

物化视图

1.1 概念

物化视图是相对于视图而言的，但是两者实际上并没有什么关系就如java/javascript一样

首先mysql的视图不是一种物化视图，他相当于一个虚拟表，本身并不存储数据，当sql在操作视图时所有数据都是从其他表中查询出来的。者带来的问题是使用视图并不能将常用数据分离出来，优化查询速度，且操作视图的很多命令和普通表一样，这回导致在业务中无法通过sql区分表和视图，是代码变得复杂。

视图是简化设计，清晰编码的东西，他并不是提高性能的，他的存在只会降低性能（如一个视图7个表关联，另一个视图 8个表，程序员不知道，觉得很方便，把两个视图关联再做一个视图，那就惨了），他的存在未了在设计上的方便性。

物化视图可以帮助加快严重依赖某些聚合结果的查询。 如果插入速度不是问题，则此功能可以帮助减少系统上的读取负载。 可以看出来数据量庞大的时候这个时间...

1.2 物化视图更新方式

名称	描述
从不更新	只在开始更新
根据需要	每天，每夜
及时	每次修改数据之后
全部更新	速度慢，完全从无到有
延时的	速度快，使用log表

1.3 延迟更新与及时更新

数据表

表结构：

34 desc purchase_order;
35

<

信息结果1概况状态

Field	Type	Null	Key	Default	Extra
order_id	smallint(5)	NO	PRI	(Null)	auto_increment
order_sn	varchar(30)	YES			
plan_id	smallint(5)	NO		0	
group_id	int(10)	YES		0	
pro_num	int(8)	YES		0	
pro_price	decimal(10,2)	YES		0.00	
pro_cat	smallint(4)	YES		0	
supply_name	varchar(255)	YES		(Null)	

表数据：

```
1 select * from `purchase_order`;
```

信息	结果1	概况	状态					
order_id	order_sn	plan_id	group_id	pro_num	pro_price	pro_cat	supply_name	
1	C171230000001_1	1	1	20	480	1	shineyork	
2	C180102000001_1	2	2	25	9040	2	peter	

延迟更新

延迟更新特性：开销小，结果响应慢

mysql实现方式：定时调用存储过程函数即可

程序实现：定时计划处理

mysql实现方式

创建物化视图

```
drop table purchase_mv;

CREATE TABLE purchase_mv(
  supply_name VARCHAR(60) NOT NULL ,
  pro_count INT NOT NULL,
  pro_price_sum INT NOT NULL,
  pro_price_avg FLOAT NOT NULL,
  pro_num_sum INT NOT NULL,
  pro_num_avg FLOAT NOT NULL,
  UNIQUE INDEX supply_name (supply_name)
);
```

--第一步：确定执行的查询语句

```
SELECT
  supply_name,
  count(*) pro_count,
  sum(pro_price) pro_price_sum,
  avg(pro_price) pro_price_avg,
  sum(pro_num) pro_num_sum,
  avg(pro_num) pro_num_avg
from
  purchase_order
group by supply_name;
```

-- 第二步：创建与物化视图相关视图

```
drop view por_view;
create view por_view
as
  SELECT
    supply_name,
    count(*) pro_count,
    sum(pro_price) pro_price_sum,
    avg(pro_price) pro_price_avg,
    sum(pro_num) pro_num_sum,
    avg(pro_num) pro_num_avg
  from
    purchase_order
  group by supply_name;
```

```
-- 添加数据
insert into purchase_mv select * from por_view
```

按需更新物化视图

根据需要更新物化视图，我们可以用存储过程来实现

```
--第三步：创建存储过程
DROP PROCEDURE refresh_mv_now;
DELIMITER $$
CREATE PROCEDURE refresh_mv_now ()
BEGIN
    TRUNCATE TABLE purchase_mv;
    INSERT INTO purchase_mv SELECT * FROM por_view;
END;
$$DELIMITER ;

-- 测试
INSERT INTO purchase_order (order_sn, `ctime`, supply_id, supply_name, pro_num, pro_price, shipping_state) VALUES
('C1803221553615160063su', '1521701992', '1', '友阿果园', 75, 479, 2)

CALL refresh_mv_now();
SELECT * FROM purchase_mv;
```

及时更新

及时更新特性：开销大，结果响应快

mysql实现方式:执行insert, update, delete, alter操作后执行触发器

程序实现：异步队列事件方式

mysql实现方式

在每个语句之后进行完全刷新是没有意义的。但我们仍然希望得到正确的结果。

要做到这一点，要复杂一点。在purchase_order表上的每一个插入项上，我们都必须更新我们的物化视图。我们可以通过purchase_order表上的INSERT/UPDATE/DELETE触发器透明地实现这一点：这里以insert为例子

思路：通过触发器，然后在添加完数据之后获取之前的聚合值的数据，然后根据新增的这条数据再做实时更新

```
-- 触发器实现
after insert on drop trigger purchase_mv_trigger_ins;

DELIMITER $$
CREATE TRIGGER purchase_mv_trigger_ins AFTER INSERT
ON purchase_order FOR EACH ROW
BEGIN
    SET @old_pro_price_sum = 0;
    SET @old_pro_price_avg = 0;
    SET @old_pro_num_sum = 0;
    SET @old_pro_num_avg = 0;
    SET @old_pro_count = 0;

    # 查询出之前的信息然后记录到不同的变量中
    SELECT
        IFNULL(pro_price_sum,0),
        IFNULL(pro_price_avg,0),
        IFNULL(pro_num_sum,0),
        IFNULL(pro_num_avg,0),
        IFNULL(pro_count,0)
    FROM
        purchase_mv
    WHERE
        supply_name = NEW.supply_name
    INTO
        @old_pro_price_sum,@old_pro_price_avg,@old_pro_num_sum,@old_pro_num_avg,@old_pro_count;
```

```
# 然后再去计算更新操作之后的内容
SET @new_pro_count = @old_pro_count + 1;
SET @new_pro_price_sum = @old_pro_price_sum + NEW.pro_price;
SET @new_pro_price_avg = @new_pro_price_sum / @new_pro_count;
SET @new_pro_num_sum = @old_pro_num_sum + NEW.pro_num;
SET @new_pro_num_avg = @new_pro_num_sum / @new_pro_count;
REPLACE INTO
    purchase_mv
VALUES(
    NEW.supply_name, @new_pro_count,
    @new_pro_price_sum, IFNULL(@new_pro_price_avg, 0),
    @new_pro_num_sum, IFNULL(@new_pro_num_avg, 0) );
END;
$$DELIMITER ;
```