分布式存储原理与TDDL

TAOBAO JM

你理解的存储有什么？

- MYSQL/ORACLE/PGSQL/SQLSERVER
- HDFS/TFS/BIGTABLE
- MEMCACHED/TAIR/REDIS
- HBASE
- Cassandra/mongodb/Couchdb/voldmort
- neo4j

问题

- http://nosql-database.org/ currently 122+
- I‘m the fastest.
- but we are web scale well.
- 我们最安全，从不会失败
- 我们最简单

如何选择适合自己的存储产品

????

提纲

- 介绍一般性的存储知识。
- 快速找到合适的存储
- TDDL在上述关键节点上的选择和思考
- 切分经验

知识准备

TAOBAO JM

准备

- 你在用关系数据库做什么？
  - 查询
  - 事务
  - 视图
  - 存储过程

准备

准备

Key-Value 存储

- 本质来说，就是“映射”，按照key找到val
  - 所有数据存储的最基本和最底层的结构
  - 不文件系统找指定的数据的作用相同，也是根据指定的key查找到对应的数据。

```
Map<String,String> mapping = new HashMap();
mapping.put(key,val);
mapping.get(key);
```

Key-Value 存储

- 对映射来说的关键特性
  - 是否支持范围查找？
  - 是否能够处理更新？
  - 读写性能指标？
  - 是否面向磁盘结构？
  - 并行指标？
  - 内存占用
  - Etc…

•回忆数据结构

排序数组
排序链表
跳表
B tree
LSM Tree
HashMap
Fractal Tree
Red-Black-tree
COLA
Etc…

准备

关系代数

- 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

–Select * from tab where id = ?

关系代数

- 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

- Select * from tab where user_id = ?

User_id	id
0	0
0	1
1	2
3	3

二级索引

关系代数

- 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

–Select …where user_id = ? And name = 袜子

User_id	Name
0	袜子
0	鞋子
1	计算机
3	电池

Primary Key: id
0
1
2
3

组合索引

关系代数

- 问题:

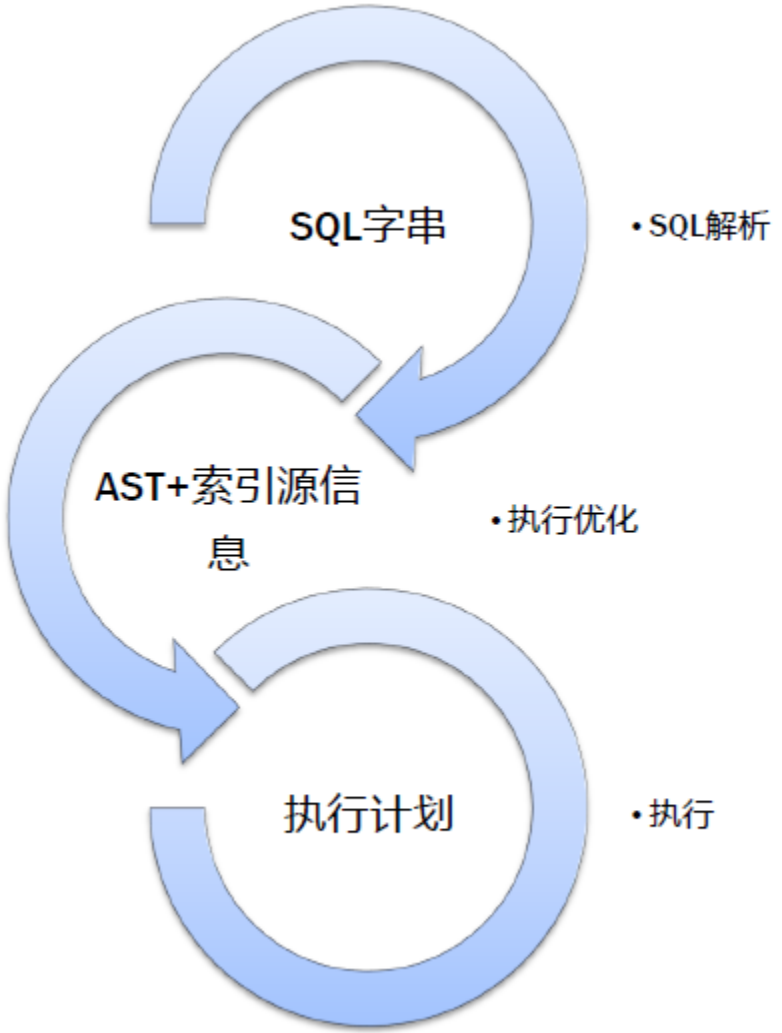
Primary Key: id	User_id	性别
0	0	男
1	0	男
2	1	女
3	3	女

- Select * from table where user_id in (?,?,?,?,?) and 性别=‘女’

准备

SQL引擎

- AST
  - 抽象语法树
  - 标记SQL的组成方式
- 执行计划
  - 告知执行器如何高效的利用K-V



分布式存储-原理

TAOBAO JM

多机Key-Value存储

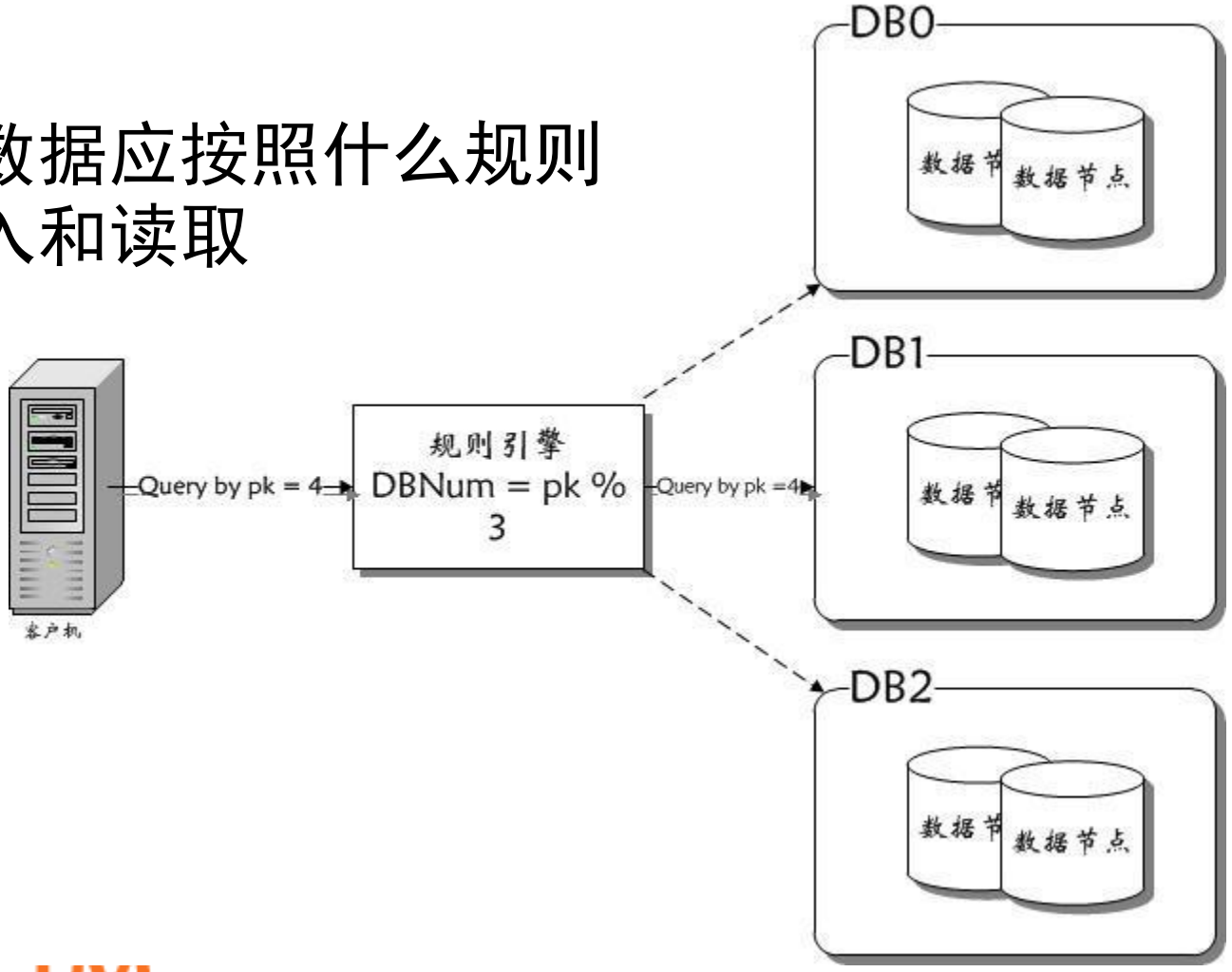
- 关键特性
  - 可运维
  - 高性能
  - 可以比较容易的扩容
  - 核心数据结构还是hash和树，部分case针对多机做了一点点优化。

多机Key-Value存储

- 代表性组件
  - mongoDB -> mongos 服务器
  - Hbase -> region server + client jar包
  - TDDL -> tddl-rule 组件+大禹数据迁移

路由

- 规则引擎
  - 有状态数据应按照什么规则进行写入和读取



路由

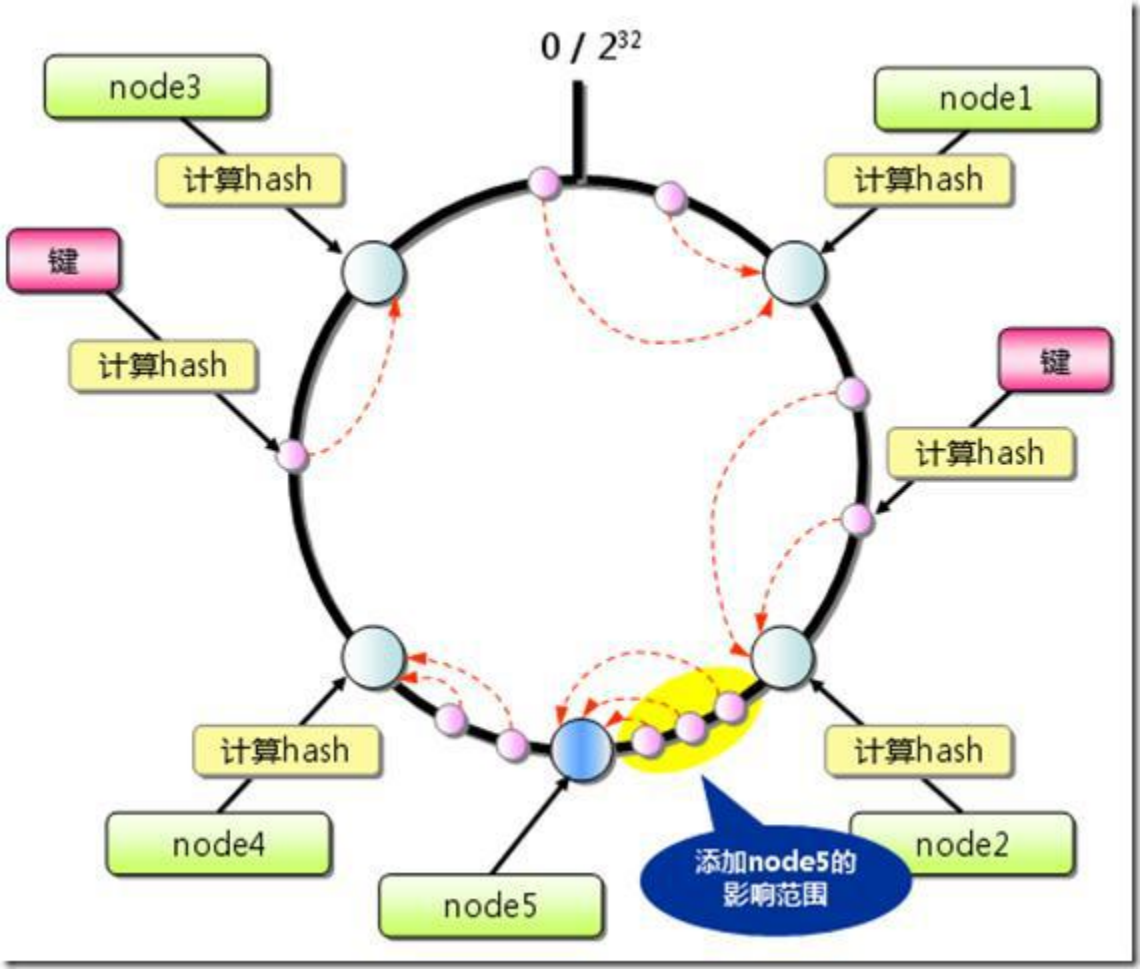
- 本质来说还是个查找的过程
  - Hash
    - O(1)效率
    - 不支持范围查询（按时间这样的查询条件就比较困难）
    - 不需要频繁调整数据分布
  - Tree
    - 主要是B-Tree
    - O(logN)效率
    - 支持范围查询
    - 需要频繁调整节点指针以适应数据分布

路由

- Hash
  - Id % n
  - 最普通的hash
  - 如果id % 3 -> id % 4 总共会有80%的数据发生移动，最好情况下是倍分 id % 3 -> id % 6，这时候会有50%的数据发生移动
  - 数据移动本身就是个要了亲命了。

路由

- Hash
  - 一致性hash



路由

- Hash
  - 一致性hash

```
Def idmod = id % 1000 ;
If(id >= 0 and id < 250)
        return db1;
Else if (id >= 250 and id < 500)
        return db2;
Else if (id >= 500 and id < 750)
        return db3;
Else
        return db4;
```

路由

- Hash
  - 一致性hash
    - 可以解决热点问题
    - 但如果热点均匀，加机器基本等于n->2n方案
    - 因为要在每个环上都加一台机器，才能保证所有节点的数据的一部分迁移到新加入的机器上

路由

- Hash
  - 虚拟节点

    Def hashid = Id % 65536

Hash id	Target node id
0	0
1	1
2	2
3	3
4	0
....	...
65535	3

路由

- Hash
- 虚拟节点
  - 解决热点问题，只需要调整对应关系即可
  - 解决n->n+1问题，规则可以规定只移动需要移动的数据
  - 方案相对的复杂一些
  - 一般推荐使用简单方案开始，使用n->2n方案扩容
  - 只有需要的情况下，再考虑平滑的扩展到虚拟节点方案即可

路由

- B-Tree
  - Hbase使用的切分方法
    - 支持范围查询
    - 对于大部分场景来说，引导列都是pk.userid一类的单值查询，用树相对复杂。
    - 需要频繁的进行切分和合并操作---region server的噩梦。
    - 固定节点情况下，跨度相对较大，查找效率可能会进一步降低

路由

- TDDL的选择
  - 规则引擎
    - Groovy脚本实现，本质是为了实现多版本的规则推送，其他事情，可以完全委托给业务的具体需求。
    - 内建对3种hash模式的支持，并且允许从简单hash平滑的过度到一致性hash和虚拟节点hash
    - 允许使用外部实现规则
    - 允许引入静态方法
  - 默认推荐使用id % n
  - 实现了多版本，并且与迁移工具绑定
  - 可以不其他产品无缝结合—MDDAL可以不用改动代码

路由

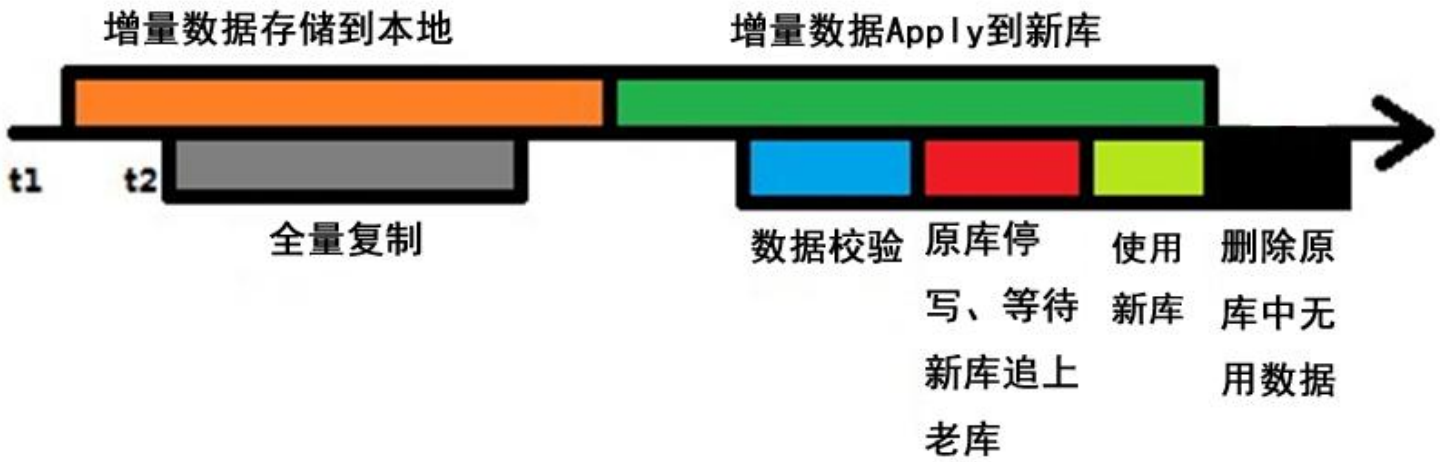
- 规则引擎
  - Groovy脚本，规则完全交由业务决定

```
if(#seller_id# is bigSeller)
{
        //如果特殊卖家，返回特殊库。
        return "db_spec";
}
else
{
        //如果一般卖家，那么返回正常库
        return "db_"+#seller_id# % 1024;
}
```

数据平滑迁移@淘宝

- 自动迁移
  - 不规则引擎相互配合
  - 不需要关心具体的规则，只要能够表示的规则，都可以进行数据迁移
  - 对业务的影响几乎为零

数据平滑迁移@淘宝



TAOBAO JM

数据平滑迁移@淘宝

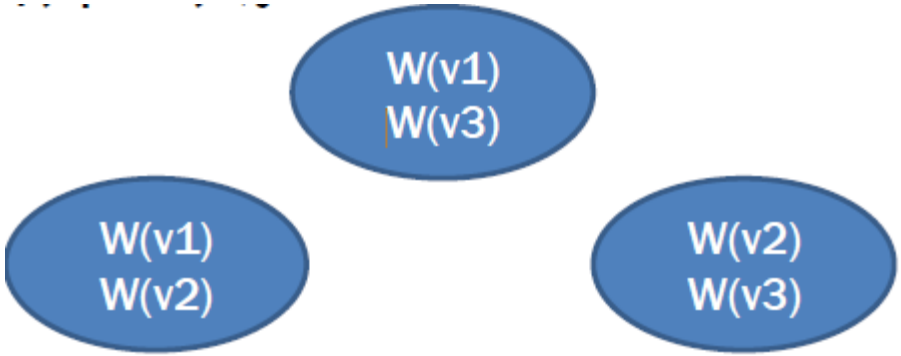
- 规则和迁移
  - 解决scale out问题
  - 可以解决大部分有状态节点的扩容问题
    - Bdb je?
    - Mongo db?
    - Pl SQL?
  - 规则+动态创建数据源可以实现存储资源的管理
    - 根据访问量和磁盘容量，决定你的数据节点应该如何分配。

一致性

- 寻求一种能够保证，在给定多台计算机，并且他们相互之间由网络相互连通，中间的数据没有拜占庭将军问题（数据不会被伪造）的前提下(P)，能够做到以下两个特性的方法：
  - 数据每次成功的写入，数据不会丢失，并且按照写入的顺序排列(C)
  - 给定安全级别（美国被爆菊？火星人入侵？），保证服务可用性，并尽可能减少机器的消耗。(A)

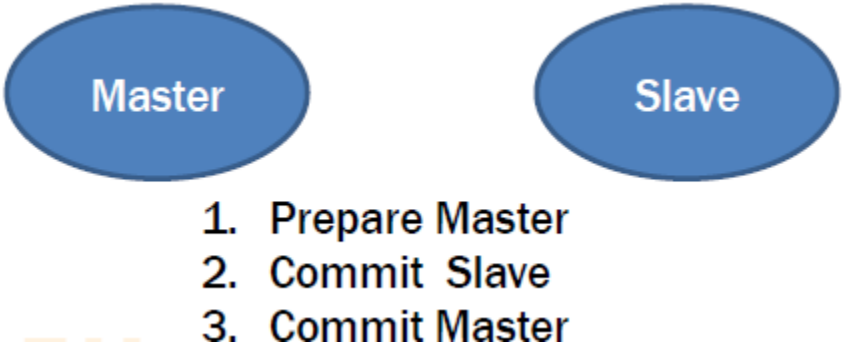
一致性

- •无主机方案
- –Dynamo /cassandra: gossip,W+R>N
- –所有节点可写，不存在单点故障
- –读数据的最新版本，需要将所有可写节点的数据都读出来合并一次



一致性选择

- 有主机方案
  - Mysql OcceanBase MongoDB Ora+fibreChannel
  - 只有一个节点可写，切换时存在短暂leader election 过程。会出现短暂不可写
  - 数据一致性比较好控制，读最新数据只需要读主机就可以。一致性读性能较好。

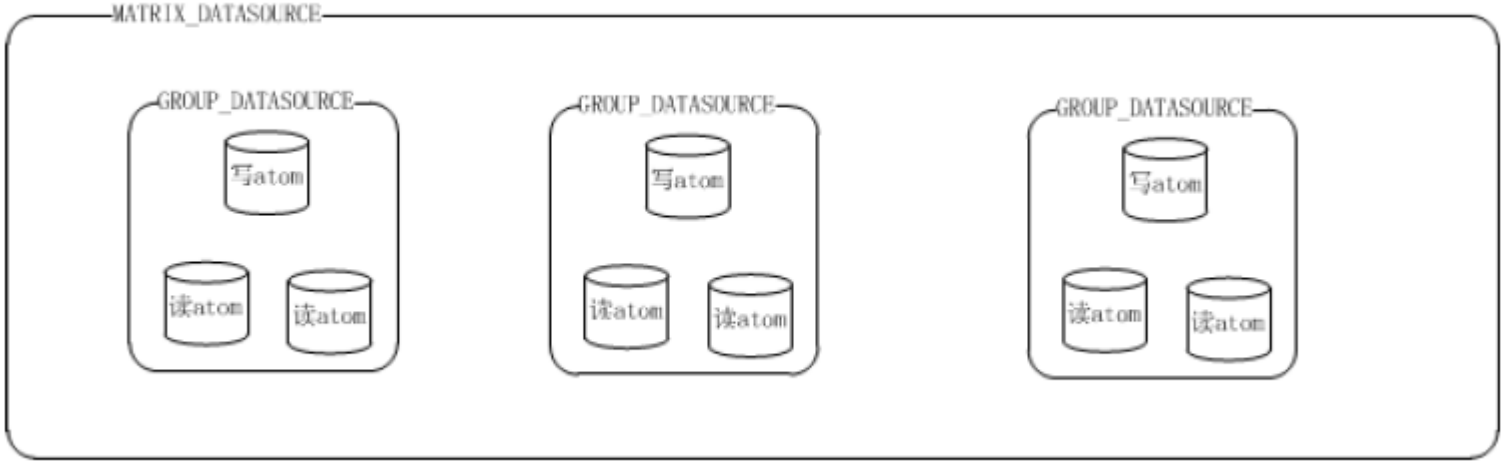


超越数据库HA

- 在实际的线上系统，还需要考虑以下问题
  - 有些机器负担写任务，因此读压力可能不均衡，因此必须有权重设置
  - 单个节点挂掉的时候，TCP超时会导致业务APP的线程花费更多的时间来处理单个请求，这样会降低APP的处理能力，导致雪崩。
  - 因为突发情况，导致数据库请求数增加，数据库响应变慢，导致雪崩

超越数据库HA

- TDDL的选择
  - 分为三层



TDDL的选择

- Matrix 层
  - 核心是规则引擎
    - 可以单独抽取出来，放在其他实现里，比如自己封装的ibatis DAO里面。
  - 通过规则引擎实现了动态扩容
  - 主要路径
    - Sql解析->规则引擎计算->数据执行->合并结果
  - 如果有更好的封装，我们非常欢迎

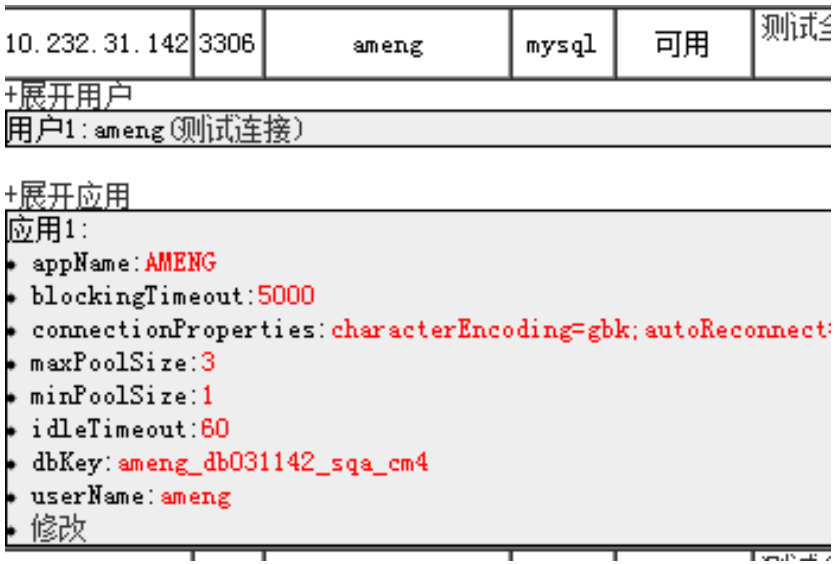
TDDL的选择

- GROUP 层
  - 读写分离
  - 权重
  - 写的HA切换
  - 读的HA切换
  - 允许动态添加新的slave节点

AD_MSGCENTER_GROUP	v031080_sqa_cm4_db_msgcenter:r10w10p0,v132068_sqa_cm4_db_msgcenter:r0w0p0	修改
AD_MSG_GROUP	v031080_sqa_cm4_db_msgcenter:r10w10p0,v132068_sqa_cm4_db_msgcenter:r0w0p0	修改

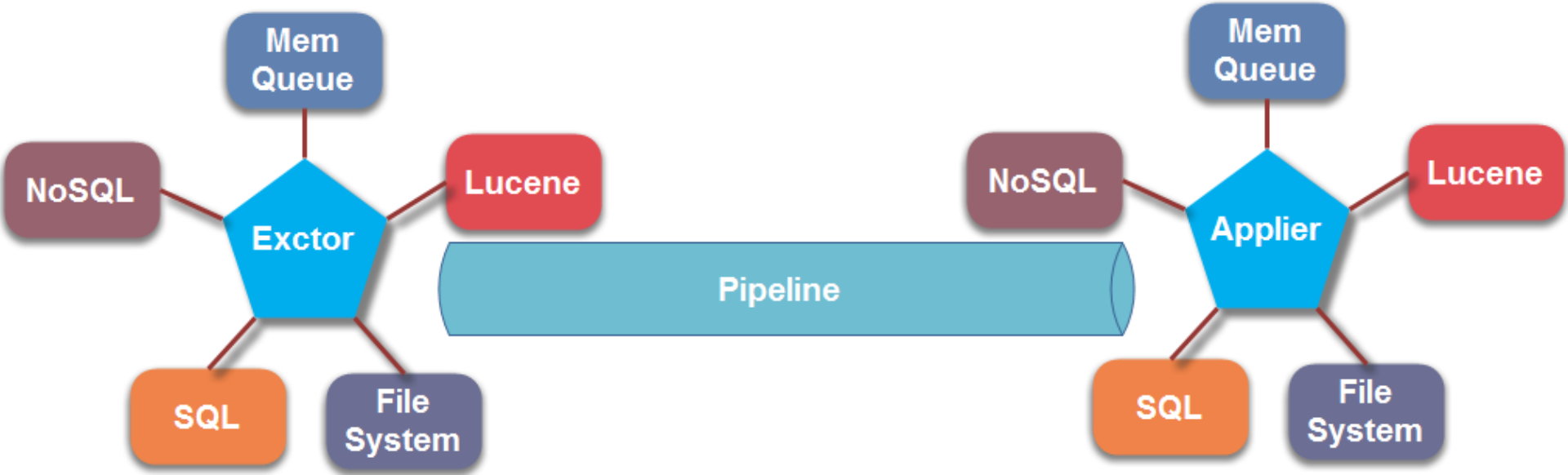
TDDL的选择

- Atom层
  - 单个数据库的抽象
  - 动态化的jboss数据源，ip port 用户名密码都可以动态修改。
  - Thread count模式，保护业务的处理线程，超过指定值，保护启动。
  - 动态阻止某个sql执行
  - 执行次数统计和限制



数据增量复制---精卫

- 处理数据的异构增量复制
  - 按买家分库，按卖家查询
  - 评价，被评价双维度查询
  - 数据的增量实时共享，通知给缓存系统，倒排索引等



数据增量复制---精卫

- 精卫要考虑的事情
  - Mysql 切换后应该如何配合？
  - 精卫单机出现故障，如何保证服务可靠？
  - 数据量过大，如何减小传输量？
  - 如何保证数据尽可能不丢失？
  - 易用性？接入的方便性

数据库实践

TAOBAO JM

TDDL最佳实践

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

TDDL最佳实践

- 加锁解锁 25 ns，走一次网络20000ns
- 顺序读内存1mb 250 000 ns.
- 顺序读磁盘1mb 20 000 000 ns 40X于内存
- Disk seek 额外的需要 10 000 000 ns 进行一次。Seek越多，顺序读越少，disk性能越低。

TDDL最佳实践

- 尽可能使用一对多规则中的一进行数据切分
  - 互联网行业，用户是个非常简单好用的维度。
- 卖家买家问题，使用数据增量复制的方式冗余数据进行查询。这种冗余从本质来说，就是索引。
- 合理利用表的冗余结构，减少走网络的次数
  - 卖家买家，都存了全部的数据，这样，按照卖家去查的时候，不需要再走一次网络回表到买家去查一次。

TDDL最佳实践

- 尽一切可能利用单机资源
  - 单机事务
  - 单机join
- 好的存储模型，就是尽可能多的做到以下几点:
  - 尽可能走内存
  - 尽可能将一次要查询到的数据物理的放在一起
  - 通过合理的数据冗余，减少走网络的次数
  - 合理并行提升响应时间
  - 读取数据瓶颈，可以通过加slave节点解决
  - 写入瓶颈，用规则sharding和扩容来解决

小结

- 存储= 数据结构+算法…
- 好的存储要考虑
  - 性能
  - 易用性
  - 可运维
  - 可监控
  - 结构尽可能简单
  - 组件化
- TDDL的核心组件
  - TDDL 规则引擎+自动迁移（大禹）
  - 数据异步Trigger （精卫）
  - 主备切换和半自动容灾（TDDL Group)
  - 监控 统计 易用性提升（TDDL Atom）
  - 易用性接口层 （And_Or)

THANK YOU

- http://qing.weibo.com/whisperxd 我的博客
- http://baike.corp.taobao.com/index.php/TDDL