数据库

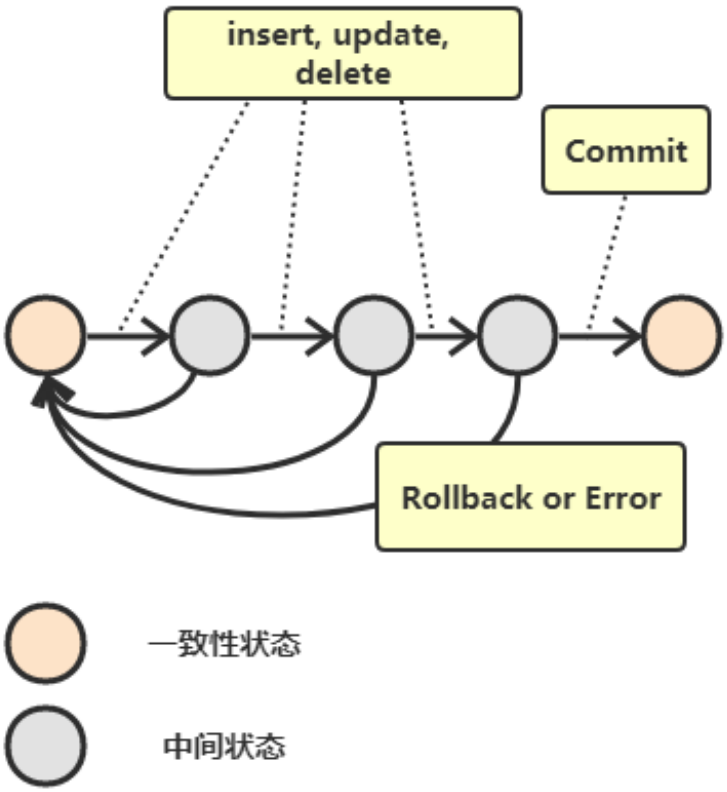
数据库系统原理

https://github.com/CyC2018/Interview-Notebook

PDF制作github: https://github.com/sjsdfg/Interview-Notebook-PDF

一、事务

概念



ACID

1. 原子性（Atomicity）

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用日志来实现，日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。

2. 一致性（Consistency）

数据库在事务执行前后都保持一致性状态。

在一致性状态下，所有事务对一个数据的读取结果都是相同的。

3. 隔离性（Isolation）

一个事务所做的修改在最终提交以前，对其它事务是不可见的。

4. 持久性（Durability）

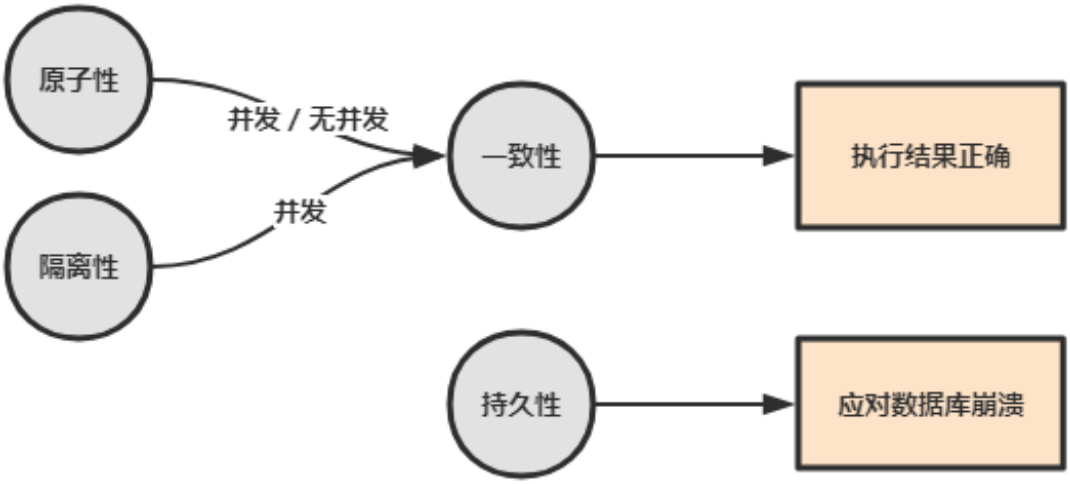
一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

可以通过数据库备份和恢复来实现，在系统发生奔溃时，使用备份的数据库进行数据恢复。

事务的 ACID 特性概念简单，但不是很好理解，主要是因为这几个特性不是一种平级关系：

- 只有满足一致性，事务的执行结果才是正确的。
- 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时要只要能满足原子性，就一定能满足一致性。
- 在并发的情况下，多个事务并发执行，事务不仅要满足原子性，还需要满足隔离性，才能满足一致性。

- 事务满足持久化是为了能应对数据库奔溃的情况。



AUTOCOMMIT

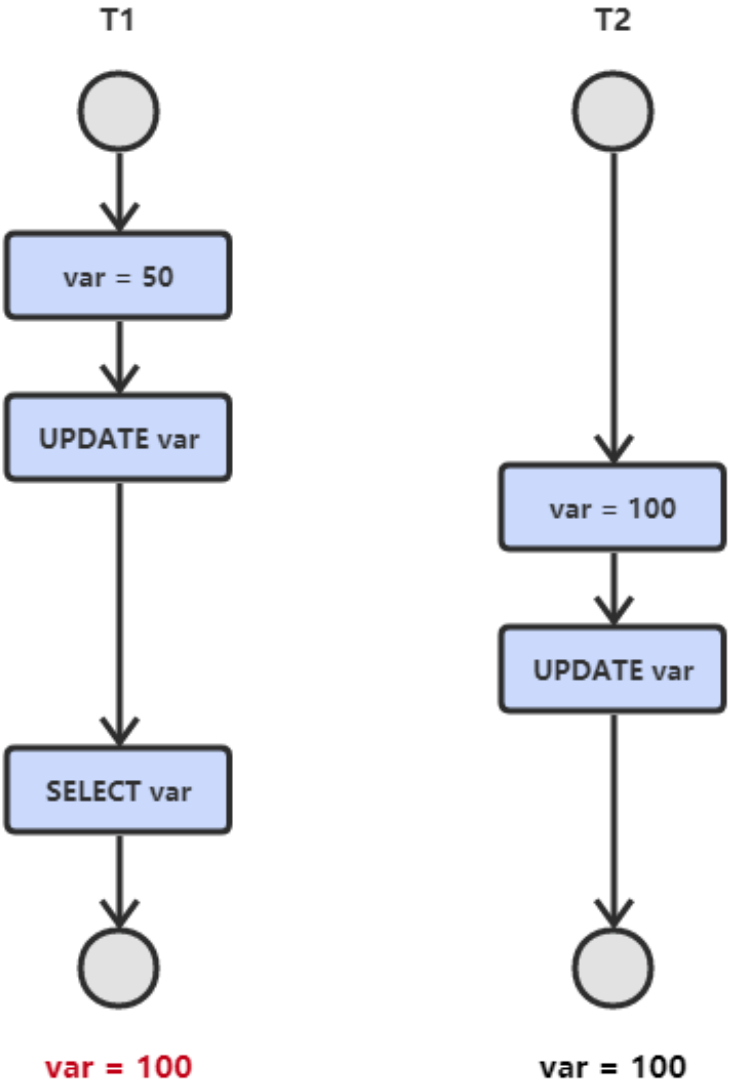
MySQL 默认采用自动提交模式。也就是说，如果不显式使用`START TRANSACTION`语句来开始一个事务，那么每个查询都会被当做一个事务自动提交。

二、并发一致性问题

在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性问题。

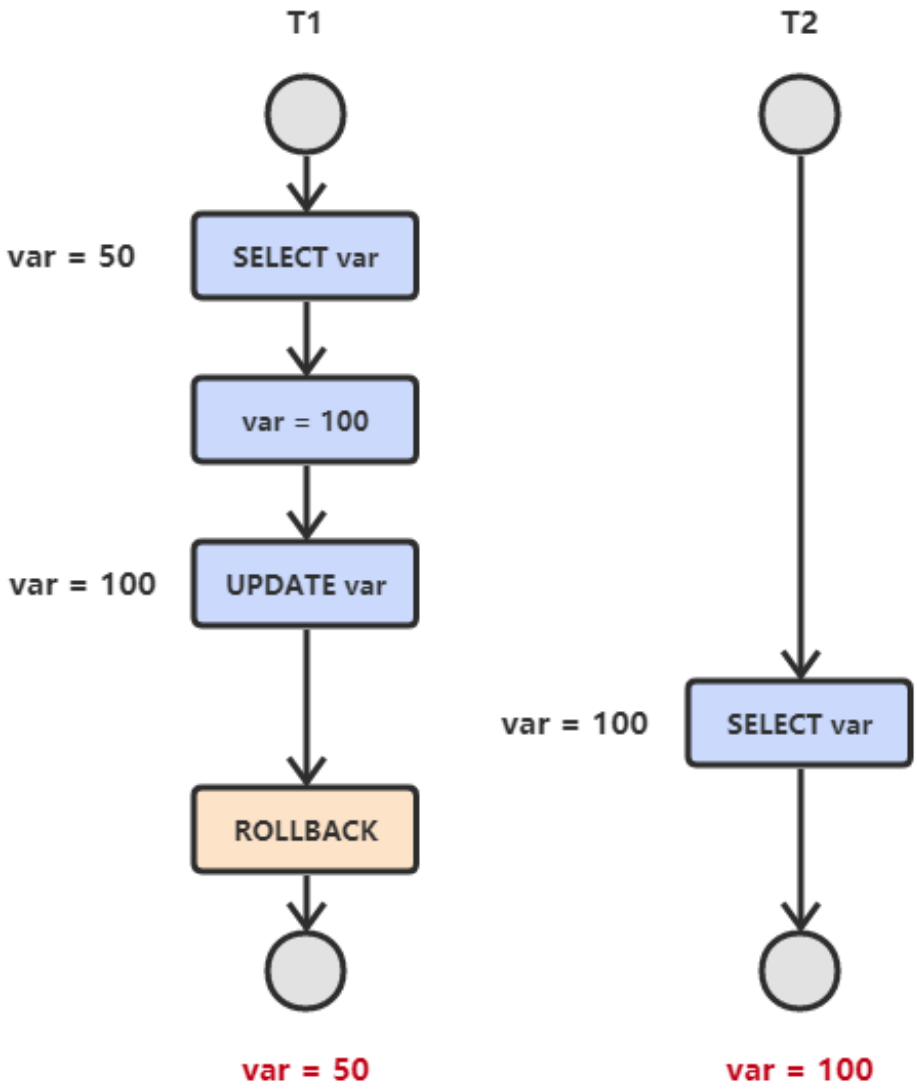
丢失修改

T_1 和 T_2 两个事务都对一个数据进行修改，T_1 先修改，T_2 随后修改，T_2 的修改覆盖了 T_1 的修改。



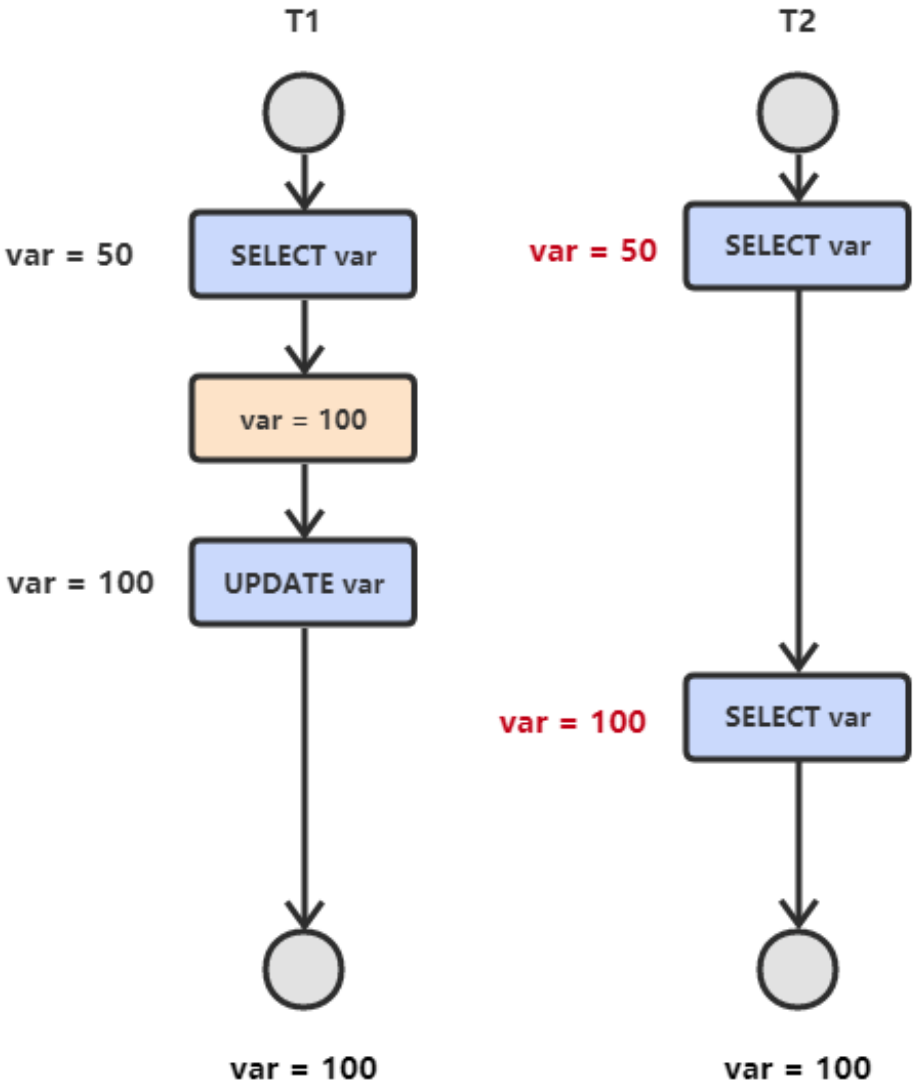
读脏数据

T_1 修改一个数据，T_2 随后读取这个数据。如果 T_1 撤销了这次修改，那么 T_2 读取的数据是脏数据。



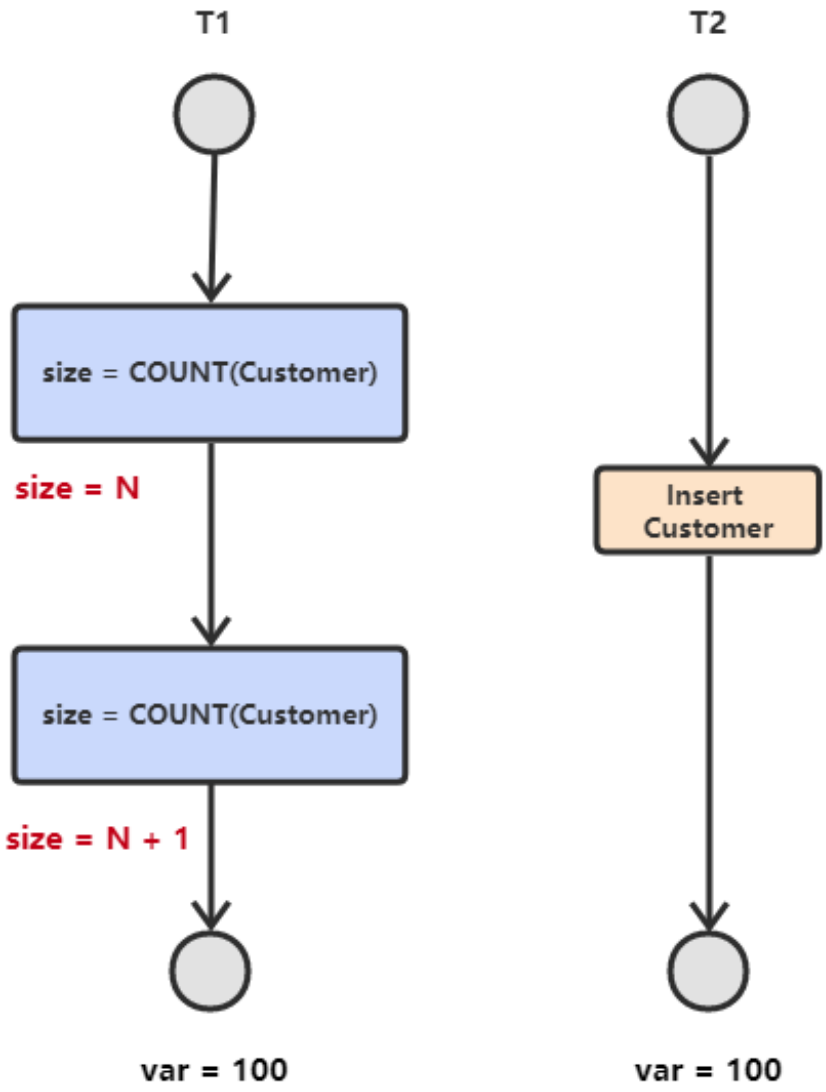
不可重复读

T_2 读取一个数据，T_1 对该数据做了修改。如果 T_2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。



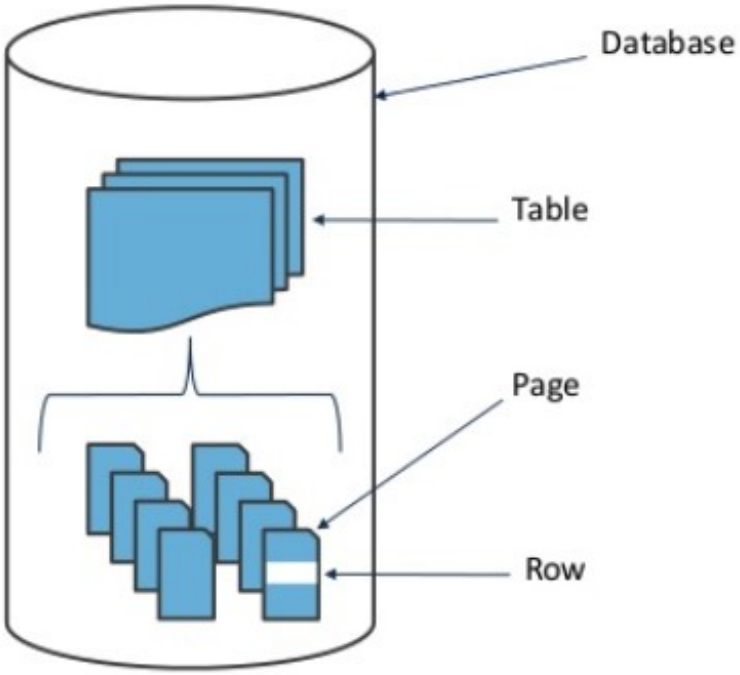
幻影读

T_1 读取某个范围的数据，T_2 在这个范围内插入新的数据，T_1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。


T1
T2
size = COUNT(Customer)
size = N
Insert Customer
size = COUNT(Customer)
size = N + 1
var = 100
var = 100

三、封锁

封锁粒度



封锁类型

1. 读写锁

- 排它锁（Exclusive），简写为 X 锁，又称写锁。
- 共享锁（Shared），简写为 S 锁，又称读锁。

有以下两个规定：

- 一个事务对数据对象 A 加了 X 锁，就可以对 A 进行读取和更新。加锁期间其它事务不能对 A 加任何锁。
- 一个事务对数据对象 A 加了 S 锁，可以对 A 进行读取操作，但是不能进行更新操作。加锁期间其它事务能对 A 加 S 锁，但是不能加 X 锁。

锁的兼容关系如下：

-	X	S
X	NO	NO

-	X	S
S	NO	YES

2. 意向锁

使用意向锁（Intention Locks）可以更容易地支持多粒度封锁。

在存在行级锁和表级锁的情况下，事务 T 想要对表 A 加 X 锁，就需要先检测是否有其它事务对表 A 或者表 A 中的任意一行加了锁，那么就需要对表 A 的每一行都检测一次，这是非常耗时的。

意向锁在原来的 X/S 锁之上引入了 IX/IS，IX/IS 都是表锁，用来表示一个事务想要在表中的某个数据行上加 X 锁或 S 锁。有以下两个规定：

- 一个事务在获得某个数据行对象的 S 锁之前，必须先获得表的 IS 锁或者更强的锁；
- 一个事务在获得某个数据行对象的 X 锁之前，必须先获得表的 IX 锁。

通过引入意向锁，事务 T 想要对表 A 加 X 锁，只需要先检测是否有其它事务对表 A 加了 X/IX/S/IS 锁，如果加了就表示有其它事务正在使用这个表或者表中某一行的锁，因此事务 T 加 X 锁失败。

各种锁的兼容关系如下：

-	X	IX	S	IS
X	NO	NO	NO	NO
IX	NO	YES	NO	YES
S	NO	NO	YES	YES
IS	NO	YES	YES	YES

解释如下：

- 任意 IS/IX 锁之间都是兼容的，因为它们只是表示想要对表加锁，而不是真正加锁；
- S 锁只与 S 锁和 IS 锁兼容，也就是说事务 T 想要对数据行加 S 锁，其它事务可以已经获

得对表或者表中的行的 S 锁。

封锁协议

1. 三级封锁协议

一级封锁协议

事务 T 要修改数据 A 时必须加 X 锁，直到 T 结束才释放锁。

可以解决丢失修改问题，因为不能同时有两个事务对同一个数据进行修改，那么事务的修改就不会被覆盖。

T_1	T_2
lock-x(A)	
read A=20	
	lock-x(A)
	wait
write A=19	.
commit	.
unlock-x(A)	.
	obtain
	read A=19
	write A=21
	commit
	unlock-x(A)

二级封锁协议

在一级的基础上，要求读取数据 A 时必须加 S 锁，读取完马上释放 S 锁。

可以解决读脏数据问题，因为如果一个事务在对数据 A 进行修改，根据 1 级封锁协议，会加 X 锁，那么就不能再加 S 锁了，也就是不会读入数据。

T_1	T_2
lock-x(A)	
read A=20	
write A=19	
	lock-s(A)
	wait
rollback	.
A=20	.
unlock-x(A)	.
	obtain
	read A=20
	commit
	unlock-s(A)

三级封锁协议

在二级的基础上，要求读取数据 A 时必须加 S 锁，直到事务结束了才能释放 S 锁。

可以解决不可重复读的问题，因为读 A 时，其它事务不能对 A 加 X 锁，从而避免了在读的期间数据发生改变。

T_1	T_2
lock-s(A)	
read A=20	

T	T
	lock-x(A)
	wait
read A=20	.
commit	.
unlock-s(A)	.
	obtain
	read A=20
	write A=19
	commit
	unlock-X(A)

2. 两段锁协议

加锁和解锁分为两个阶段进行。

可串行化调度是指，通过并发控制，使得并发执行的事务结果与某个串行执行的事务结果相同。

事务遵循两段锁协议是保证可串行化调度的充分条件。例如以下操作满足两段锁协议，它是可串行化调度。

```
lock-x(A)...lock-s(B)...lock-s(C)...unlock(A)...unlock(C)...unlock(B)
```

但不是必要条件，例如以下操作不满足两段锁协议，但是它还是可串行化调度。

```
lock-x(A)...unlock(A)...lock-s(B)...unlock(B)...lock-s(C)...unlock(C)
```

MySQL 隐式与显示锁定

MySQL 的 InnoDB 存储引擎采用两段锁协议，会根据隔离级别在需要的时候自动加锁，并且所有的锁都是在同一时刻被释放，这被称为隐式锁定。

InnoDB 也可以使用特定的语句进行显示锁定：

```
SELECT ... LOCK In SHARE MODE;
SELECT ... FOR UPDATE;
```

四、隔离级别

未提交读（READ UNCOMMITTED）

事务中的修改，即使没有提交，对其它事务也是可见的。

提交读（READ COMMITTED）

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。

可重复读（REPEATABLE READ）

保证在同一个事务中多次读取同样数据的结果是一样的。

可串行化（SERIALIZABLE）

强制事务串行执行。

隔离级别	脏读	不可重复读	幻影读
未提交读	YES	YES	YES
提交读	NO	YES	YES
可重复读	NO	NO	YES
可串行化	NO	NO	NO

五、多版本并发控制

多版本并发控制（Multi-Version Concurrency Control, MVCC）是 MySQL 的 InnoDB 存储引擎实现隔离级别的一种具体方式，用于实现提交读和可重复读这两种隔离级别。而未提交读隔离级别总是读取最新的数据行，无需使用 MVCC；可串行化隔离级别需要对所有读取的行都加锁，单纯使用 MVCC 无法实现。

版本号

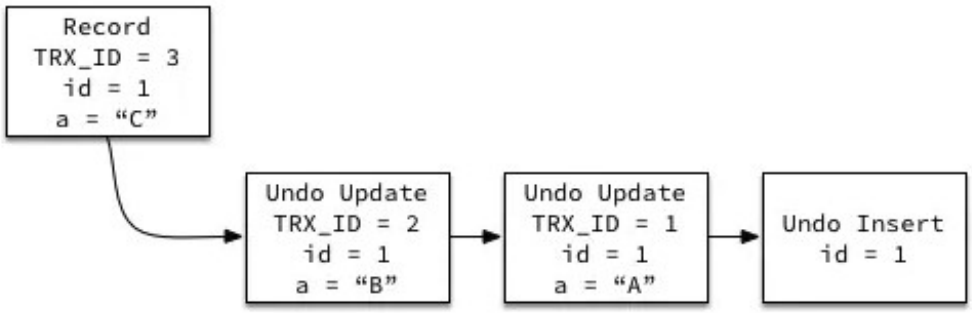
- 系统版本号：是一个递增的数字，每开始一个新的事务，系统版本号就会自动递增。
- 事务版本号：事务开始时的系统版本号。

InooDB 的 MVCC 在每行记录后面都保存着两个隐藏的列，用来存储两个版本号：

- 创建版本号：指示创建一个数据行的快照时的系统版本号；
- 删除版本号：如果该快照的删除版本号大于当前事务版本号表示该快照有效，否则表示该快照已经被删除了。

Undo 日志

InnoDB 的 MVCC 使用到的快照存储在 Undo 日志中，该日志通过回滚指针把一个数据行（Record）的所有快照连接起来。



实现过程

以下实现过程针对可重复读隔离级别。

1. SELECT

当开始新一个事务时，该事务的版本号肯定会大于当前所有数据行快照的创建版本号，理解这一点很关键。

多个事务必须读取到同一个数据行的快照，并且这个快照是距离现在最近的一个有效快照。但是也有例外，如果有一个事务正在修改该数据行，那么它可以读取事务本身所做的修改，而不用和其它事务的读取结果一致。

把没有对一个数据行做修改的事务称为 T，T 所要读取的数据行快照的创建版本号必须小于 T 的版本号，因为如果大于或者等于 T 的版本号，那么表示该数据行快照是其它事务的最新修改，因此不能去读取它。

除了上面的要求，T 所要读取的数据行快照的删除版本号必须大于 T 的版本号，因为如果小于等于 T 的版本号，那么表示该数据行快照是已经被删除的，不应该去读取它。

2. INSERT

将当前系统版本号作为数据行快照的创建版本号。

3. DELETE

将当前系统版本号作为数据行快照的删除版本号。

4. UPDATE

将当前系统版本号作为更新后的数据行快照的创建版本号，同时将当前系统版本号作为更新前的数据行快照的删除版本号。可以理解为先执行 DELETE 后执行 INSERT。

快照读与当前读

1. 快照读

使用 MVCC 读取的是快照中的数据，这样可以减少加锁所带来的开销。

```
select * from table ...;
```

2. 当前读

读取的是最新的数据，需要加锁。以下第一个语句需要加 S 锁，其它都需要加 X 锁。

```
select * from table where ? lock in share mode;
select * from table where ? for update;
insert;
update;
delete;
```

六、Next-Key Locks

Next-Key Locks 也是 MySQL 的 InnoDB 存储引擎的一种锁实现。MVCC 不能解决幻读的问题，Next-Key Locks 就是为了解决这个问题而存在的。在可重复读（REPEATABLE READ）隔离级别下，使用 MVCC + Next-Key Locks 可以解决幻读问题。

Record Locks

锁定整个记录（行）。锁定的对象是记录的索引，而不是记录本身。如果表没有设置索引，InnoDB 会自动在主键上创建隐藏的聚集索引，因此 Record Locks 依然可以使用。

Gap Locks

锁定一个范围内的索引，例如当一个事务执行以下语句，其它事务就不能在 t.c 中插入 15。

```
SELECT c FROM t WHERE c BETWEEN 10 and 20 FOR UPDATE;
```

Next-Key Locks

它是 Record Locks 和 Gap Locks 的结合，不仅锁定一个记录，也锁定范围内的索引。在 user 中有以下记录：

```
|   id | last_name    | first_name   |   age |
|------|--------------|--------------|-------|
|    4 | stark        | tony         |    21 |
|    1 | tom          | hiddleston   |    30 |
|    3 | morgan       | freeman      |    40 |
|    5 | jeff         | dean         |    50 |
|    2 | donald       | trump        |    80 |
+------|--------------|--------------|-------+
```

那么就需要锁定以下范围：

```
(-∞, 21]
(21, 30]
(30, 40]
(40, 50]
(50, 80]
(80, ∞)
```

七、关系数据库设计理论

函数依赖

记 A->B 表示 A 函数决定 B，也可以说 B 函数依赖于 A。

如果 {A1，A2，... ，An} 是关系的一个或多个属性的集合，该集合函数决定了关系的其它所有属性并且是最小的，那么该集合就称为键码。

对于 A->B，如果能找到 A 的真子集 A'，使得 A'-> B，那么 A->B 就是部分函数依赖，否则就是完全函数依赖；

对于 A->B，B->C，则 A->C 是一个传递函数依赖。

异常

以下的学生课程关系的函数依赖为 Sno, Cname -> Sname, Sdept, Mname, Grade，键码为 {Sno, Cname}。也就是说，确定学生和课程之后，就能确定其它信息。

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

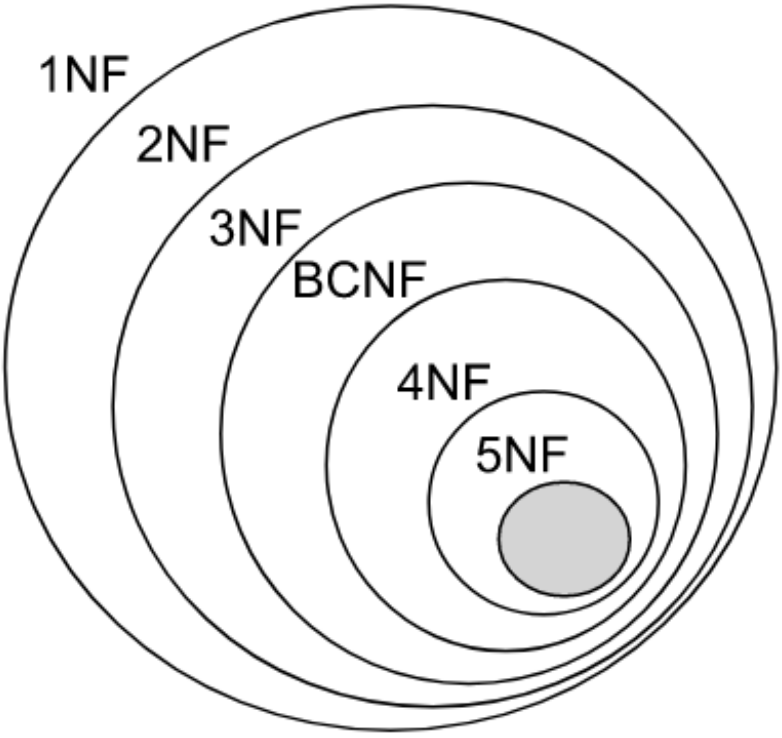
不符合范式的关系，会产生很多异常，主要有以下四种异常：

- 冗余数据：例如 学生-2 出现了两次。
- 修改异常：修改了一个记录中的信息，但是另一个记录中相同的信息却没有被修改。
- 删除异常：删除一个信息，那么也会丢失其它信息。例如如果删除了 课程-1，需要删除第一行和第三行，那么 学生-1 的信息就会丢失。
- 插入异常，例如想要插入一个学生的信息，如果这个学生还没选课，那么就无法插入。

范式

范式理论是为了解决以上提到四种异常。

高级别范式的依赖于低级别的范式，1NF 是最低级别的范式。



1. 第一范式 (1NF)

属性不可分；

2. 第二范式 (2NF)

每个非主属性完全函数依赖于键码。

可以通过分解来满足。

分解前

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

以上学生课程关系中，{Sno, Cname} 为键码，有如下函数依赖：

- Sno -> Sname, Sdept
- Sdept -> Mname
- Sno, Cname-> Grade

Grade 完全函数依赖于键码，它没有任何冗余数据，每个学生的每门课都有特定的成绩。

Sname, Sdept 和 Mname 都部分依赖于键码，当一个学生选修了多门课时，这些数据就会出现多次，造成大量冗余数据。

分解后

关系-1

Sno	Sname	Sdept	Mname
1	学生-1	学院-1	院长-1
2	学生-2	学院-2	院长-2
3	学生-3	学院-2	院长-2

有以下函数依赖：

- Sno -> Sname, Sdept, Mname
- Sdept -> Mname

关系-2

Sno	Cname	Grade
1	课程-1	90
2	课程-2	80
2	课程-1	100
3	课程-2	95

有以下函数依赖：

- Sno, Cname -> Grade

3. 第三范式 (3NF)

非主属性不传递函数依赖于键码。

上面的 关系-1 中存在以下传递函数依赖：Sno -> Sdept -> Mname，可以进行以下分解：

关系-11

Sno	Sname	Sdept
1	学生-1	学院-1
2	学生-2	学院-2
3	学生-3	学院-2

关系-12

Sdept	Mname
学院-1	院长-1
学院-2	院长-2

八、ER 图

Entity-Relationship，有三个组成部分：实体、属性、联系。

用来进行关系型数据库系统的概念设计。

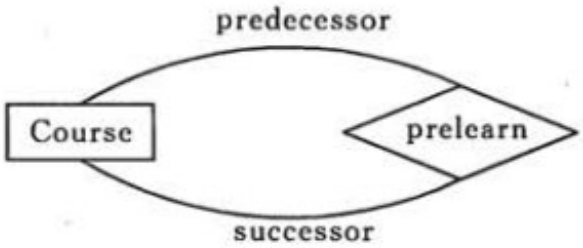
实体的三种联系

包含一对一，一对多，多对多三种。

如果 A 到 B 是一对多关系，那么画个带箭头的线段指向 B；如果是一对一，画两个带箭头的线段；如果是多对多，画两个不带箭头的线段。下图的 Course 和 Student 是一对多的关系。

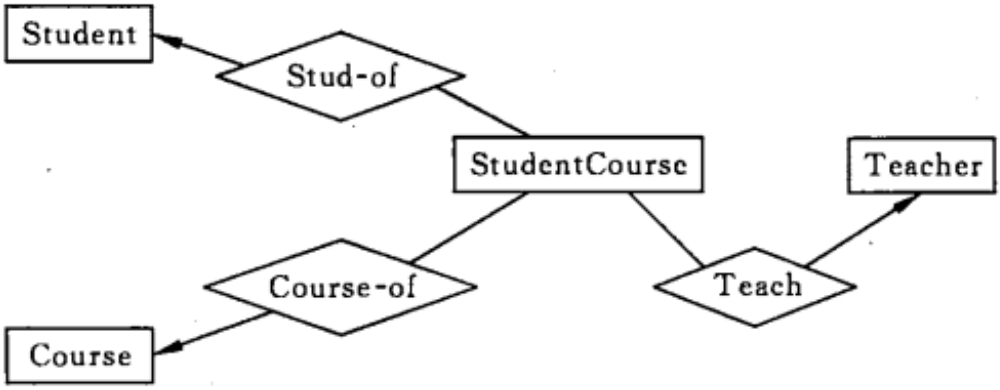
表示出现多次的关系

一个实体在联系出现几次，就要用几条线连接。下图表示一个课程的先修关系，先修关系出现两个 Course 实体，第一个是先修课程，后一个是后修课程，因此需要用两条线来表示这种关系。



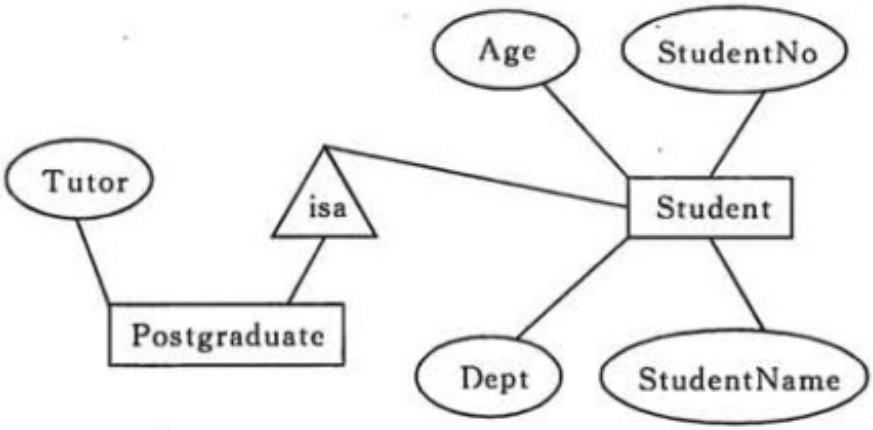
联系的多向性

虽然老师可以开设多门课，并且可以教授多名学生，但是对于特定的学生和课程，只有一个老师教授，这就构成了一个三元联系。



表示子类

用一个三角形和两条线来连接类和子类，与子类有关的属性和联系都连到子类上，而与父类和子类都有关的连到父类上。



参考资料

- AbrahamSilberschatz, HenryF.Korth, S.Sudarshan, 等. 数据库系统概念 [M]. 机械工业出版社, 2006.
- 施瓦茨. 高性能 MYSQL(第3版)[M]. 电子工业出版社, 2013.
- 史嘉权. 数据库系统概论[M]. 清华大学出版社有限公司, 2006.
- The InnoDB Storage Engine
- Transaction isolation levels
- Concurrency Control
- The Nightmare of Locking, Blocking and Isolation Levels!
- Database Normalization and Normal Forms with an Example
- The basics of the InnoDB undo logging and history system
- MySQL locking for the busy web developer
- 浅入浅出 MySQL 和 InnoDB
- Innodb 中的事务隔离级别和锁的关系

SQL

https://github.com/CyC2018/Interview-Notebook

PDF制作github: https://github.com/sjsdfg/Interview-Notebook-PDF

一、基础

模式定义了数据如何存储、存储什么样的数据以及数据如何分解等信息，数据库和表都有模式。

主键的值不允许修改，也不允许复用（不能使用已经删除的主键值赋给新数据行的主键）。

SQL（Structured Query Language)，标准 SQL 由 ANSI 标准委员会管理，从而称为 ANSI SQL。各个 DBMS 都有自己的实现，如 PL/SQL、Transact-SQL 等。

SQL 语句不区分大小写，但是数据库表名、列名和值是否区分依赖于具体的 DBMS 以及配置。

SQL 支持以下三种注释：

```
# 注释
SELECT *
FROM mytable; -- 注释
/* 注释1
   注释2 */
```

数据库创建与使用：

```
CREATE DATABASE test;
USE test;
```

二、创建表

```
CREATE TABLE mytable (
  id INT NOT NULL AUTO_INCREMENT,
  col1 INT NOT NULL DEFAULT 1,
  col2 VARCHAR(45) NULL,
  col3 DATE NULL,
  PRIMARY KEY (`id`));
```

三、修改表

添加列

```
ALTER TABLE mytable
ADD col CHAR(20);
```

删除列

```
ALTER TABLE mytable
DROP COLUMN col;
```

删除表

```
DROP TABLE mytable;
```

四、插入

普通插入

```
INSERT INTO mytable(col1, col2)
VALUES(val1, val2);
```

插入检索出来的数据

```
INSERT INTO mytable1(col1, col2)
SELECT col1, col2
FROM mytable2;
```

将一个表的内容插入到一个新表

```
CREATE TABLE newtable AS
SELECT * FROM mytable;
```

五、更新

```
UPDATE mytable
SET col = val
WHERE id = 1;
```

六、删除

```
DELETE FROM mytable
WHERE id = 1;
```

TRUNCATE TABLE 可以清空表，也就是删除所有行。

```
TRUNCATE TABLE mytable;
```

使用更新和删除操作时一定要用 WHERE 子句，不然会把整张表的数据都破坏。可以先用 SELECT 语句进行测试，防止错误删除。

七、查询

DISTINCT

相同值只会出现一次。它作用于所有列，也就是说所有列的值都相同才算相同。

```
SELECT DISTINCT col1, col2
FROM mytable;
```

LIMIT

限制返回的行数。可以有两个参数，第一个参数为起始行，从 0 开始；第二个参数为返回的总行数。

返回前 5 行：

```
SELECT *
FROM mytable
LIMIT 5;
```

```
SELECT *
FROM mytable
LIMIT 0, 5;
```

返回第 3 \~ 5 行：

```
SELECT *
FROM mytable
LIMIT 2, 3;
```

八、排序

- **ASC** ：升序（默认）
- **DESC** ：降序

可以按多个列进行排序，并且为每个列指定不同的排序方式：

```
SELECT *
FROM mytable
ORDER BY col1 DESC, col2 ASC;
```

九、过滤

不进行过滤的数据非常大，导致通过网络传输了多余的数据，从而浪费了网络带宽。因此尽量使用 SQL 语句来过滤不必要的数据，而不是传输所有的数据到客户端中然后由客户端进行过滤。

```
SELECT *
FROM mytable
WHERE col IS NULL;
```

下表显示了 WHERE 子句可用的操作符

操作符	说明
=	等于
<	小于
>	大于
<> !=	不等于
<= !>	小于等于
>= !<	大于等于
BETWEEN	在两个值之间
IS NULL	为 NULL 值

应该注意到，NULL 与 0、空字符串都不同。

AND 和 OR 用于连接多个过滤条件。优先处理 AND，当一个过滤表达式涉及到多个 AND 和 OR 时，可以使用 () 来决定优先级，使得优先级关系更清晰。

IN 操作符用于匹配一组值，其后也可以接一个 SELECT 子句，从而匹配子查询得到的一组值。

NOT 操作符用于否定一个条件。

十、通配符

通配符也是用在过滤语句中，但它只能用于文本字段。

- **%** 匹配 >=0 个任意字符；
- _ 匹配 ==1 个任意字符；
- **[ ]** 可以匹配集合内的字符，例如 [ab] 将匹配字符 a 或者 b。用脱字符 ^ 可以对其进行否定，也就是不匹配集合内的字符。

使用 Like 来进行通配符匹配。

```
SELECT *
FROM mytable
WHERE col LIKE '[^AB]%'; -- 不以 A 和 B 开头的任意文本
```

不要滥用通配符，通配符位于开头处匹配会非常慢。

十一、计算字段

在数据库服务器上完成数据的转换和格式化的工作往往比客户端上快得多，并且转换和格式化后的数据量更少的话可以减少网络通信量。

计算字段通常需要使用 **AS** 来取别名，否则输出的时候字段名为计算表达式。

```
SELECT col1 * col2 AS alias
FROM mytable;
```

CONCAT() 用于连接两个字段。许多数据库会使用空格把一个值填充为列宽，因此连接的结果会出现一些不必要的空格，使用 **TRIM()** 可以去除首尾空格。

```
SELECT CONCAT(TRIM(col1), '(', TRIM(col2), ')') AS concat_col
FROM mytable;
```

十二、函数

各个 DBMS 的函数都是不相同的，因此不可移植，以下主要是 MySQL 的函数。

汇总

函 数	说 明
AVG()	返回某列的平均值
COUNT()	返回某列的行数

函 数	说 明
MAX()	返回某列的最大值
MIN()	返回某列的最小值
SUM()	返回某列值之和

AVG() 会忽略 NULL 行。

使用 DISTINCT 可以让汇总函数值汇总不同的值。

```
SELECT AVG(DISTINCT col1) AS avg_col
FROM mytable;
```

文本处理

函数	说明
LEFT()	左边的字符
RIGHT()	右边的字符
LOWER()	转换为小写字符
UPPER()	转换为大写字符
LTRIM()	去除左边的空格
RTRIM()	去除右边的空格
LENGTH()	长度
SOUNDEX()	转换为语音值

其中， **SOUNDEX()** 可以将一个字符串转换为描述其语音表示的字母数字模式。

```
SELECT *
FROM mytable
WHERE SOUNDEX(col1) = SOUNDEX('apple')
```

日期和时间处理

- 日期格式：YYYY-MM-DD
- 时间格式：HH:MM:SS

函 数	说 明
AddDate()	增加一个日期（天、周等）
AddTime()	增加一个时间（时、分等）
CurDate()	返回当前日期
CurTime()	返回当前时间
Date()	返回日期时间的日期部分
DateDiff()	计算两个日期之差
Date_Add()	高度灵活的日期运算函数
Date_Format()	返回一个格式化的日期或时间串
Day()	返回一个日期的天数部分
DayOfWeek()	对于一个日期，返回对应的星期几
Hour()	返回一个时间的小时部分
Minute()	返回一个时间的分钟部分
Month()	返回一个日期的月份部分
Now()	返回当前日期和时间
Second()	返回一个时间的秒部分
Time()	返回一个日期时间的时间部分
Year()	返回一个日期的年份部分

```
mysql> SELECT NOW();
```

```
2018-4-14 20:25:11
```

数值处理

函数	说明
SIN()	正弦
COS()	余弦
TAN()	正切
ABS()	绝对值
SQRT()	平方根
MOD()	余数
EXP()	指数
PI()	圆周率
RAND()	随机数

十三、分组

分组就是把具有相同的数据值的行放在同一组中。

可以对同一分组数据使用汇总函数进行处理，例如求分组数据的平均值等。

指定的分组字段除了能按该字段进行分组，也会自动按该字段进行排序。

```
SELECT col, COUNT(*) AS num
FROM mytable
GROUP BY col;
```

GROUP BY 自动按分组字段进行排序，ORDER BY 也可以按汇总字段来进行排序。

```
SELECT col, COUNT(*) AS num
FROM mytable
GROUP BY col
```

```
ORDER BY num;
```

WHERE 过滤行，HAVING 过滤分组，行过滤应当先于分组过滤。

```
SELECT col, COUNT(*) AS num
FROM mytable
WHERE col > 2
GROUP BY col
HAVING num >= 2;
```

分组规定：

- GROUP BY 子句出现在 WHERE 子句之后，ORDER BY 子句之前；
- 除了汇总字段外，SELECT 语句中的每一字段都必须在 GROUP BY 子句中给出；
- NULL 的行会单独分为一组；
- 大多数 SQL 实现不支持 GROUP BY 列具有可变长度的数据类型。

十四、子查询

子查询中只能返回一个字段的数据。

可以将子查询的结果作为 WHRER 语句的过滤条件：

```
SELECT *
FROM mytable1
WHERE col1 IN (SELECT col2
               FROM mytable2);
```

下面的语句可以检索出客户的订单数量，子查询语句会对第一个查询检索出的每个客户执行一次：

```
SELECT cust_name, (SELECT COUNT(*)
                   FROM Orders
                   WHERE Orders.cust_id = Customers.cust_id)
                   AS orders_num
FROM Customers
```

```
6.  ORDER BY cust_name;
```

十五、连接

连接用于连接多个表，使用 JOIN 关键字，并且条件语句使用 ON 而不是 WHERE。

连接可以替换子查询，并且比子查询的效率一般会更快。

可以用 AS 给列名、计算字段和表名取别名，给表名取别名是为了简化 SQL 语句以及连接相同表。

内连接

内连接又称等值连接，使用 INNER JOIN 关键字。

```
1.  SELECT A.value, B.value
2.  FROM tablea AS A INNER JOIN tableb AS B
3.  ON A.key = B.key;
```

可以不明确使用 INNER JOIN，而使用普通查询并在 WHERE 中将两个表中要连接的列用等值方法连接起来。

```
1.  SELECT A.value, B.value
2.  FROM tablea AS A, tableb AS B
3.  WHERE A.key = B.key;
```

在没有条件语句的情况下返回笛卡尔积。

自连接

自连接可以看成内连接的一种，只是连接的表是自身而已。

一张员工表，包含员工姓名和员工所属部门，要找出与 Jim 处在同一部门的所有员工姓名。

子查询版本

```
SELECT name
FROM employee
WHERE department = (
      SELECT department
      FROM employee
      WHERE name = "Jim");
```

自连接版本

```
SELECT e1.name
FROM employee AS e1 INNER JOIN employee AS e2
ON e1.department = e2.department
      AND e2.name = "Jim";
```

自然连接

自然连接是把同名列通过等值测试连接起来的，同名列可以有多个。

内连接和自然连接的区别：内连接提供连接的列，而自然连接自动连接所有同名列。

```
SELECT A.value, B.value
FROM tablea AS A NATURAL JOIN tableb AS B;
```

外连接

外连接保留了没有关联的那些行。分为左外连接，右外连接以及全外连接，左外连接就是保留左表没有关联的行。

检索所有顾客的订单信息，包括还没有订单信息的顾客。

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

customers 表：

cust_id	cust_name
1	a
2	b
3	c

orders 表：

order_id	cust_id
1	1
2	1
3	3
4	3

结果：

cust_id	cust_name	order_id
1	a	1
1	a	2
3	c	3
3	c	4
2	b	Null

十六、组合查询

使用 **UNION** 来组合两个查询，如果第一个查询返回 M 行，第二个查询返回 N 行，那么组合查询的结果一般为 M+N 行。

每个查询必须包含相同的列、表达式和聚集函数。

默认会去除相同行，如果需要保留相同行，使用 UNION ALL。

只能包含一个 ORDER BY 子句，并且必须位于语句的最后。

```
SELECT col
FROM mytable
WHERE col = 1
UNION
SELECT col
FROM mytable
WHERE col =2;
```

十七、视图

视图是虚拟的表，本身不包含数据，也就不能对其进行索引操作。

对视图的操作和对普通表的操作一样。

视图具有如下好处：

- 简化复杂的 SQL 操作，比如复杂的连接；
- 只使用实际表的一部分数据；
- 通过只给用户访问视图的权限，保证数据的安全性；
- 更改数据格式和表示。

```
CREATE VIEW myview AS
SELECT Concat(col1, col2) AS concat_col, col3*col4 AS compute_col
FROM mytable
WHERE col5 = val;
```

十八、存储过程

存储过程可以看成是对一系列 SQL 操作的批处理；

使用存储过程的好处：

- 代码封装，保证了一定的安全性；
- 代码复用；
- 由于是预先编译，因此具有很高的性能。

命令行中创建存储过程需要自定义分隔符，因为命令行是以；为结束符，而存储过程中也包含了分号，因此会错误把这部分分号当成是结束符，造成语法错误。

包含 in、out 和 inout 三种参数。

给变量赋值都需要用 select into 语句。

每次只能给一个变量赋值，不支持集合的操作。

```
delimiter //

create procedure myprocedure( out ret int )
    begin
        declare y int;
        select sum(col1)
        from mytable
        into y;
        select y*y into ret;
    end //

delimiter ;
```

```
call myprocedure(@ret);
select @ret;
```

十九、游标

在存储过程中使用游标可以对一个结果集进行移动遍历。

游标主要用于交互式应用，其中用户需要对数据集中的任意行进行浏览和修改。

使用游标的四个步骤：

1. 声明游标，这个过程没有实际检索出数据；
2. 打开游标；
3. 取出数据；
4. 关闭游标；

```
delimiter //
create procedure myprocedure(out ret int)
    begin
        declare done boolean default 0;

        declare mycursor cursor for
        select col1 from mytable;
        # 定义了一个 continue handler, 当 sqlstate '02000' 这个条件出现时,会执行 set done = 1
        declare continue handler for sqlstate '02000' set done = 1;

        open mycursor;

        repeat
            fetch mycursor into ret;
            select ret;
        until done end repeat;

        close mycursor;
    end //
 delimiter ;
```

二十、触发器

触发器会在某个表执行以下语句时而自动执行：DELETE、INSERT、UPDATE。

触发器必须指定在语句执行之前还是之后自动执行，之前执行使用 BEFORE 关键字，之后执行使用 AFTER 关键字。BEFORE 用于数据验证和净化，AFTER 用于审计跟踪，将修改记录到另外一张表中。

INSERT 触发器包含一个名为 NEW 的虚拟表。

```
CREATE TRIGGER mytrigger AFTER INSERT ON mytable
FOR EACH ROW SELECT NEW.col into @result;

SELECT @result; -- 获取结果
```

DELETE 触发器包含一个名为 OLD 的虚拟表，并且是只读的。

UPDATE 触发器包含一个名为 NEW 和一个名为 OLD 的虚拟表，其中 NEW 是可以被修改地，而 OLD 是只读的。

MySQL 不允许在触发器中使用 CALL 语句，也就是不能调用存储过程。

二十一、事务处理

基本术语：

- 事务（transaction）指一组 SQL 语句；
- 回退（rollback）指撤销指定 SQL 语句的过程；
- 提交（commit）指将未存储的 SQL 语句结果写入数据库表；
- 保留点（savepoint）指事务处理中设置的临时占位符（placeholder），你可以对它发布回退（与回退整个事务处理不同）。

不能回退 SELECT 语句，回退 SELECT 语句也没意义；也不能回退 CREATE 和 DROP 语句。

MySQL 的事务提交默认是隐式提交，每执行一条语句就把这条语句当成一个事务然后进行提交。当出现 START TRANSACTION 语句时，会关闭隐式提交；当 COMMIT 或 ROLLBACK 语句执行后，事务会自动关闭，重新恢复隐式提交。

通过设置 autocommit 为 0 可以取消自动提交；autocommit 标记是针对每个连接而不是针对服务器的。

如果没有设置保留点，ROLLBACK 会回退到 START TRANSACTION 语句处；如果设置了保留点，并且在 ROLLBACK 中指定该保留点，则会回退到该保留点。

```
START TRANSACTION
// ...
```

```
SAVEPOINT delete1
// ...
ROLLBACK TO delete1
// ...
COMMIT
```

二十二、字符集

基本术语：

- 字符集为字母和符号的集合；
- 编码为某个字符集成员的内部表示；
- 校对字符指定如何比较，主要用于排序和分组。

除了给表指定字符集和校对外，也可以给列指定：

```
CREATE TABLE mytable
(col VARCHAR(10) CHARACTER SET latin COLLATE latin1_general_ci )
DEFAULT CHARACTER SET hebrew COLLATE hebrew_general_ci;
```

可以在排序、分组时指定校对：

```
SELECT *
FROM mytable
ORDER BY col COLLATE latin1_general_ci;
```

二十三、权限管理

MySQL 的账户信息保存在 mysql 这个数据库中。

```
USE mysql;
SELECT user FROM user;
```

创建账户

```
CREATE USER myuser IDENTIFIED BY 'mypassword';
```

新创建的账户没有任何权限。

修改账户名

```
RENAME myuser TO newuser;
```

删除账户

```
DROP USER myuser;
```

查看权限

```
SHOW GRANTS FOR myuser;
```

授予权限

```
GRANT SELECT, INSERT ON mydatabase.* TO myuser;
```

账户用 username@host 的形式定义，username@% 使用的是默认主机名。

删除权限

```
REVOKE SELECT, INSERT ON mydatabase.* FROM myuser;
```

GRANT 和 REVOKE 可在几个层次上控制访问权限：

- 整个服务器，使用 GRANT ALL 和 REVOKE ALL；
- 整个数据库，使用 ON database.*；
- 特定的表，使用 ON database.table；
- 特定的列；
- 特定的存储过程。

更改密码

必须使用 Password() 函数

```
SET PASSWROD FOR myuser = Password('new_password');
```

参考资料

- BenForta. SQL 必知必会 [M]. 人民邮电出版社, 2013.

MySQL

https://github.com/CyC2018/Interview-Notebook

PDF制作github: https://github.com/sjsdfg/Interview-Notebook-PDF

一、存储引擎

InnoDB

InnoDB 是 MySQL 默认的事务型存储引擎，只有在需要 InnoDB 不支持的特性时，才考虑使用其它存储引擎。

实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+ 间隙锁（next-key locking）防止幻影读。

主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。

内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其它存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

MyISAM

MyISAM 设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用 MyISAM。

MyISAM 提供了大量的特性，包括压缩表、空间数据索引等。

不支持事务。

不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在表有读取操作的同时，也可以往表中插入新的记录，这被称为并发插入（CONCURRENT INSERT）。

可以手工或者自动执行检查和修复操作，但是和事务恢复以及崩溃恢复不同，可能导致一些数据丢失，而且修复操作是非常慢的。

如果指定了 DELAY_KEY_WRITE 选项，在每次修改执行完成时，不会立即将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区，只有在清理键缓冲区或者关闭表的时候才会将对应的索引块写入磁盘。这种方式可以极大的提升写入性能，但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作。

比较

- 事务：InnoDB 是事务型的，可以使用 Commit 和 Rollback 语句。
- 并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。
- 外键：InnoDB 支持外键。
- 备份：InnoDB 支持在线热备份。
- 崩溃恢复：MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。
- 其它特性：MyISAM 支持压缩表和空间数据索引。

二、数据类型

整型

TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT 分别使用 8, 16, 24, 32, 64 位存储空间，一般情况下越小的列越好。

INT(11) 中的数字只是规定了交互工具显示字符的个数，对于存储和计算来说是没有意义的。

浮点数

FLOAT 和 DOUBLE 为浮点类型，DECIMAL 为高精度小数类型。CPU 原生支持浮点运算，但是不支持 DECIMAl 类型的计算，因此 DECIMAL 的计算比浮点类型需要更高的代价。

FLOAT、DOUBLE 和 DECIMAL 都可以指定列宽，例如 DECIMAL(18, 9) 表示总共 18 位，取 9 位存储小数部分，剩下 9 位存储整数部分。

字符串

主要有 CHAR 和 VARCHAR 两种类型，一种是定长的，一种是变长的。

VARCHAR 这种变长类型能够节省空间，因为只需要存储必要的内容。但是在执行 UPDATE 时可能会使行变得比原来长，当超出一个页所能容纳的大小时，就要执行额外的操作。MyISAM 会将行拆成不同的片段存储，而 InnoDB 则需要分裂页来使行放进页内。

VARCHAR 会保留字符串末尾的空格，而 CHAR 会删除。

时间和日期

MySQL 提供了两种相似的日期时间类型：DATATIME 和 TIMESTAMP。

1. DATATIME

能够保存从 1001 年到 9999 年的日期和时间，精度为秒，使用 8 字节的存储空间。

它与时区无关。

默认情况下，MySQL 以一种可排序的、无歧义的格式显示 DATATIME 值，例如“2008-01-16 22:37:08”，这是 ANSI 标准定义的日期和时间表示方法。

2. TIMESTAMP

和 UNIX 时间戳相同，保存从 1970 年 1 月 1 日午夜（格林威治时间）以来的秒数，使用 4 个字节，只能表示从 1970 年 到 2038 年。

它和时区有关，也就是说一个时间戳在不同的时区所代表的具体时间是不同的。

MySQL 提供了 FROM_UNIXTIME() 函数把 UNIX 时间戳转换为日期，并提供了 UNIX_TIMESTAMP() 函数把日期转换为 UNIX 时间戳。

默认情况下，如果插入时没有指定 TIMESTAMP 列的值，会将这个值设置为当前时间。

应该尽量使用 TIMESTAMP，因为它比 DATETIME 空间效率更高。

三、索引

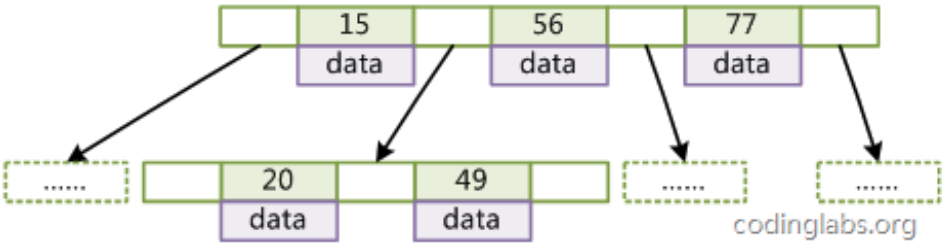
索引能够轻易将查询性能提升几个数量级。

对于非常小的表、大部分情况下简单的全表扫描比建立索引更高效。对于中到大型的表，索引就非常有效。但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

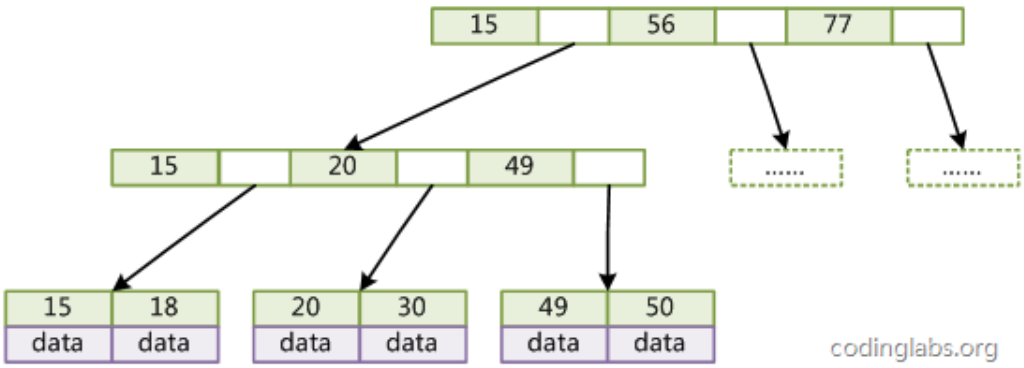
B Tree 原理

1. B-Tree

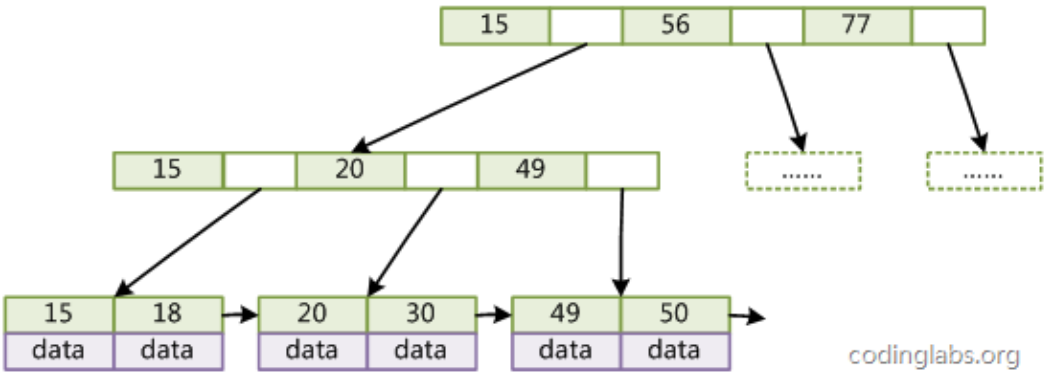


ii+1ii+1

2. B+Tree



3. 顺序访问指针



4. 优势

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用 B Tree 作为索引结构，主要有以下两个原因：

（一）更少的检索次数

平衡树检索数据的时间复杂度等于树高 h，而树高大致为 $O(h)=O(\log_d N)$，其中 d 为每个节点的出度。

红黑树的出度为 2，而 B Tree 的出度一般都非常大。红黑树的树高 h 很明显比 B Tree 大非常多，因此检索的次数也就更多。

B+Tree 相比于 B-Tree 更适合外存索引，因为 B+Tree 内节点去掉了 data 域，因此可以拥有更大的出度，检索效率会更高。

（二）利用计算机预读特性

为了减少磁盘 I/O，磁盘往往不是严格按需读取，而是每次都会预读。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的旋转时间，因此速度会非常快。

操作系统一般将内存和磁盘分割成固态大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点，并且可以利用预读特性，相邻的节点也能够被预先载入。

更多内容请参考：MySQL 索引背后的数据结构及算法原理

索引分类

1. B+Tree 索引

B+Tree 索引是大多数 MySQL 存储引擎的默认索引类型。

因为不再需要进行全表扫描，只需要对树进行搜索即可，因此查找速度快很多。除了用于查找，还可以用于排序和分组。

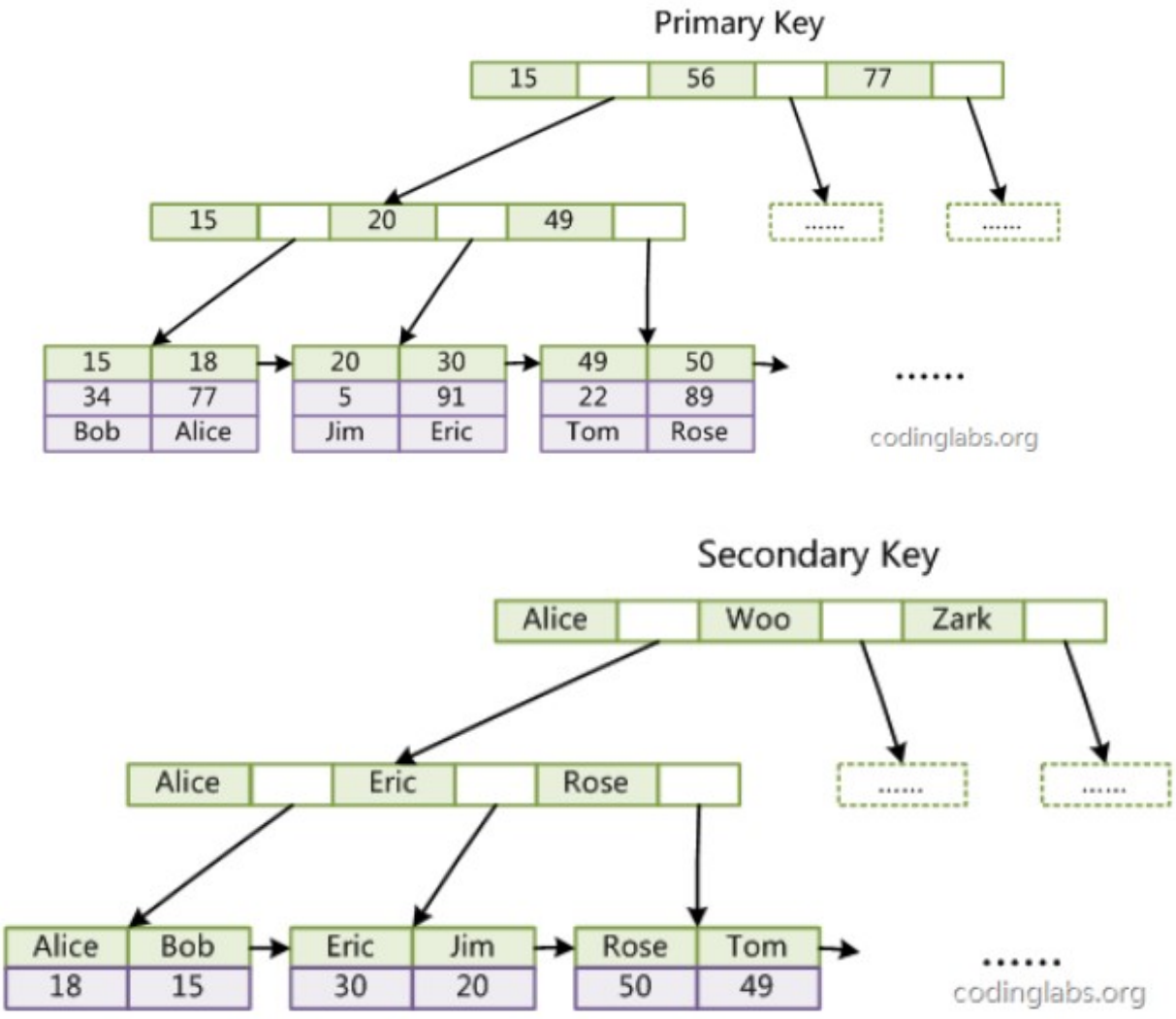
可以指定多个列作为索引列，多个索引列共同组成键。

B+Tree 索引适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。

如果不是按照索引列的顺序进行查找，则无法使用索引。

InnoDB 的 B+Tree 索引分为主索引和辅助索引。

主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。



2. 哈希索引

InnoDB 引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

哈希索引能以 O(1) 时间进行查找，但是失去了有序性，它具有以下限制：

- 无法用于排序与分组；
- 只支持精确查找，无法用于部分查找和范围查找；

3. 全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引一般使用倒排索引实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

4. 空间数据索引（R-Tree）

MyISAM 存储引擎支持空间数据索引，可以用于地理数据存储。空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用 GIS 相关的函数来维护数据。

索引的优点

- 大大减少了服务器需要扫描的数据行数。
- 帮助服务器避免进行排序和创建临时表（B+Tree 索引是有序的，可以用来做 ORDER BY 和 GROUP BY 操作）；

- 将随机 I/O 变为顺序 I/O（B+Tree 索引是有序的，也就将相邻的数据都存储在一起）。

索引优化

1. 独立的列

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。

例如下面的查询不能使用 actor_id 列的索引：

```
1.  SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

2. 多列索引

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。例如下面的语句中，最好把 actor_id 和 film_id 设置为多列索引。

```
1.  SELECT film_id, actor_ id FROM sakila.film_actor
2.  WHERE actor_id = 1 AND film_id = 1;
```

3. 索引列的顺序

让选择性最强的索引列放在前面，索引的选择性是指：不重复的索引值和记录总数的比值。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，查询效率也越高。

例如下面显示的结果中 customer_id 的选择性比 staff_id 更高，因此最好把 customer_id 列放在多列索引的前面。

```
1.  SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
2.  COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
3.  COUNT(*)
4.  FROM payment;
```

```
1.     staff_id_selectivity: 0.0001
```

```
customer_id_selectivity: 0.0373
                COUNT(*): 16049
```

4. 前缀索引

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。

对于前缀长度的选取需要根据索引选择性来确定。

5. 覆盖索引

索引包含所有需要查询的字段的值。

具有以下优点：

- 因为索引条目通常远小于数据行的大小，所以若只读取索引，能大大减少数据访问量。
- 一些存储引擎（例如 MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
- 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引。

四、查询性能优化

使用 Explain 进行分析

Explain 用来分析 SELECT 查询语句，开发人员可以通过分析 Explain 结果来优化查询语句。

比较重要的字段有：

- select_type : 查询类型，有简单查询、联合查询、子查询等
- key : 使用的索引
- rows : 扫描的行数

更多内容请参考：MySQL 性能优化神器 Explain 使用分析

优化数据访问

1. 减少请求的数据量

（一）只返回必要的列

最好不要使用 SELECT * 语句。

（二）只返回必要的行

使用 WHERE 语句进行查询过滤，有时候也需要使用 LIMIT 语句来限制返回的数据。

（三）缓存重复查询的数据

使用缓存可以避免在数据库中进行查询，特别要查询的数据经常被重复查询，缓存可以带来的查询性能提升将会是非常明显的。

2. 减少服务器端扫描的行数

最有效的方式是使用索引来覆盖查询。

重构查询方式

1. 切分大查询

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

```
DELEFT FROM messages WHERE create < DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

```
rows_affected = 0
do {
    rows_affected = do_query(
    "DELETE FROM messages WHERE create  < DATE_SUB(NOW(), INTERVAL 3 MONTH) LIMIT 10000")
```

```
} while rows_affected > 0
```

2. 分解大连接查询

将一个大连接查询（JOIN）分解成对每一个表进行一次单表查询，然后将结果在应用程序中进行关联，这样做的好处有：

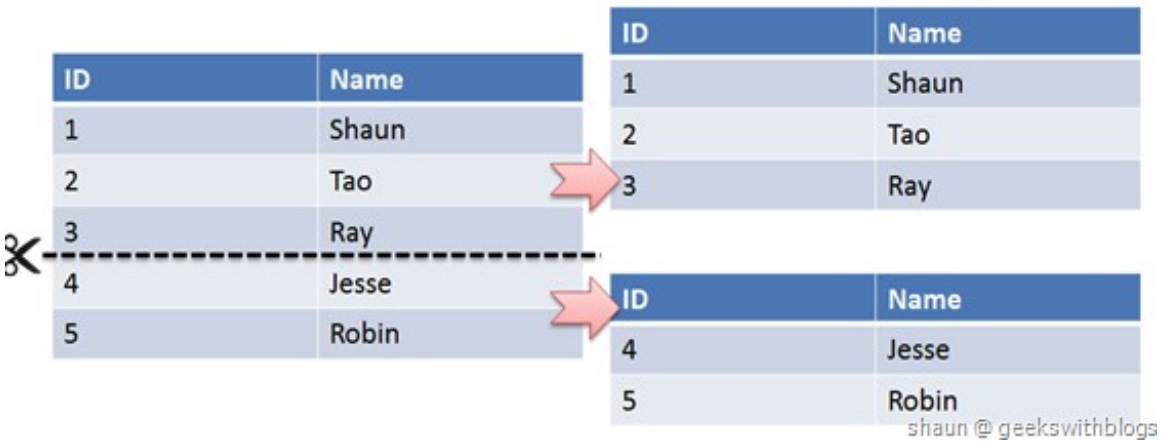
- 让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用。
- 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。
- 减少锁竞争；
- 在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可扩展。
- 查询本身效率也可能会有所提升。例如下面的例子中，使用 IN() 代替连接查询，可以让 MySQL 按照 ID 顺序进行查询，这可能比随机的连接要更高效。

```
SELECT * FROM tab
JOIN tag_post ON tag_post.tag_id=tag.id
JOIN post ON tag_post.post_id=post.id
WHERE tag.tag='mysql';
```

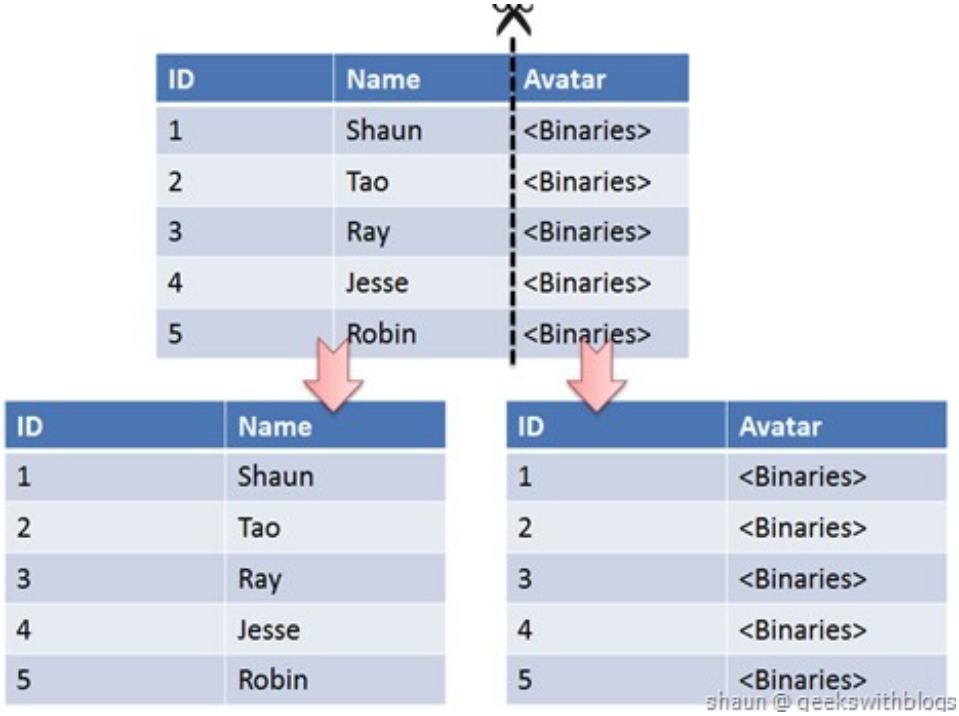
```
SELECT * FROM tag WHERE tag='mysql';
SELECT * FROM tag_post WHERE tag_id=1234;
SELECT * FROM post WHERE post.id IN (123,456,567,9098,8904);
```

五、切分

水平切分



垂直切分



Sharding 策略

- 哈希取模：hash(key) % NUM_DB
- 范围：可以是 ID 范围也可以是时间范围
- 映射表：使用单独的一个数据库来存储映射关系

Sharding 存在的问题及解决方案

1. 事务问题

使用分布式事务来解决，比如 XA 接口。

2. JOIN

可以将原来的 JOIN 查询分解成多个单表查询，然后在用户程序中进行 JOIN。

3. ID 唯一性

- 使用全局唯一 ID：GUID。
- 为每个分片指定一个 ID 范围。
- 分布式 ID 生成器 (如 Twitter 的 Snowflake 算法)。

更多内容请参考：

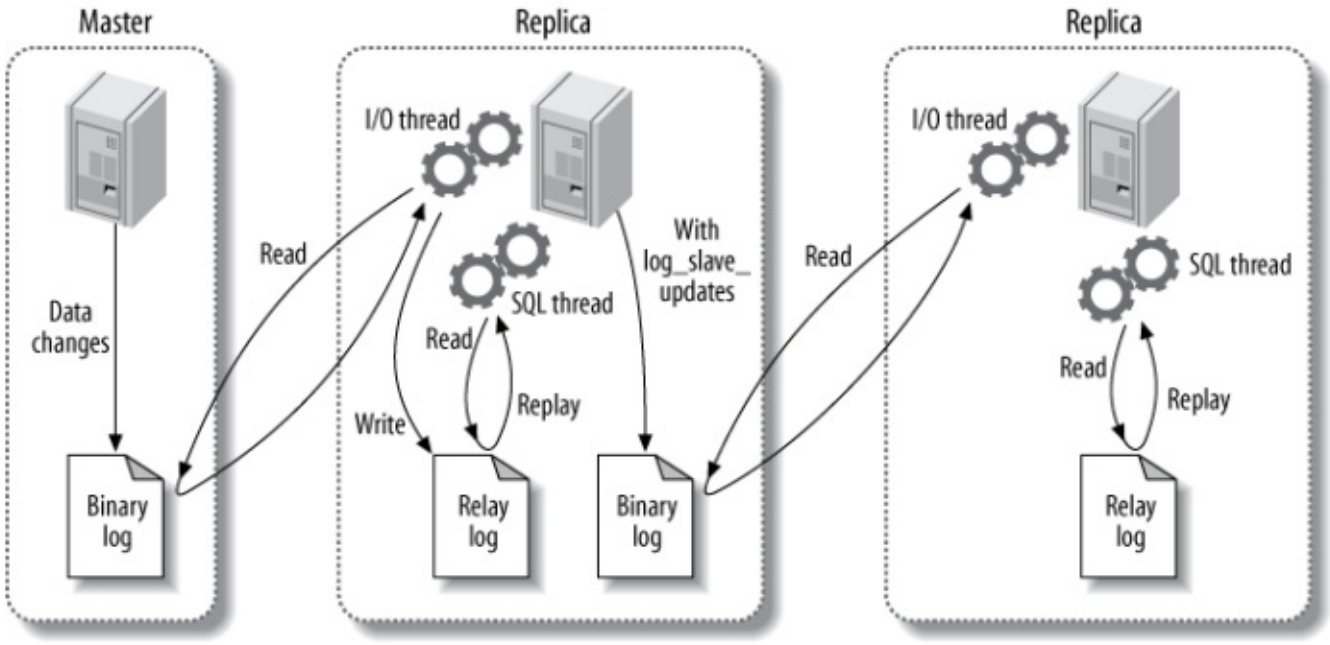
- How Sharding Works
- 大众点评订单系统分库分表实践

六、复制

主从复制

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

- **binlog 线程** ：负责将主服务器上的数据更改写入二进制文件（binlog）中。
- **I/O 线程** ：负责从主服务器上读取二进制日志文件，并写入从服务器的中继日志中。
- **SQL 线程** ：负责读取中继日志并重放其中的 SQL 语句。



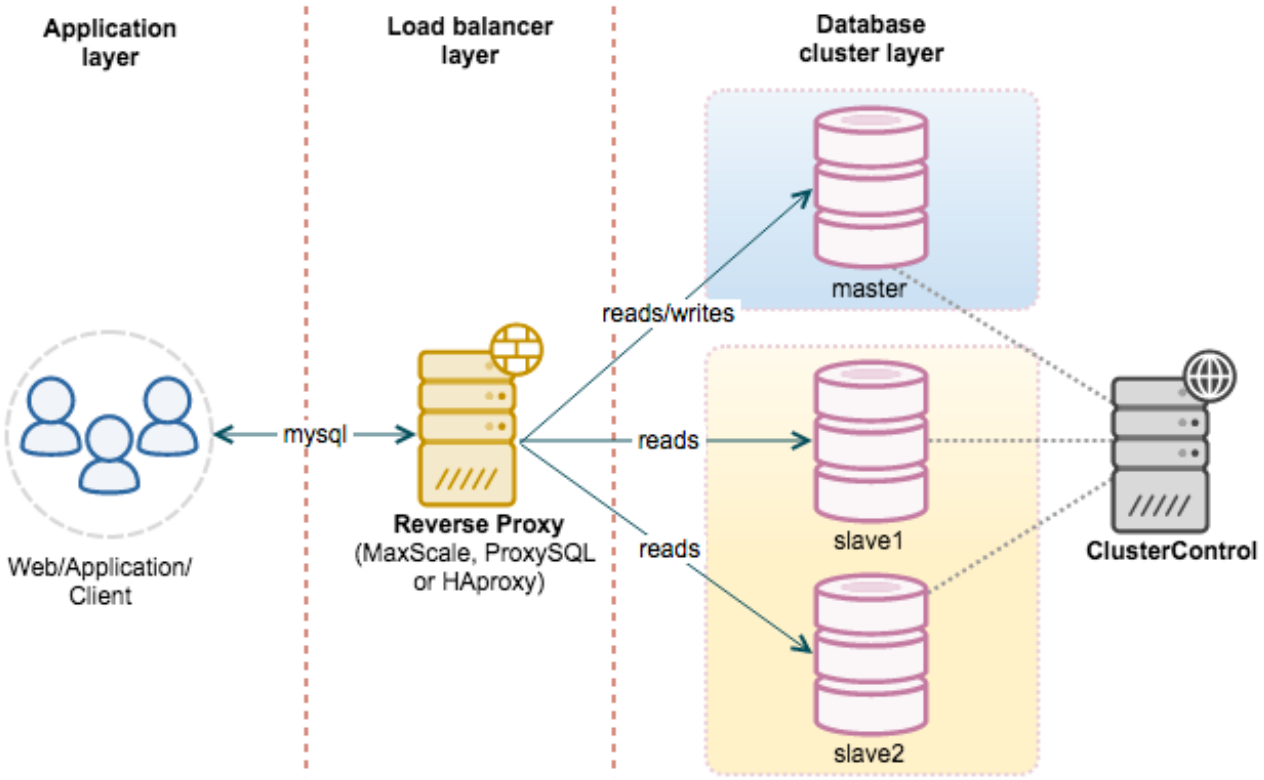
读写分离

主服务器用来处理写操作以及实时性要求比较高的读操作，而从服务器用来处理读操作。

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。

MySQL 读写分离能提高性能的原因在于：

- 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
- 从服务器可以配置 MyISAM 引擎，提升查询性能以及节约系统开销；
- 增加冗余，提高可用性。



参考资料

- BaronScbwartz, PeterZaitsev, VadimTkacbenko, 等. 高性能 MySQL[M]. 电子工业出版社, 2013.
- 姜承尧. MySQL 技术内幕: InnoDB 存储引擎 [M]. 机械工业出版社, 2011.
- 20+ 条 MySQL 性能优化的最佳经验
- 服务端指南 数据存储篇 | MySQL（09） 分库与分表带来的分布式困境与应对之策
- How to create unique row ID in sharded databases?
- SQL Azure Federation – Introduction

Redis

https://github.com/CyC2018/Interview-Notebook

PDF制作github: https://github.com/sjsdfg/Interview-Notebook-PDF

一、概述

Redis 是速度非常快的非关系型（NoSQL）内存键值数据库，可以存储键和五种不同类型的值之间的映射。

键的类型只能为字符串，值支持的五种类型数据类型为：字符串、列表、集合、有序集合、散列表。

Redis 支持很多特性，例如将内存中的数据持久化到硬盘中，使用复制来扩展读性能，使用分片来扩展写性能。

二、数据类型

数据类型	可以存储的值	操作
STRING	字符串、整数或者浮点数	对整个字符串或者字符串的其中一部分执行操作 对整数和浮点数执行自增或者自减操作
LIST	列表	从两端压入或者弹出元素 读取单个或者多个元素 进行修剪，只保留一个范围内的元素
SET	无序集合	添加、获取、移除单个元素 检查一个元素是否存在于集合中 计算交集、并集、差集 从集合里面随机获取元素
HASH	包含键值对的无序散列表	添加、获取、移除单个键值对 获取所有键值对 检查某个键是否存在
ZSET	有序集合	添加、获取、删除元素 根据分值范围或者成员来获取元素 计算一个键的排名

> What Redis data structures look like

STRING

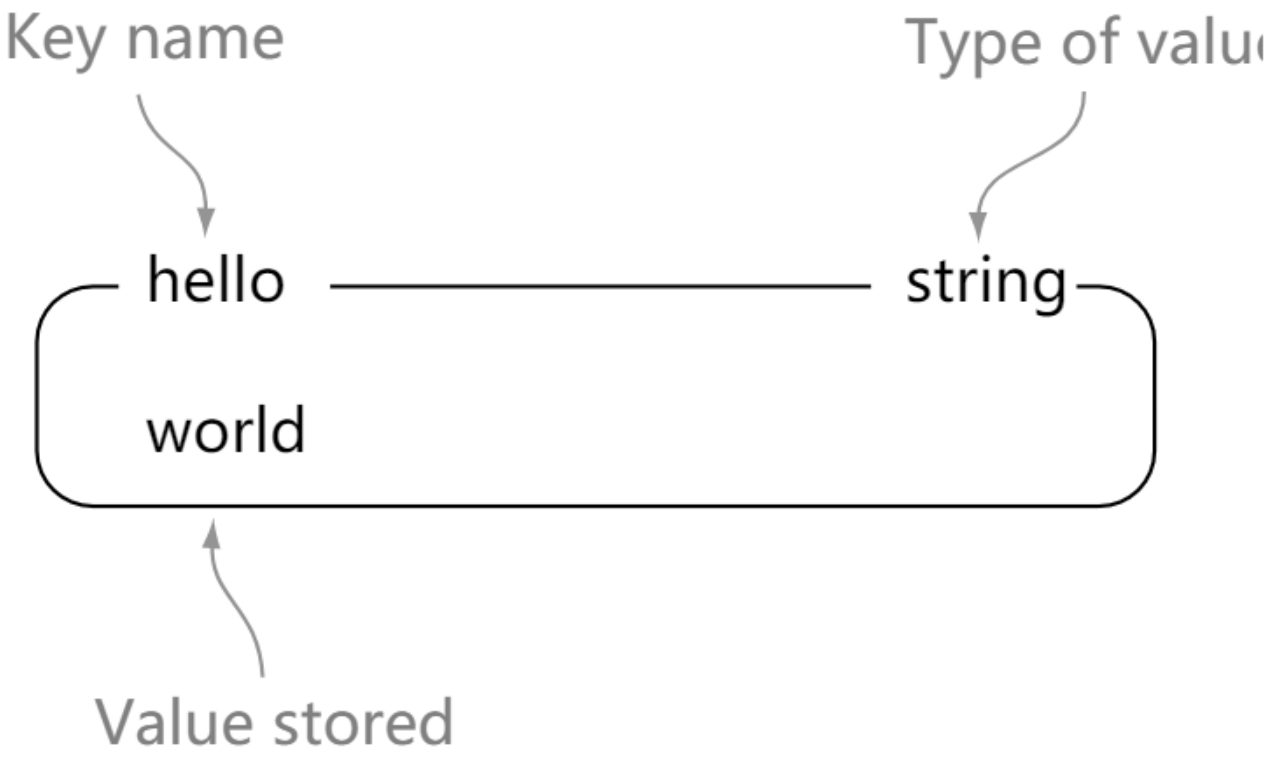


Figure 1.1 An example of a STRING, world, stored under a key, hello

```
> set hello world
OK
> get hello
"world"
> del hello
(integer) 1
> get hello
(nil)
```

LIST

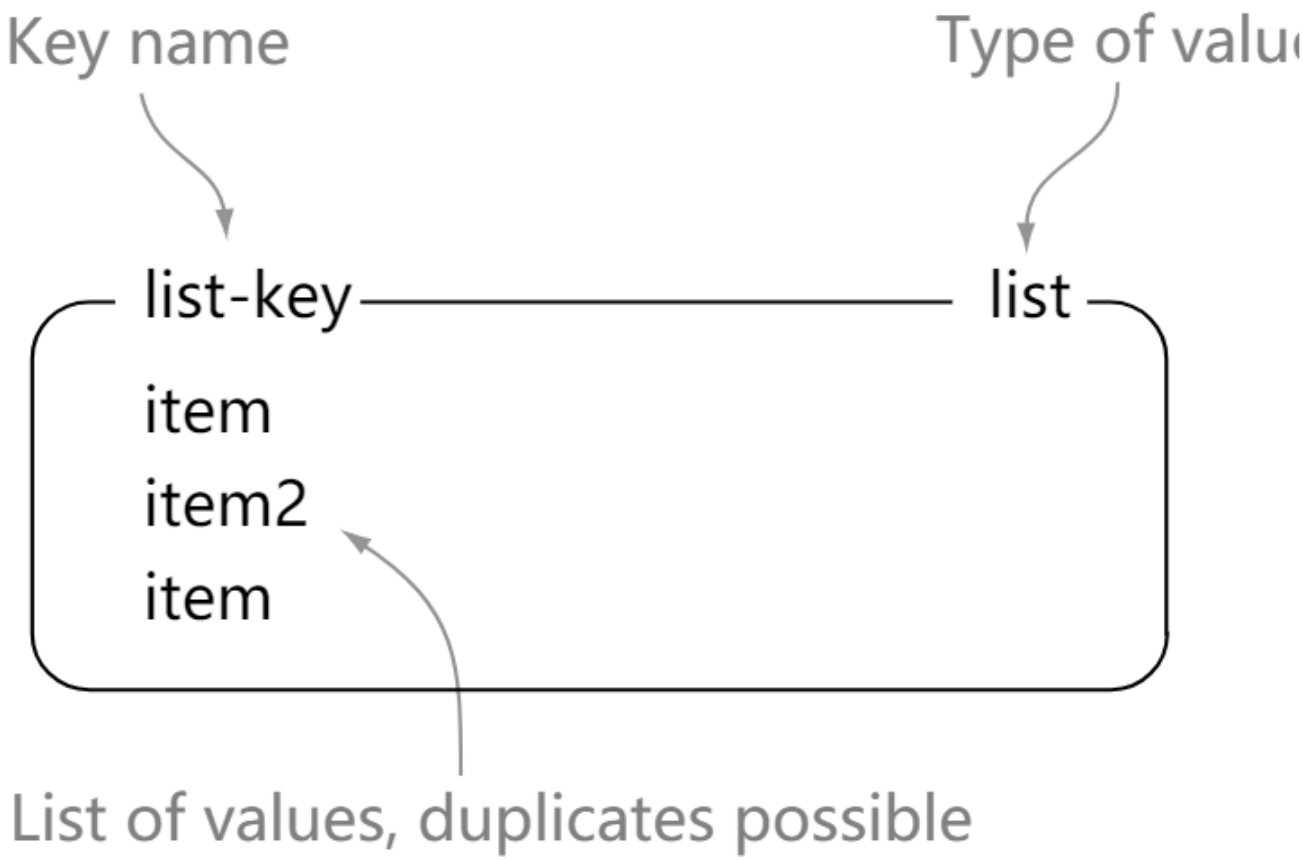


Figure 1.2 An example of a LIST with three items under the key, list-key. Note that item can be in the list more than once.

```
> rpush list-key item
(integer) 1
> rpush list-key item2
(integer) 2
> rpush list-key item
(integer) 3

> lrange list-key 0 -1
1) "item"
2) "item2"
3) "item"

> lindex list-key 1
"item2"

> lpop list-key
"item"

> lrange list-key 0 -1
```

```
1) "item2"
2) "item"
```

SET

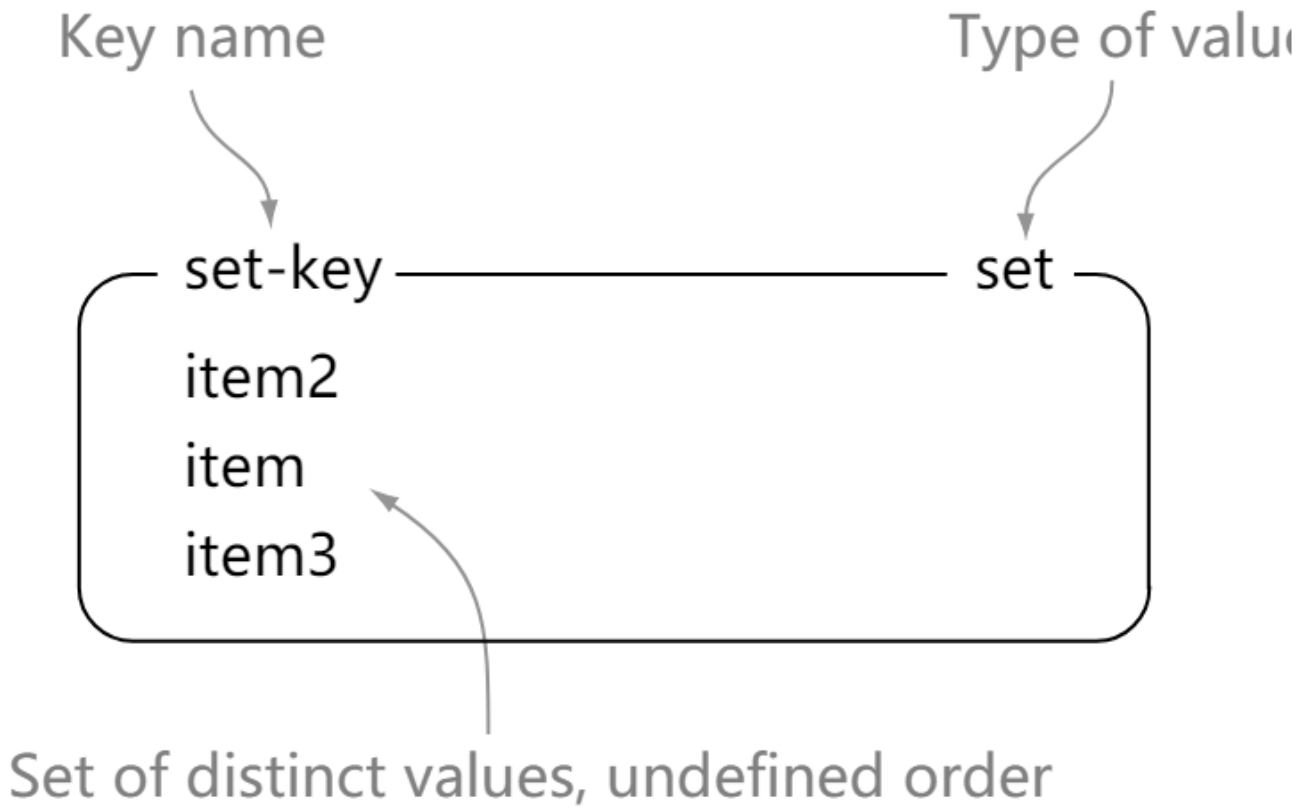


Figure 1.3 An example of a SET with three items under the key, set-key

```
> sadd set-key item
(integer) 1
> sadd set-key item2
(integer) 1
> sadd set-key item3
(integer) 1
> sadd set-key item
(integer) 0

> smembers set-key
1) "item"
2) "item2"
3) "item3"
```

```
> sismember set-key item4
(integer) 0
> sismember set-key item
(integer) 1

> srem set-key item2
(integer) 1
> srem set-key item2
(integer) 0

> smembers set-key
1) "item"
2) "item3"
```

HASH

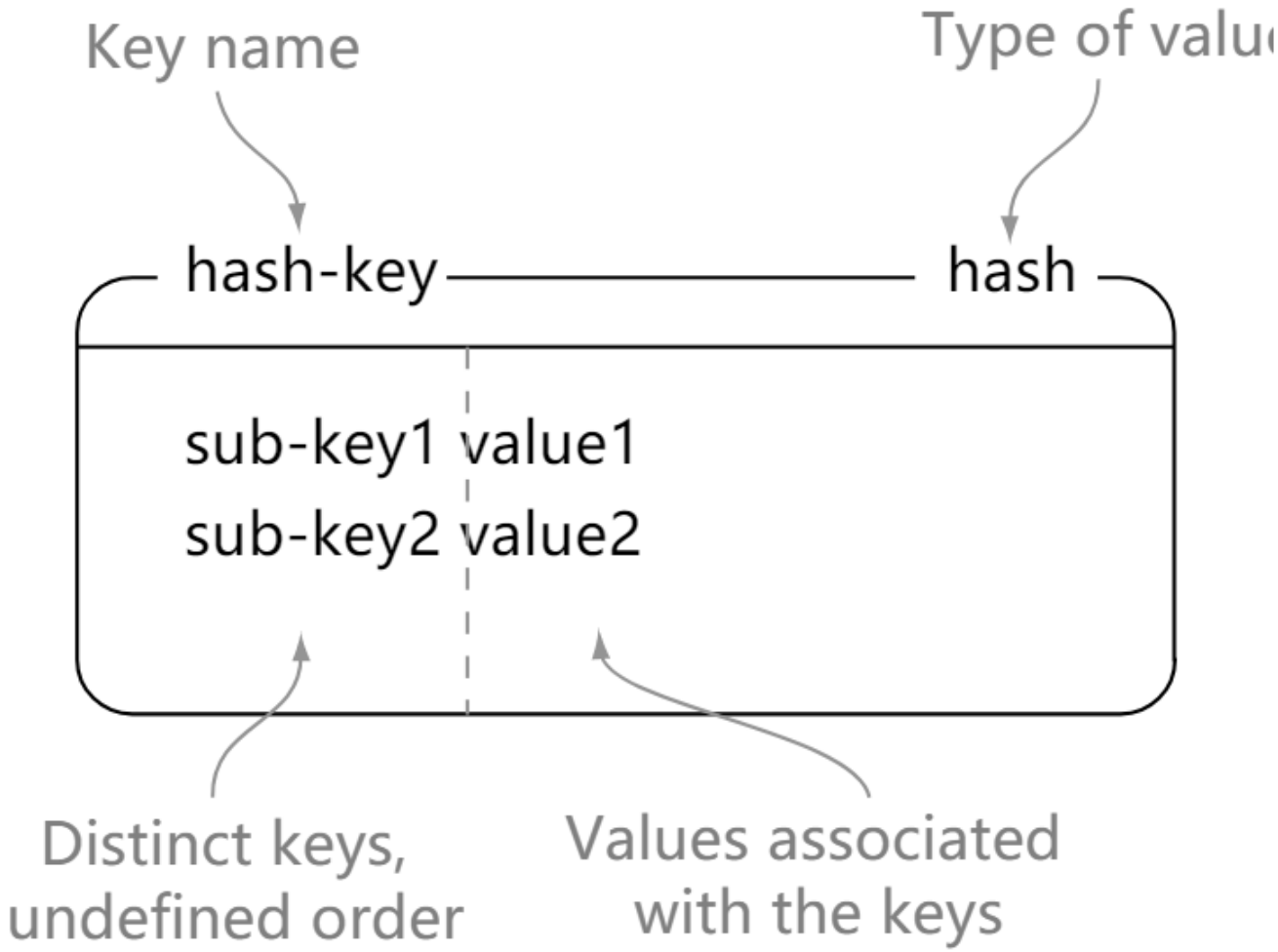


Figure 1.4 An example of a `HASH` with two keys/values under the key `hash-key`

```
> hset hash-key sub-key1 value1
(integer) 1
> hset hash-key sub-key2 value2
(integer) 1
> hset hash-key sub-key1 value1
(integer) 0

> hgetall hash-key
1) "sub-key1"
2) "value1"
3) "sub-key2"
4) "value2"

> hdel hash-key sub-key2
(integer) 1
> hdel hash-key sub-key2
(integer) 0
```

```
> hget hash-key sub-key1
"value1"

> hgetall hash-key
1) "sub-key1"
2) "value1"
```

ZSET

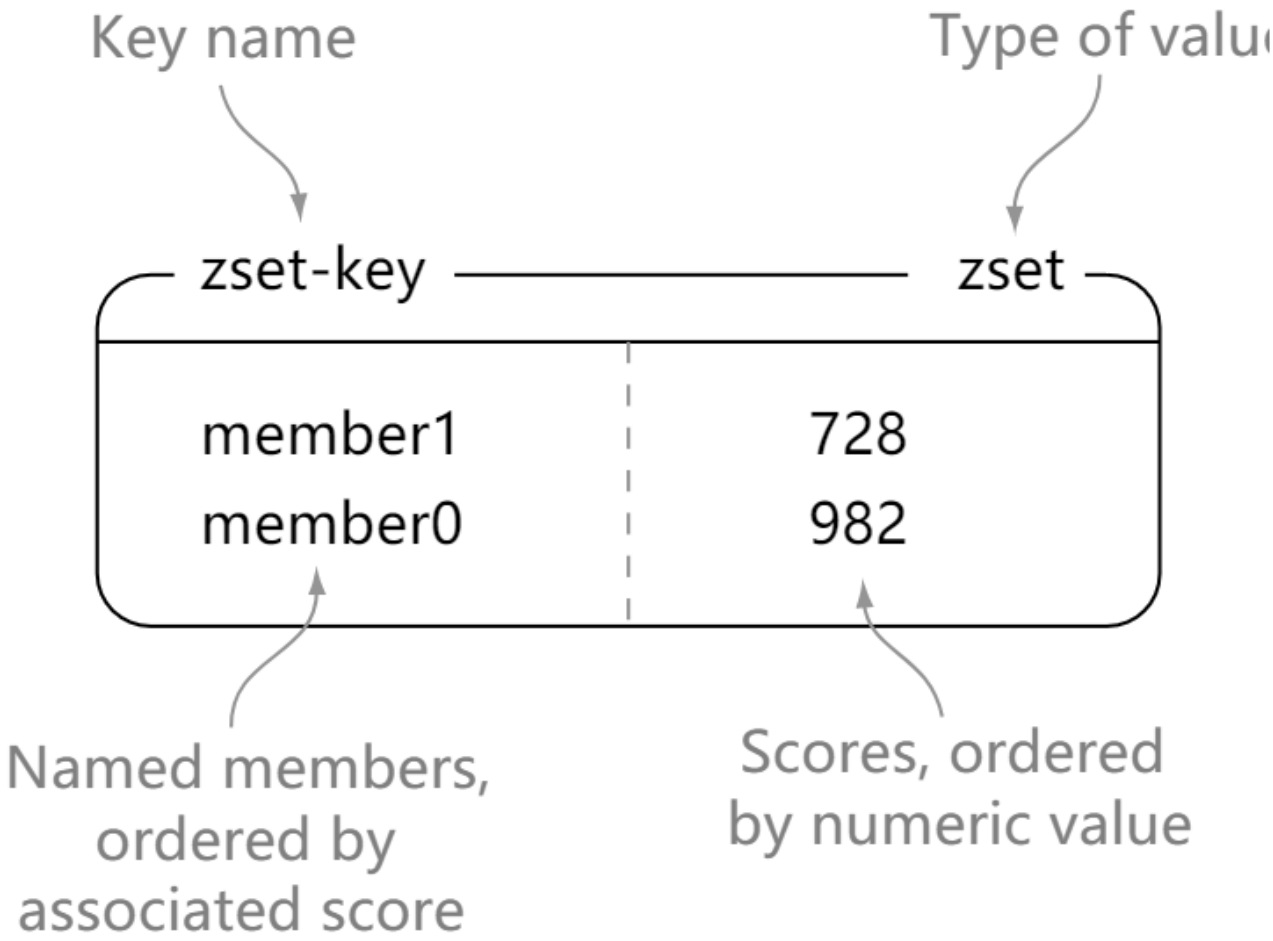


Figure 1.5 An example of a ZSET with two members/scores under the key zset-key

```
> zadd zset-key 728 member1
(integer) 1
> zadd zset-key 982 member0
(integer) 1
> zadd zset-key 982 member0
```

```
(integer) 0

> zrange zset-key 0 -1 withscores
1) "member1"
2) "728"
3) "member0"
4) "982"

> zrangebyscore zset-key 0 800 withscores
1) "member1"
2) "728"

> zrem zset-key member1
(integer) 1
> zrem zset-key member1
(integer) 0

> zrange zset-key 0 -1 withscores
1) "member0"
2) "982"
```

三、数据结构

字典

以下是 Redis 字典的主要数据结构，从上往下分析，一个 dict 有两个 dictht，一个 dictht 有一个 dictEntry 数组，每个 dictEntry 有 next 指针因此是一个链表结构。从上面的分析可以看出 Redis 的字典是一个基于拉链法解决冲突的哈希表结构。

```
typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */
    unsigned long iterators; /* number of iterators currently running
*/
} dict;
```

```
/* This is our hash table structure. Every dictionary has two of this as we
 * implement incremental rehashing, for the old to the new table. */
typedef struct dictht {
    dictEntry **table;
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
} dictht;
```

```
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next;
} dictEntry;
```

哈希表需要具备扩容能力，在扩容时就需要对每个键值对进行 rehash。dict 有两个 dictht，在 rehash 的时候会将一个 dictht 上的键值对重新插入另一个 dictht 上面，完成之后释放空间并交换两个 dictht 的角色。

rehash 操作不是一次性完成，而是采用渐进方式，这是为了避免一次性执行过多的 rehash 操作给服务器带来过大的负担。

渐进式 rehash 通过记录 dict 的 rehashidx 完成，它从 0 开始然后每执行一次 rehash 都会递增。例如在一次 rehash 中，要把 dict[0] rehash 到 dict[1]，这一次会把 dict[0] 上 table[rehashidx] 的键值对 rehash 到 dict[1] 上，dict[0] 的 table[rehashidx] 指向 null，并令 rehashidx++。

在 rehash 期间，每次对字典执行添加、删除、查找或者更新操作时，都会执行一次渐进式 rehash。

采用渐进式 rehash 会导致字典中的数据分散在两个 dictht 上，因此对字典的操作也需要到对应的 dictht 去执行。

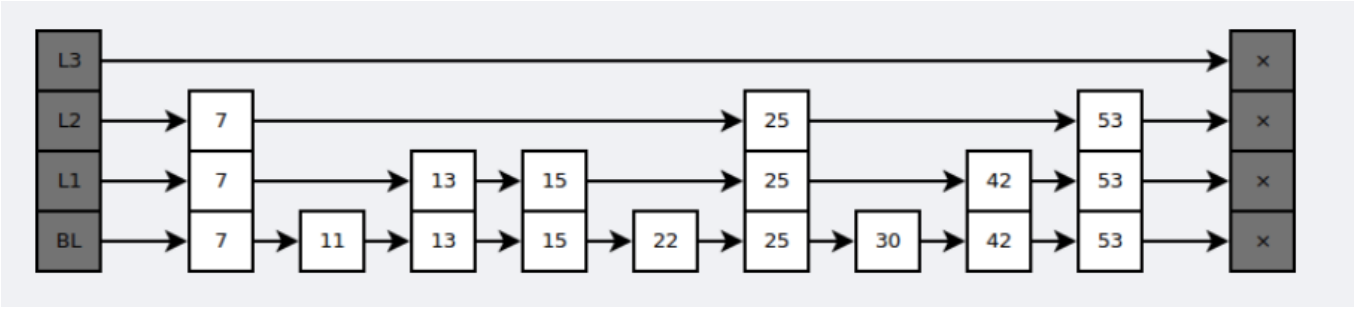
```c
/* Performs N steps of incremental rehashing. Returns 1 if there are still
 * keys to move from the old to the new hash table, otherwise 0 is returned.
 *
 * Note that a rehashing step consists in moving a bucket (that may have more
 * than one key as we use chaining) from the old to the new hash table, however
 * since part of the hash table may be composed of empty spaces, it is not
 * guaranteed that this function will rehash even a single bucket, since it
 * will visit at max N*10 empty buckets in total, otherwise the amount of
 * work it does would be unbound and the function may block for a long
 * time. */
int dictRehash(dict *d, int n) {
    int empty_visits = n * 10; /* Max number of empty buckets to visit. */
    if (!dictIsRehashing(d)) return 0;

    while (n-- && d->ht[0].used != 0) {
        dictEntry *de, *nextde;

        /* Note that rehashidx can't overflow as we are sure there are more
         * elements because ht[0].used != 0 */
        assert(d->ht[0].size > (unsigned long) d->rehashidx);
        while (d->ht[0].table[d->rehashidx] == NULL) {
            d->rehashidx++;
            if (--empty_visits == 0) return 1;
        }
        de = d->ht[0].table[d->rehashidx];
        /* Move all the keys in this bucket from the old to the new hash HT */
        while (de) {
            uint64_t h;

            nextde = de->next;
            /* Get the index in the new hash table */
            h = dictHashKey(d, de->key) & d->ht[1].sizemask;
            de->next = d->ht[1].table[h];
            d->ht[1].table[h] = de;
            d->ht[0].used--;
```

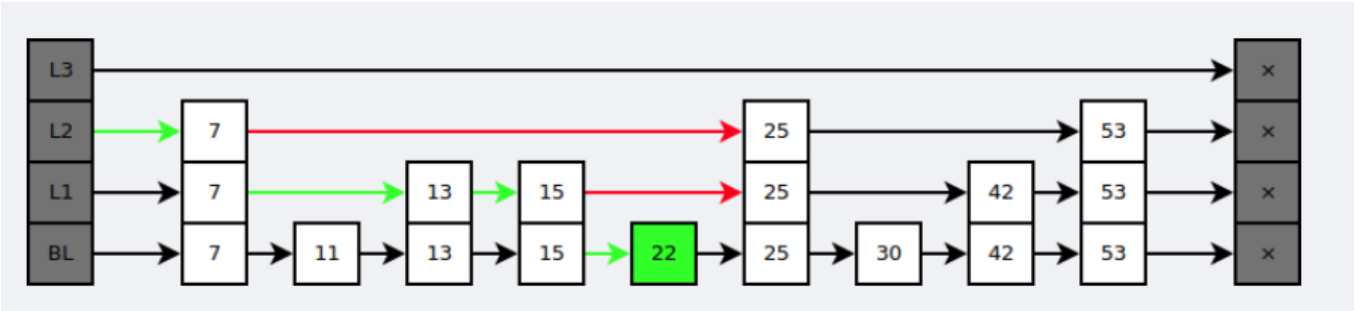
```
            d->ht[1].used++;
            de = nextde;
        }
        d->ht[0].table[d->rehashidx] = NULL;
        d->rehashidx++;
    }

    /* Check if we already rehashed the whole table... */
    if (d->ht[0].used == 0) {
        zfree(d->ht[0].table);
        d->ht[0] = d->ht[1];
        _dictReset(&d->ht[1]);
        d->rehashidx = -1;
        return 0;
    }

    /* More to rehash... */
    return 1;
}
```

跳跃表

是有序集合的底层实现之一。

跳跃表是基于多指针有序链表实现的，可以看成多个有序链表。





四、使用场景

计数器

可以对 String 进行自增自减运算，从而实现计数器功能。

例如对于网站访问量，如果使用 MySQL 数据库进行存储，那么每访问一次网站就要对磁盘进行读写操作。而对 Redis 这种内存型数据库的读写性能非常高，很适合存储这种频繁读写的计数量。

缓存

将热点数据放到内存中，设置内存的最大使用量以及淘汰策略来保证缓存的命中率。

查找表

例如 DNS 记录就很适合使用 Redis 进行存储。

查找表和缓存类似，也是利用了 Redis 快速的查找特性。但是查找表的内容不能失效，而缓存的内容可以失效。

消息队列

List 是一个双向链表，可以通过 lpop 和 lpush 写入和读取消息。

不过最好使用 Kafka、RabbitMQ 等消息中间件。

会话缓存

在分布式场景下具有多个应用服务器，可以使用 Redis 来统一存储这些应用服务器的会话信息，使得某个应用服务器宕机时不会丢失会话信息，从而保证高可用。

分布式锁实现

在分布式场景下，无法使用单机环境下的锁实现。当多个节点上的进程都需要获取同一个锁时，就需要使用分布式锁来进行同步。

除了可以使用 Redis 自带的 SETNX 命令实现分布式锁之外，还可以使用官方提供的 RedLock 分布式锁实现。

其它

Set 可以实现交集、并集等操作，例如共同好友功能。

ZSet 可以实现有序性操作，例如排行榜功能。

五、Redis 与 Memcached

两者都是非关系型内存键值数据库。有以下主要不同：

数据类型

Memcached 仅支持字符串类型，而 Redis 支持五种不同种类的数据类型，使得它可以更灵活地解决问题。

数据持久化

Redis 支持两种持久化策略：RDB 快照和 AOF 日志，而 Memcached 不支持持久化。

分布式

Memcached 不支持分布式，只能通过在客户端使用一致性哈希这样的分布式算法来实现分布式存储，这种方式在存储和查询时都需要先在客户端计算一次数据所在的节点。

Redis Cluster 实现了分布式的支持。

内存管理机制

在 Redis 中，并不是所有数据都一直存储在内存中，可以将一些很久没用的 value 交换到磁盘。而 Memcached 的数据则会一直在内存中。

Memcached 将内存分割成特定长度的块来存储数据，以完全解决内存碎片的问题，但是这种方式会使得内存的利用率不高，例如块的大小为 128 bytes，只存储 100 bytes 的数据，那么剩下的 28 bytes 就浪费掉了。

六、键的过期时间

Redis 可以为每个键设置过期时间，当键过期时，会自动删除该键。

对于散列表这种容器，只能为整个键设置过期时间（整个散列表），而不能为键里面的单个元素设置过期时间。

七、数据淘汰策略

可以设置内存最大使用量，当内存使用量超过时施行淘汰策略，具体有 6 种淘汰策略。

策略	描述
volatile-lru	从已设置过期时间的数据集中挑选最近最少使用的数据淘汰
volatile-ttl	从已设置过期时间的数据集中挑选将要过期的数据淘汰

策略	描述
volatile-random	从已设置过期时间的数据集中任意选择数据淘汰
allkeys-lru	从所有数据集中挑选最近最少使用的数据淘汰
allkeys-random	从所有数据集中任意选择数据进行淘汰
noeviction	禁止驱逐数据

如果使用 Redis 来缓存数据时，要保证所有数据都是热点数据，可以将内存最大使用量设置为热点数据占用的内存量，然后启用 allkeys-lru 淘汰策略，将最近最少使用的数据淘汰。

作为内存数据库，出于对性能和内存消耗的考虑，Redis 的淘汰算法（LRU、TTL）实际实现上并非针对所有 key，而是抽样一小部分 key 从中选出被淘汰 key。

八、持久化

Redis 是内存型数据库，为了保证数据在断电后不会丢失，需要将内存中的数据持久化到硬盘上。

快照持久化

将某个时间点的所有数据都存放到硬盘上。

可以将快照复制到其它服务器从而创建具有相同数据的服务器副本。

如果系统发生故障，将会丢失最后一次创建快照之后的数据。

如果数据量很大，保存快照的时间会很长。

AOF 持久化

将写命令添加到 AOF 文件（Append Only File）的末尾。

对硬盘的文件进行写入时，写入的内容首先会被存储到缓冲区，然后由操作系统决定什么时候将该内容同步到硬盘，用户可以调用 file.flush() 方法请求操作系统尽快将缓冲区存储的数据同步到硬盘。可以看出写入文件的数据不会立即同步到硬盘上，在将写命令添加到 AOF 文件时，要根据需求来保证何时同步到硬盘上。

有以下同步选项：

选项	同步频率
always	每个写命令都同步
everysec	每秒同步一次
no	让操作系统来决定何时同步

- always 选项会严重减低服务器的性能；
- everysec 选项比较合适，可以保证系统奔溃时只会丢失一秒左右的数据，并且 Redis 每秒执行一次同步对服务器性能几乎没有任何影响；
- no 选项并不能给服务器性能带来多大的提升，而且也会增加系统奔溃时数据丢失的数量。

随着服务器写请求的增多，AOF 文件会越来越大。Redis 提供了一种将 AOF 重写的特性，能够去除 AOF 文件中的冗余写命令。

九、发布与订阅

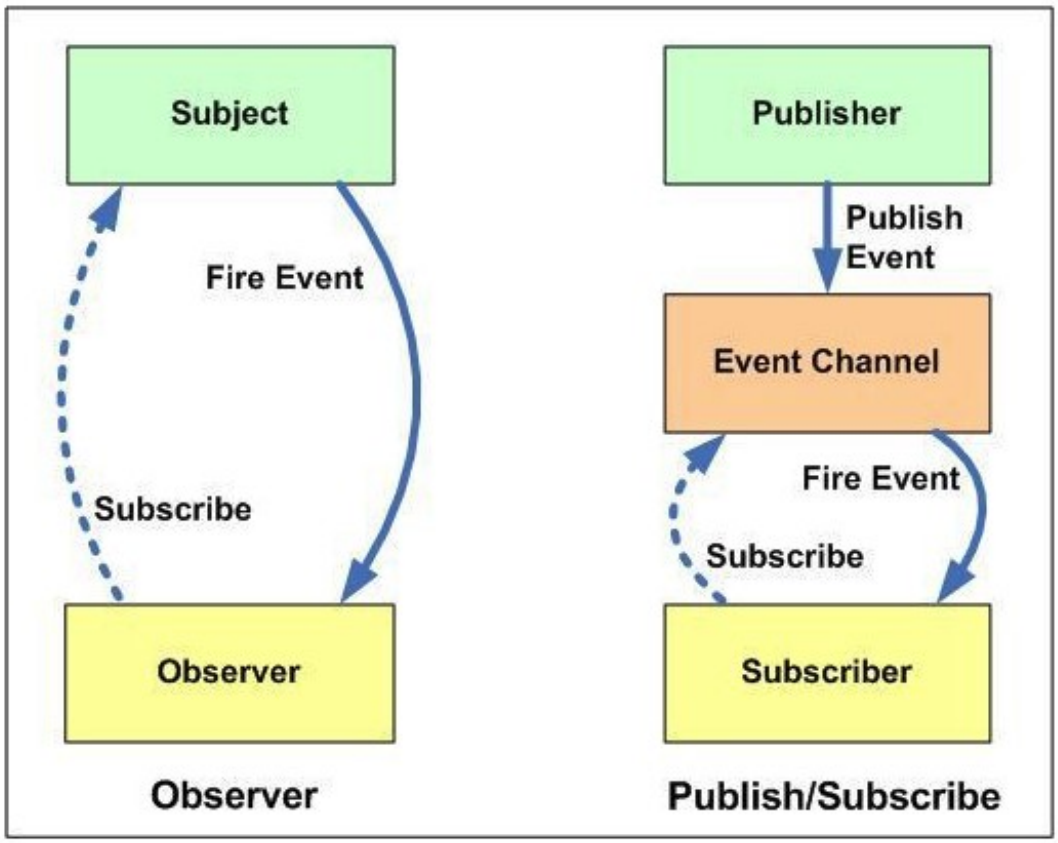
订阅者订阅了频道之后，发布者向频道发送字符串消息会被所有订阅者接收到。

某个客户端使用 SUBSCRIBE 订阅一个频道，其它客户端可以使用 PUBLISH 向这个频道发送消息。

发布与订阅模式和观察者模式有以下不同：

- 观察者模式中，观察者和主题都知道对方的存在；而在发布与订阅模式中，发布者与订阅者不知道对方的存在，它们之间通过频道进行通信。
- 观察者模式是同步的，当事件触发时，主题会去调用观察者的方法，然后等待方法返回；而发布与订阅模式是异步的，发布者向频道发送一个消息之后，就不需要关心订阅者何时

去订阅这个消息。



十、事务

一个事务包含了多个命令，服务器在执行事务期间，不会改去执行其它客户端的命令请求。

事务中的多个命令被一次性发送给服务器，而不是一条一条发送，这种方式被称为流水线，它可以减少客户端与服务器之间的网络通信次数从而提升性能。

Redis 最简单的事务实现方式是使用 MULTI 和 EXEC 命令将事务操作包围起来。

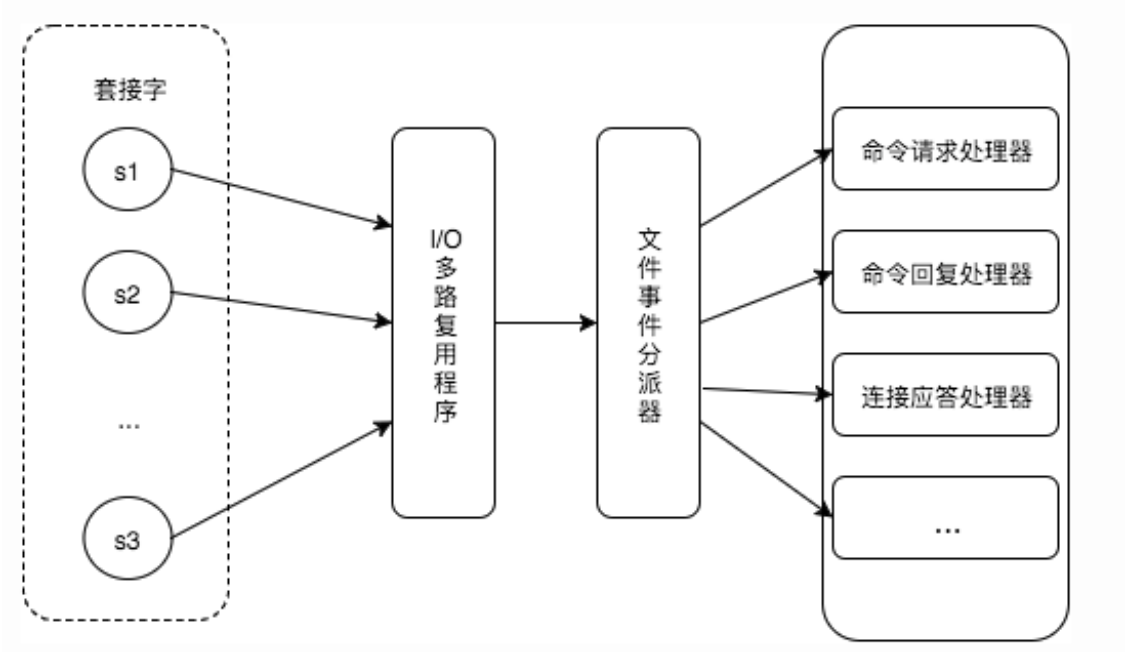
十一、事件

Redis 服务器是一个事件驱动程序。

文件事件

服务器通过套接字与客户端或者其它服务器进行通信，文件事件就是对套接字操作的抽象。

Redis 基于 Reactor 模式开发了自己的网络事件处理器，使用 I/O 多路复用程序来同时监听多个套接字，并将到达的事件传送给文件事件分派器，分派器会根据套接字产生的事件类型调用相应的事件处理器。



时间事件

服务器有一些操作需要在给定的时间点执行，时间事件是对这类定时操作的抽象。

时间事件又分为：

- 定时事件：是让一段程序在指定的时间之内执行一次；
- 周期性事件：是让一段程序每隔指定时间就执行一次。

Redis 将所有时间事件都放在一个无序链表中，通过遍历整个链表查找出已到达的时间事件，并调用相应的事件处理器。

事件的调度与执行

服务器需要不断监听文件事件的套接字才能得到待处理的文件事件，但是不能一直监听，否则时间事件无法在规定的时间内执行，因此监听时间应该根据距离现在最近的时间事件来决定。

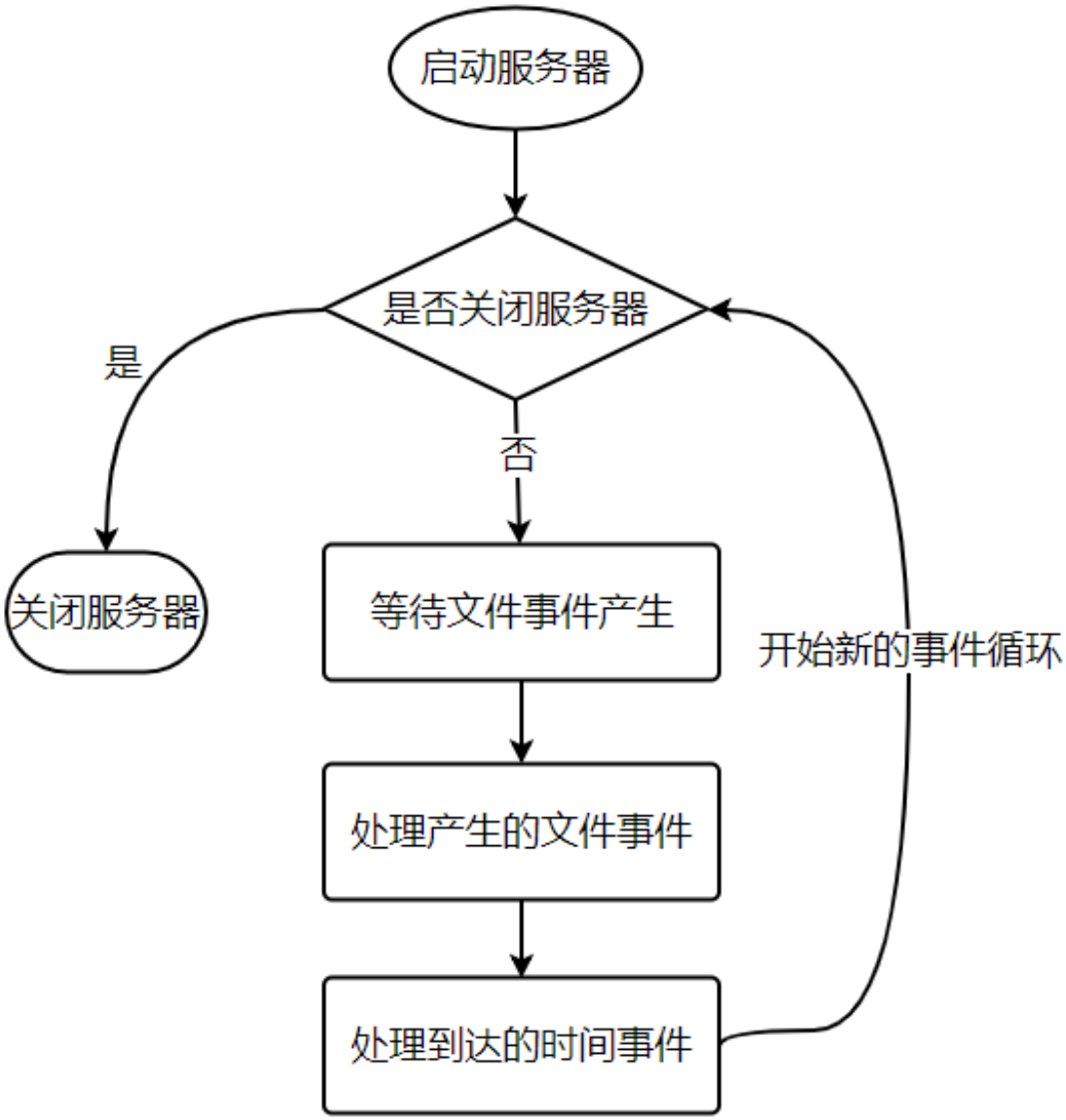
事件调度与执行由 aeProcessEvents 函数负责，伪代码如下：

```
def aeProcessEvents():
    # 获取到达时间离当前时间最接近的时间事件
    time_event = aeSearchNearestTimer()
    # 计算最接近的时间事件距离到达还有多少毫秒
    remaind_ms = time_event.when - unix_ts_now()
    # 如果事件已到达，那么 remaind_ms 的值可能为负数，将它设为 0
    if remaind_ms < 0:
        remaind_ms = 0
    # 根据 remaind_ms 的值，创建 timeval
    timeval = create_timeval_with_ms(remaind_ms)
    # 阻塞并等待文件事件产生，最大阻塞时间由传入的 timeval 决定
    aeApiPoll(timeval)
    # 处理所有已产生的文件事件
    procesFileEvents()
    # 处理所有已到达的时间事件
    processTimeEvents()
```

将 aeProcessEvents 函数置于一个循环里面，加上初始化和清理函数，就构成了 Redis 服务器的主函数，伪代码如下：

```
def main():
    # 初始化服务器
    init_server()
    # 一直处理事件，直到服务器关闭为止
    while server_is_not_shutdown():
        aeProcessEvents()
    # 服务器关闭，执行清理操作
    clean_server()
```

从事件处理的角度来看，服务器运行流程如下：



十二、复制

通过使用 slaveof host port 命令来让一个服务器成为另一个服务器的从服务器。

一个从服务器只能有一个主服务器，并且不支持主主复制。

连接过程

1. 主服务器创建快照文件，发送给从服务器，并在发送期间使用缓冲区记录执行的写命令。快照文件发送完毕之后，开始向从服务器发送存储在缓冲区中的写命令；
2. 从服务器丢弃所有旧数据，载入主服务器发来的快照文件，之后从服务器开始接受主服务器发来的写命令；
3. 主服务器每执行一次写命令，就向从服务器发送相同的写命令。

主从链

随着负载不断上升，主服务器可能无法很快地更新所有从服务器，或者重新连接和重新同步从服务器将导致系统超载。为了解决这个问题，可以创建一个中间层来分担主服务器的复制工作。中间层的服务器是最上层服务器的从服务器，又是最下层服务器的主服务器。

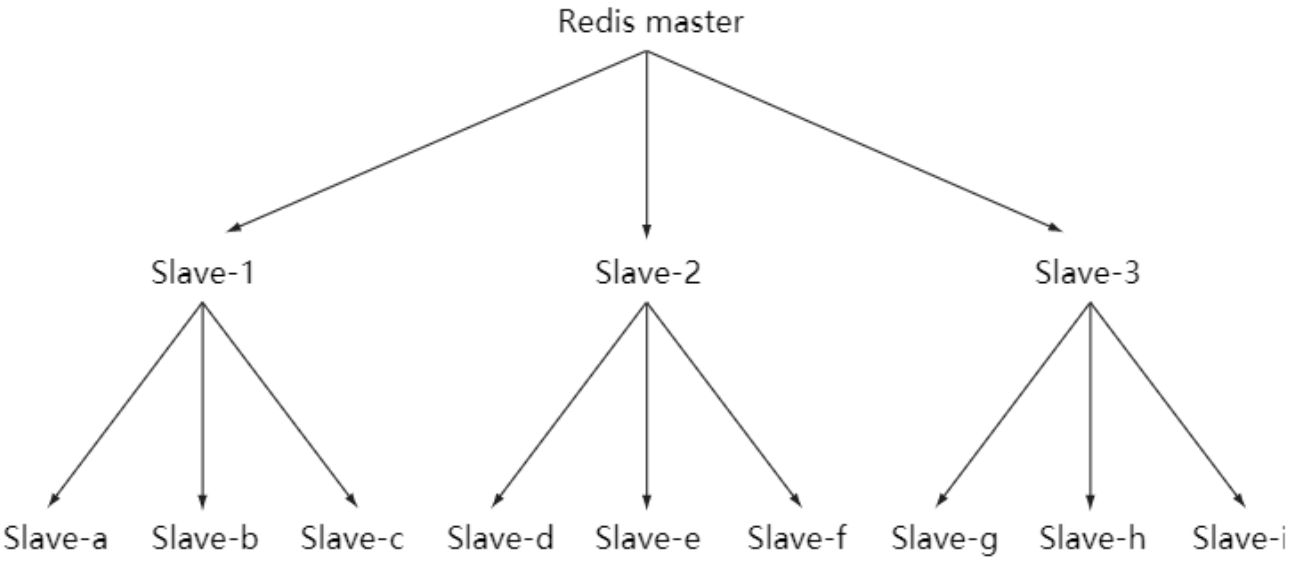


Figure 4.1An example Redis master/slave replica tree with nine lowest-level slaves and three intermediate replication helper servers

十三、Sentinel

Sentinel（哨兵）可以监听主服务器，并在主服务器进入下线状态时，自动从从服务器中选举出新的主服务器。

十四、分片

分片是将数据划分为多个部分的方法，可以将数据存储到多台机器里面，也可以从多台机器里面获取数据，这种方法在解决某些问题时可以获得线性级别的性能提升。

假设有 4 个 Reids 实例 R0，R1，R2，R3，还有很多表示用户的键 user:1，user:2，... 等等，有不同的方式来选择一个指定的键存储在哪个实例中。最简单的方式是范围分片，例如用户 id 从 0\~1000 的存储到实例 R0 中，用户 id 从 1001\~2000 的存储到实例 R1 中，等等。但是这样需要维护一张映射范围表，维护操作代价很高。还有一种方式是哈希分片，使用 CRC32 哈希函数将键转换为一个数字，再对实例数量求模就能知道应该存储的实例。

根据执行分片的位置，可以分为三种分片方式：

- 客户端分片：客户端使用一致性哈希等算法决定键应当分布到哪个节点。
- 代理分片：将客户端请求发送到代理上，由代理转发请求到正确的节点上。
- 服务器分片：Redis Cluster。

十五、一个简单的论坛系统分析

该论坛系统功能如下：

- 可以发布文章；
- 可以对文章进行点赞；
- 在首页可以按文章的发布时间或者文章的点赞数进行排序显示。

文章信息

文章包括标题、作者、赞数等信息，在关系型数据库中很容易构建一张表来存储这些信息，在 Redis 中可以使用 HASH 来存储每种信息以及其对应的值的映射。

Redis 没有关系型数据库中的表这一概念来将同种类型的数据存放在一起，而是使用命名空间的方式来实现这一功能。键名的前面部分存储命名空间，后面部分的内容存储 ID，通常使用 : 来进行分隔。例如下面的 HASH 的键名为 article:92617，其中 article 为命名空间，ID 为 92617。

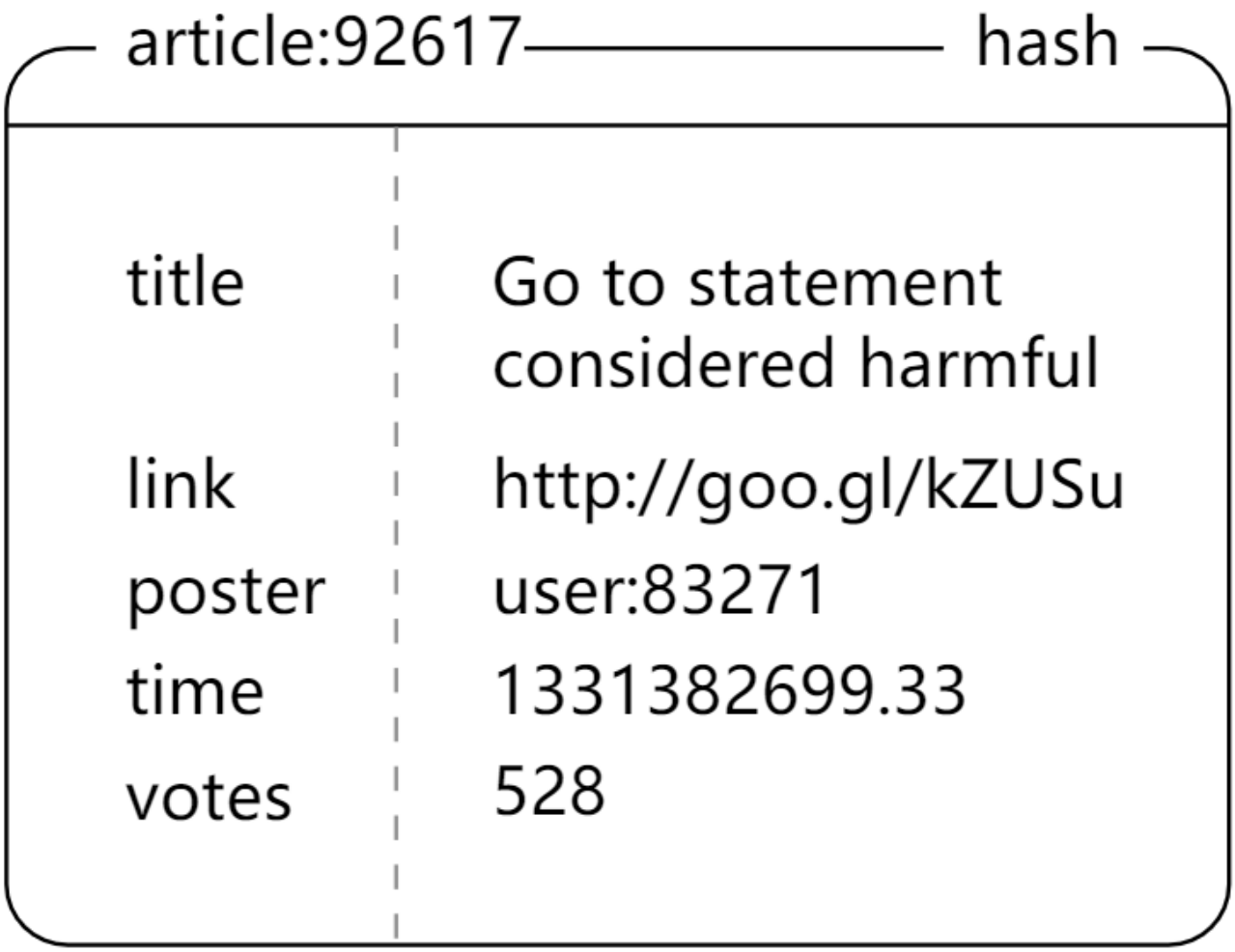


Figure 1.8 An example article stored as a HASH for our article voting system

点赞功能

当有用户为一篇文章点赞时，除了要对该文章的 votes 字段进行加 1 操作，还必须记录该用户已经对该文章进行了点赞，防止用户点赞次数超过 1。可以建立文章的已投票用户集合来进行记录。

为了节约内存，规定一篇文章发布满一周之后，就不能再对它进行投票，而文章的已投票集合也会被删除，可以为文章的已投票集合设置一个一周的过期时间就能实现这个规定。

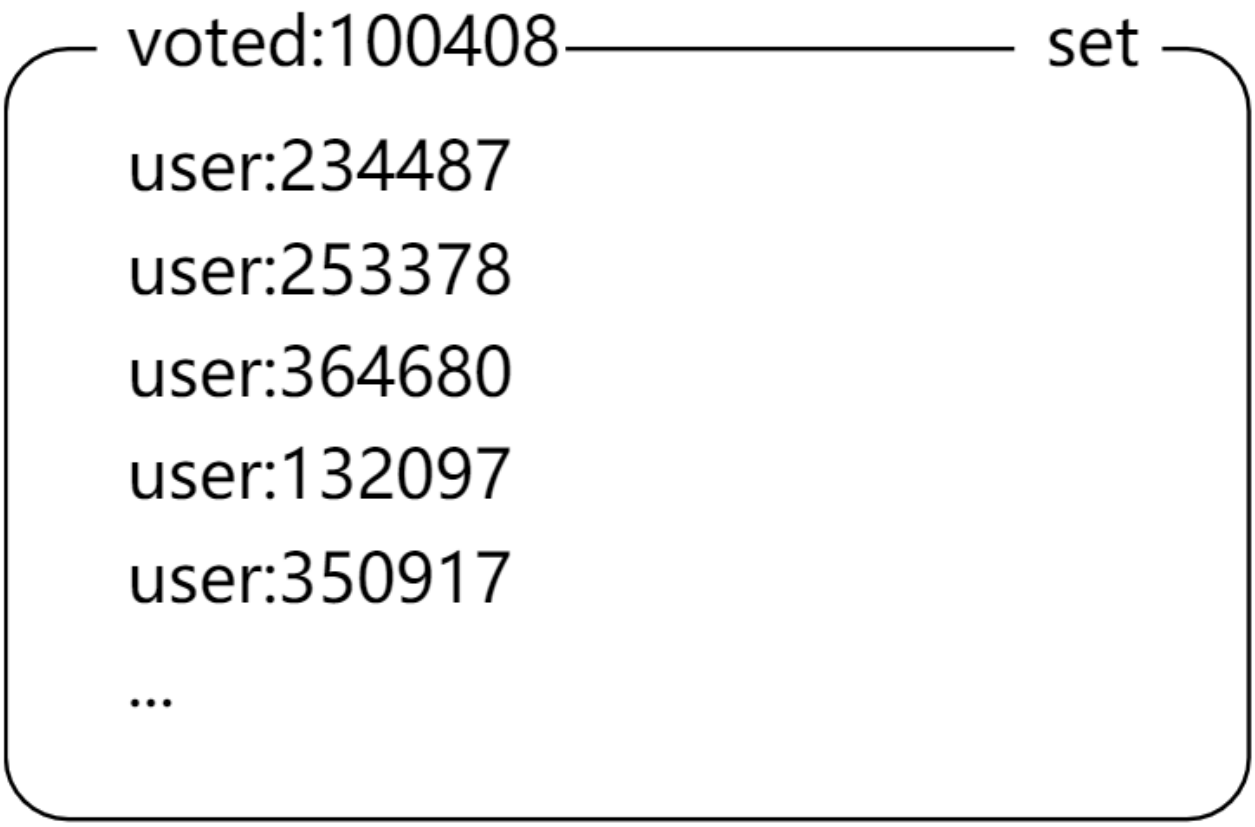


Figure 1.10 Some users who have voted for article 100408

对文章进行排序

为了按发布时间和点赞数进行排序，可以建立一个文章发布时间的有序集合和一个文章点赞数的有序集合。（下图中的 score 就是这里所说的点赞数；下面所示的有序集合分值并不直接是时间和点赞数，而是根据时间和点赞数间接计算出来的）

Figure 1.9 Two sorted sets representing time-ordered and score-ordered article indexes

参考资料

- Carlson J L. Redis in Action[J]. Media.johnwiley.com.au, 2013.
- 黄健宏. Redis 设计与实现 [M]. 机械工业出版社, 2014.
- REDIS IN ACTION
- Skip Lists: Done Right
- 论述 Redis 和 Memcached 的差异
- Redis 3.0 中文版- 分片
- Redis 应用场景
- Observer vs Pub-Sub