 11g R2 Enterprise Edition w/RAC w/Partitioning
	IBM Power 780 Server Model 9179-MHB	10,366,254	1.38 USD	10/13/2010	8/17/2010	IBM DB2 9.7
	SPARC T5-8 Server	8,552,523	0.55 USD	9/25/2013	3/26/2013	Oracle 11g Release 2 Enterprise Edition with Oracle Partitioning
	Sun SPARC Enterprise T5440 Server Cluster	7,646,486	2.36 USD	3/19/2010	11/4/2009	Oracle Database 11g Enterprise Edition w/RAC w/Partitioning
	IBM Power 595 Server Model 9119-FHA	6,085,166	2.81 USD	12/10/2008	6/10/2008	IBM DB2 9.5
	Bull Escala PL6460R	6,085,166	2.81 USD	12/15/2008	6/15/2008	IBM DB2 9.5
	Sun Server X2-8	5,055,888	0.89 USD	7/10/2012	6/20/2012	Oracle Database 11g R2 Enterprise Edition w/Partitioning
	Sun Fire X4800 M2 Server	4,803,718	0.98 USD	6/26/2012	1/18/2012	Oracle Database 11g R2 Enterprise Edition
	HP Integrity Superdome-Itanium2/1.6GHz/24MB IL3	4,092,799	2.93 USD	8/6/2007	2/27/2007	Oracle Database 10g R2 Enterprise Edition w/Partitioning

可以看到，阿里的 OceanBase 的性能排在第一位，每分钟 6088 万笔交易，后面依次是 Oracle 11g 的 3000 万，IBM DB2 9.7 的 1000 万，看这个数据确实还是非常不错的，但同时也应当看到另一个方面。如下图所示：



Alibaba Cloud Elastic Compute Service Cluster

Reference URL: <http://www.tpc.org/1799>

Benchmark Stats

Result ID:	119100101
Status: 	Result In Review
Report Date:	10/01/19
TPC-C Rev:	5.11.0

System Information

Total System Cost:	380,452,842 CNY
Performance:	60,880,800 tpmC
Price/Performance:	6.25 CNY per tpmC
TPC-Energy Metric:	Not reported
Availability Date:	10/02/19
Active Expiration Date:	10/01/22
Database Manager:	OceanBase v2.2 Enterprise Edition with Partitioning, Horizontal Scalab
Operating System:	Aliyun Linux 2
Transaction Monitor:	Nginx 1.15.8

Server Specific Information

CPU Type:	Intel Xeon Platinum 8163 2.50GHz
Total # of Processors:	420
Total # of Cores:	6720
Total # of Threads:	13440
Cluster:	Y

Client Specific Information

# of Clients:	64
CPU Type:	Intel Xeon Platinum 8163 2.50GHz
Total # of Processors:	128
Total # of Cores:	2048
Total # of Threads:	4096

<https://blog.csdn.net/jiangbp8686>


阿里的 OceanBase 服务器使用了 6720 个 2.5GHz 的处理器，整体系统价值 3.8 亿元人民币，总体性能达到 6088 万笔/分钟，平均每笔交易成本为 6.25 元人民币，按资源计算，平均 8928 笔/核。



SPARC SuperCluster with T3-4 Servers

Reference URL: <http://www.tpc.org/1780>

Benchmark Stats

Result ID:	110120201
Status: 	Historical Result
Report Date:	12/02/10
TPC-C Rev:	5.11.0

System Information

Total System Cost:	30,528,863 USD
Performance:	30,249,688 tpmC
Price/Performance:	1.01 USD per tpmC
TPC-Energy Metric:	Not reported
Availability Date:	06/01/11
Active Expiration Date:	12/02/13
Database Manager:	Oracle Database 11g R2 Enterprise Edition w/RAC w/Partitioning
Operating System:	Oracle Solaris 10 09/10
Transaction Monitor:	Oracle Tuxedo CFSR

Server Specific Information

CPU Type:	Oracle SPARC T3 - 1.65 GHz
Total # of Processors:	108
Total # of Cores:	1728
Total # of Threads:	13824
Cluster:	Y

Client Specific Information

# of Clients:	81
CPU Type:	Intel Xeon X5670 6-Core- 2.93 GHz
Total # of Processors:	162
Total # of Cores:	972
Total # of Threads:	1944

<https://blog.csdn.net/jiangbb8686>

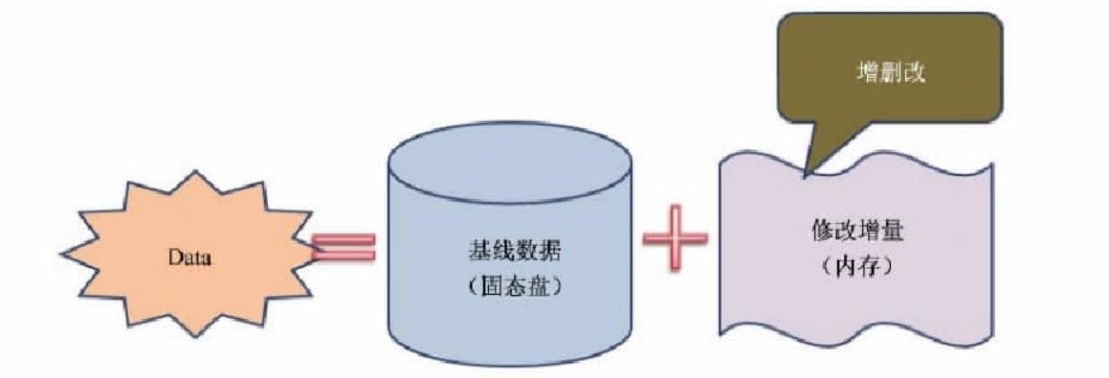
Oracle 11g 服务器采用了 1728 个 1.65GHz 的处理器，整体系统花费 3000 万美元，按照 2011 年的汇率计算约 2 亿元人民币，总体性能达到了 3000 万笔/分钟，平均每笔交易成本为 1 美元，约 7 元人民币，按资源计算，平均 17361 笔/核。

综合来看，一方面，阿里的 OceanBase 是分布式数据库，分布式数据库相比传统的集中式数据库有先天的优势，性能优于集中式数据库是必然的结果，理论上来讲，分布式数据库是传统集中式数据库性能的数倍甚至数十倍，但是我们看到的仅仅是两倍的性能，可以说，从这一点上来看，阿里的分布式数据库没有体现出分布式数据库的优势，或者说，阿里的分布式数据库技术还有很大的上升空间。另一方面，从成本的角度来考虑，OceanBase 的测试结果构建在近 4 亿元系统的之上，平均每笔交易成本 6.25 元人民币。Oracle 11g 的测试结果构建在 3000 万美元的基础上，平均每笔交易成本 7 元人民币。Oracle 无论是硬件还是软件，都是基于自家生产，成本本来就很低，而阿里云是基于 PC Server，操作系统使用的免费的 Linux，成本理应很低，而现在的硬件成本相比 2011 年要便宜不止数倍的价格。资源使用效率上，OceanBase 平均 8928 笔/核/2.5GHz，Oracle 11g 平均 17361 笔/核/1.65GHz。这样算下来的话，阿里的资源

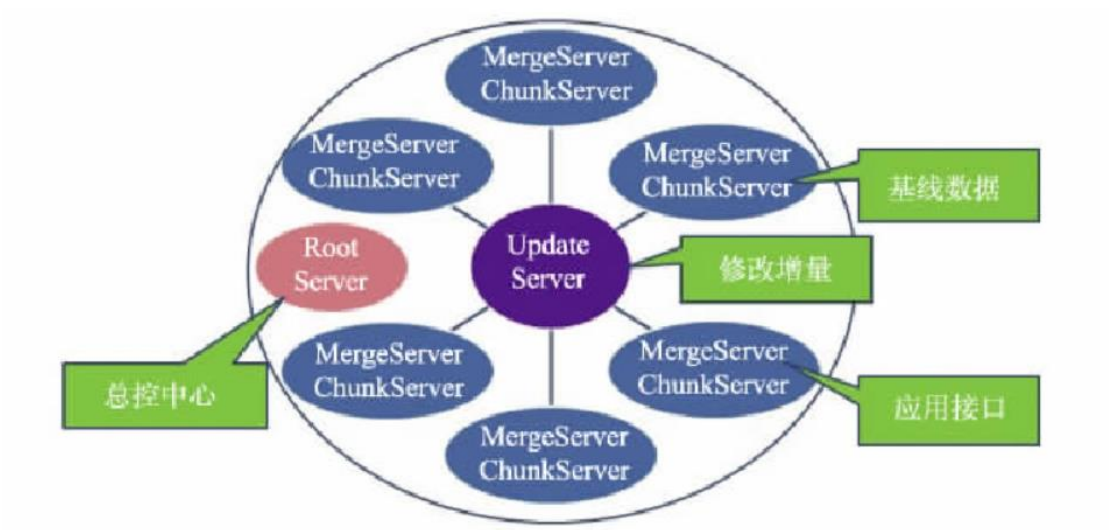
使用效率要远远低于 Oracle，因此还有很大空间很多地方需要提升和改进，来不断提升性能的同时降低价格，达到更好的性价比。

2.4 技术特点分析

由于大多数数据库一天的增删改数据量相对于数据库的记录数比例较小，因此 OceanBase 以增量方式把当天增删改的数据保存在服务器内存中(Redo log 保存在磁盘)，称为 MemTable；对应的基线数据（即数据库在 MemTable 开始时刻的快照）则分割后保存在磁盘（通常是固态硬盘）上，称为 Sstable。



由于增删改数据 MemTable 通常较少，所以 MemTable 通常保存在服务器（称为 Up-dateServer）的内存，以增量方式保存，多次修改之间以指针连接；基线数据 Sstable 通常保存在多台服务器（称为 ChunkServer）的磁盘，以数据页（page，例如 8 KB）方式存储和访问。如图所示是单机群 OceanBase 的架构图。



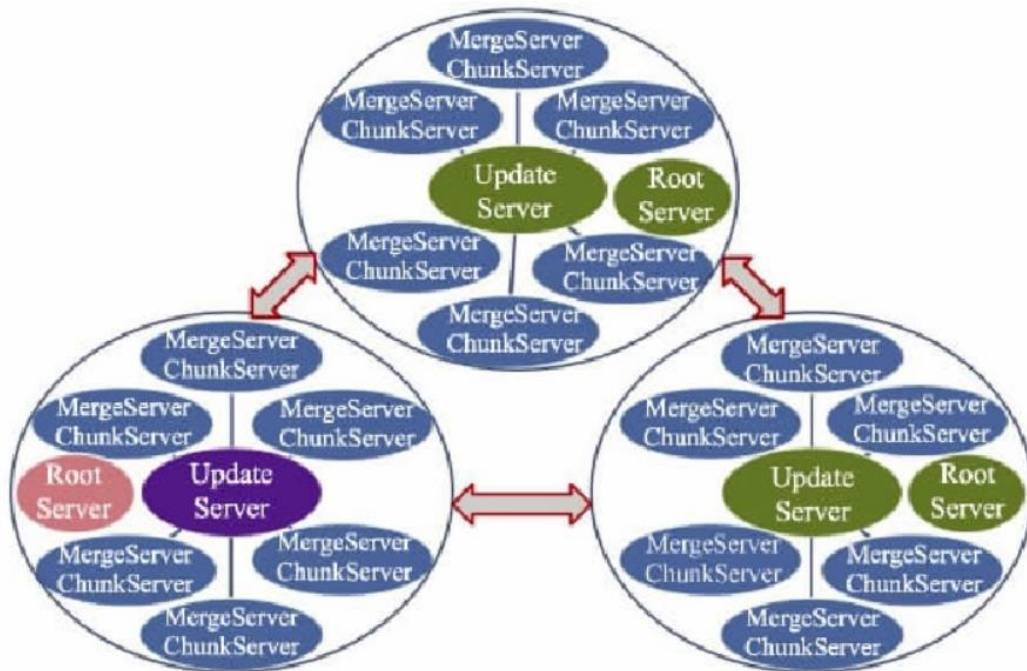
OceanBase 系统架构中，包括四类服务器：主控服务器 RootServer、基线数据服务器 ChunkServer、合并服务器 MergeServer，以及更新服务器 UpdateServer。

（1）主控服务器 RootServer。该服务器是 OceanBase 的总控中心，负责 ChunkServer/MergeServer 的上线、下线管理以及 Sstable 的负载均衡。

（2）基线数据服务器 ChunkServer。ChunkServer 保存基线数据 Sstable 并提供读访问。为了防止由于 ChunkServer 故障导致服务不可用甚至数据丢失，每个 Sstable 保存了多个副本并分布在不同的 ChunkServer 上。

(3) 合并服务器 MergeServer。OceanBase 的应用接口，支持 JDBC/ODBC 协议，接收并解析用户的 SQL 请求，经过词法分析、语法分析、生成执行计划等一系列操作后，发送到相应的 ChunkServer。

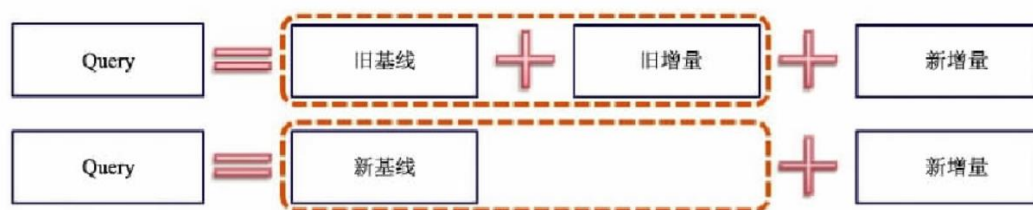
(4) 更新服务器 UpdateServer。执行写事务、保存修改增量（到内存）、写 Redo log（到磁盘）并提供读服务。UpdateServer 通常也有多个副本。由于 UpdateServer 内存是有限的，修改后的增量无法一直保存在内存中，因此 OceanBase 通常每天在某个时刻（例如后半夜业务低谷期）冻结当前 MemTable 并开启新的 MemTable；此后新的增删改写入新的 MemTable，然后系统在后台把冻结的 MemTable 与当前基线数据融合，生成新的基线数据，这个过程称为每日合并。每日合并完成后，冻结的 MemTable 以及旧的 Sstable 即可释放，其占用的内存和磁盘空间也被回收。为了防止 UpdateServer 故障或者机房故障导致服务不可用甚至数据丢失，OceanBase 通常部署多个机群（例如一主两备），主机群执行写事务并且至少同步到一个或多个备用机群（超过半数）。这样任何一个机群异常都不会导致服务不可用，从而保证了数据库的高可用性。其架构如图所示。传统关系数据库通常采用主备库镜像，主库备库之间的网络异常以及备库异常都可能导致主库备库不同步；OceanBase 的多机群（ ≥ 3 ）方式使得不仅单个备机群异常不会影响业务，而且使得，即使主机群突然故障，数据库系统也会在最多若干秒后自动恢复服务，不会造成任何数据丢失。



2.4.1 读事务

对于用户的读取任务，MergeServer 接受 SQL 请求并解析生成 SQL 执行计划。MergeServer 确定事务中数据的基线数据在哪些 ChunkServer 上，然后通知这些 ChunkServer 执行对应的读请求。由于基线数据与修改增量的分离，如图所示，读事务需要融合基线数据和修改的更新数据，返回满足查询条件的数据。一个特殊的情况是在每日合并期间，让所有 ChunkServer 同步完成每日合并或者同步启用新的 Sstable 可能非常复杂甚至无法完成（比如

某个 ChunkServer 网络异常)，因此某些 ChunkServer 的部分 Sstable 完成了与冻结的 MemTable 的融合，生成了新的 Sstable，其他 Sstable 尚未完成融合，有些查询可能用到旧的 Sstable，有些查询可能用到新的 Sstable，还有的查询可能同时用到旧的 Sstable 和新的 Sstable。相同内容的 SQL 语句，到达的 MergeServer 不同或到达时间不同，使用新旧 Sstable 的情况可能有所不同，那么查询的结果是否一致呢？答案是肯定的。因为查询使用旧基线数据时，会融合旧增量数据（即冻结的 MemTable）和新增量数据（即新的 MemTable），查询使用新基线数据时，则只融合新增量数据（新的 MemTable），因此得到的结果是一致的，如图所示。



尽管使用了与传统关系数据库相似的方式存储，Sstable 的随机读并没有成为 OceanBase 的瓶颈，这得益于 OceanBase 通常采用固态硬盘作为存储并且没有随机磁盘写，一个没有随机磁盘写的固态硬盘可以提供每秒几万次的随机读（一个机械盘每秒只能提供几百次的随机读）。传统的基于磁盘的关系数据库，其读事务操作的基本步骤是先从磁盘中读出数据页（page），再从中取出需要的内容，然后根据用户的 SQL 进行操作得到结果。OceanBase 的读事务操作与它们类似，不同的是 OceanBase 从读出的数据页中取出需要的内容时，还得与对应的修改增量融合。由于被修改数据所占的比例比较小、修改增量以及融合操作都在内存中，这个操作对性能的损耗很小。与此同时，由于使用固态硬盘并且没有随机写，OceanBase 能够充分利用固态硬盘优异的随机读性能，因此能够获得很好的读性能。

2.4.2 写事务

MergeServer 解析 SQL 请求生成 SQL 执行计划后，如果不是只读事务，则向 ChunkServer 获取对应的基线数据，然后连同执行计划提交给主机群 UpdateServer 执行。主机群 UpdateServer 执行写事务，生成 Redo log，同步给备机群并持久化到各自磁盘，如果成功者（包括自己）超过了半数，则在 MemTable 中提交该事务并应答客户。尽管有大量的随机访问，由于增删改的修改增量放置在内存，Redo log 是顺序写，所以 OceanBase 系统中没有随机写磁盘。不仅如此，OceanBase 的 UpdateServer 服务器通常配置带电池或电容的 RAID 卡，这种 RAID 带有内存（例如 1 GB）并且在服务器断电时能够保持其中的数据不丢失，当小块的 Redo log 写入磁盘时，其实只是写到了 RAID 卡的内存中，该 RAID 卡稍后批量把其内存中数据写到磁盘上，这样不仅缩短了日志刷盘的时间（大约 0.1ms），而且降低了小块 Redo log 的写入对固态硬盘的性能和寿命的影响（因为固态硬盘的写入也是以页（page）为单位的，例如 4 KB，并且写入前需要先擦除，在已有页上即使追加一个字节也需要把原有页读出来，与追加字节合并，然后写入一个新的页）。每日合并期间把修改增量与旧

基线数据合并生成新基线数据时，对磁盘的访问是顺序读和顺序写，当施加一定的流量控制后，这种顺序读和顺序写对磁盘的随机读的影响也是可控的。传统的基于磁盘的关系数据库，其写事务操作的基本步骤是从磁盘中读出数据页（page），再从中取出需要的内容，然后根据用户的 SQL 进行修改，再把修改后的结果与原数据页融合生成新的数据页，写日志（Redo log、Undo log）并把新的数据页刷到磁盘，由于每次修改通常在几十到几百字节，而数据页的大小通常在几 KB（例如 8 KB），这就出现了明显的写放大（ $8\text{ KB}/100 \approx 80X$ ）。OceanBase 的写事务操作也是先从磁盘中读出数据页，再从中取出需要的内容，然后根据用户的 SQL 进行修改操作以生成增删改的增量，写日志（Redo log）并且把修改增量加入到 MemTable。由于内存中修改增量没有也不需要做成数据页，因此 OceanBase 不仅省去了写新的数据页到磁盘的操作（随机写磁盘），也避免了传统数据库的写入放大（每日合并通常在系统低谷期间进行，对业务影响很小，更不会影响高峰期的业务）。这使得 OceanBase 在写事务性能上有明显的优势。

2.4.3 性能经济分析

目前，国内外的学术界和工业界在支撑互联网应用的数据库产品方面做了大量努力。Google 的 Bigtable 是一个基于 Google File System 的分布式表格系统，只支持单行事务。Google 的 Spanner 实现了跨越全球的分布式事务，但复杂 SQL 性能较低，且对时钟有很强的依赖。OceanBase 系统是在保证“强一致性”的前提下，追求系统的“最大可用性”，实现面向互联网业务需求的强伸缩性、高性能、高可用和低成本通用数据库。

根据数据存储方式的不同，可以把目前的数据库分为三类：外存型数据库、全内存数据库以及内外存混合型数据库。

（1）外存型数据库。这种类型的数据库的读写事务都基于外存（磁盘），内存作为缓存（Cache），数据的存储基于数据页（page），读和写都有一定的放大效应，写入放大和磁盘随机写性能限制了数据库的性能。多数传统的关系数据库都属于这个类型。

（2）全内存数据库。这种类型的数据库数据全部在内存中（log 除外），没有数据页的概念，数据读写都在内存中进行（写日志除外），性能很高。VolteDB (<http://voltdb.com>) 和 MemSQL (<http://www.memsql.com>) 属于这个类型，并且都号称是全世界最快的内存数据库。

（3）内外存混合型的数据库。这种类型的数据库部分数据在外存（磁盘），部分数据在内存，在磁盘上的数据以数据页为单位存储，内存中的数据不使用数据页，内存也不仅仅作为缓存。OceanBase 属于这个类型，其基线数据以数据页方式保存在磁盘上，而增删改的增量则在内存中，性能介于磁盘型的数据库与全内存数据库之间。

从存储方式来看，OceanBase 属于内外存混合型数据库；与内存数据库相比，OceanBase 在读写事务性能上的劣势不大，但在经济性上优势十分明显，具体在以下几方面。

（1）读事务性能。内存数据库完全避免了磁盘 I/O（写日志除外）并且也没有数据页导致的读放大。然而，由于固态盘的随机读响应时间（0.1 ms）仅为机械盘（3~5 ms）的几十分之一，并且由于二八法则，OceanBase 可以用相对较

小的内存代价（比如全量数据的 20%）缓存少部分比较热的数据，再加上 query Cache，因此读事务并不会成为 OceanBase 的性能瓶颈。

（2）写事务性能。OceanBase 的写事务也是内存操作，与全内存数据库类似，其中的读操作，由于 SSD 优异的随机读性能以及缓存，因此 OceanBase 与全内存数据库的写性能的差距不大。

（3）经济性。二八法则表明，一段时间（例如小时或者天）内数据库中的数据只有小部分会被频繁访问，大部分数据被很少访问或者根本没有被访问。把这些很少或没有被访问的数据与频繁访问的数据同等对待并一样占用昂贵的内存资源，显然是不经济的。由于固态硬盘容量大大高于内存容量，同等尺寸的数据量，内存数据库需要的服务器数量也比内外存混合型的 OceanBase 数据库需要的服务器数量多得多。

2.5 OceanBase 前景展望

最近，即使是数据库这样非常成熟的细分领域也发生了不少动荡：谷歌凭借 Spanner 从一招鲜玩家杀入到远见者，阿里云一举跻身远见者，且拥有最多的 DBMS 服务品种；亚马逊连年快速上升，如今已经跟 Oracle、微软非常接近。

OceanBase 当初没有选择基于开源或已有的技术思路开发，而是选择走分布式自研这条路，虽然走得艰难，但做成之后就会成为不可替代的优势。过去这十来年正好是分布式系统发展的十来年，转型到分布式已经成为所有人都认可的一个选择。如今，以 Google Spanner、蚂蚁金服的 OceanBase 为代表的分布式关系数据库，不仅解决了关系数据库的扩展性问题，也极大地降低了关系数据库的成本（数量级的硬件成本的降低），还提升了可用性。

现在，兼容 Oracle 的工作是 OceanBase 的重中之重。OceanBase 团队的目标是，用两年时间做到 Oracle 业务的平滑迁移，不需要修改一行代码、不需要业务做任何调整就能够将数据库迁移过来。

对于数据库的未来，蚂蚁金服高级研究员阳振坤表示：“尽管今天在业界，数据仓库主要依赖的不是关系数据库，但可以看看 Google。今天 Google 的大数据分析/数据库仓库基本都统一到了 Spanner，这应该是 5-10 年后产业界的写照。”未来，OceanBase 还会走得更快、更远。