基于变化数据捕获机制的分布式缓存一致性策略①

江 勇[1,2], 苗宗利[3], 王 伟[2], 段世凯[1,2], 刘财政[1,2], 支孟轩[2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院软件研究所 软件工程技术研究开发中心, 北京 100190)
[3](中国电子技术标准化研究院, 北京 100007)

摘 要: 分布式缓存被广泛应用于解决传统关系型数据库的性能瓶颈问题, 但是当不能感知分布式缓存的第三方应用直接更新后台数据库时, 缓存数据会获得不一致的状态, 存在过时缓存问题. 本文提出一种基于变化数据捕获机制的分布式缓存一致性策略, 集成了基于触发器和基于日志的两种变化数据捕获机制实时捕获后台数据库更新, 实现了数据模型自动转换方法和 SQL 翻译引擎, 实时更新缓存, 从而保障分布式缓存的一致性. 实验模拟 TPC-W 测试基准中的关键操作, 验证了基于日志的变化数据捕获机制相比基于触发器的变化数据捕获机制有更好的数据库性能和缓存一致性效果.

关键词: 分布式缓存; 变化数据捕获; 模型转换; SQL 翻译

Distributed Cache Coherency Strategy Based on Change Data Capture Mechanism

JIANG Yong[1,2], MIAO Zong-Li[3], WANG Wei[2], DUAN Shi-Kai[1,2], LIU Cai-Zheng[1,2], ZHI Meng-Xuan[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[3](China Electronics Standardization Institute, Beijing 100007, China)

Abstract: Distributed cache is widely used to solve the performance bottleneck problem in traditional relational database, but when third-party applications that are not cache-aware update the back-end database, the distributed cache will end up in an inconsistent state, which has the problem of stale cache data. This paper proposes a distributed cache consistency strategy based on change data capture mechanism. The work integrates trigger-based and log-based change data capture mechanism that can get the real-time data from backend database, and implements data model transformation and SQL translation engine, which can update cache in real-time to guarantee distributed cache coherence. The experiment simulates the key operation in TPC-W benchmark, which verifies that the change data capture based on log has the better database performance and cache consistency effects compared with the change data capture based on trigger.

Key words: distributed cache; change data capture; model transforming; SQL translation

1 引言

为了应对海量数据与大规模用户请求带来的挑战, 解决传统数据库面临的大规模数据访问瓶颈问题, 分布式缓存被广泛应用, 为用户提供高性能、高可用、可伸缩的数据缓存服务. 但是, 当不能感知缓存的第三方应用程序直接更新后台数据库数据时, 缓存会获得不一致的状态, 存在过时缓存问题[1], 如图 1 所示.

保持缓存和后台数据库数据的数据一致性一直是开发人员重点关注的问题. 现有的典型分布式缓存方案, 如 Memcached[2], Redis[3], Hazelcast[4]等, 主要通过基于过期的缓存一致性策略来保持缓存和后台数据库的数据一致性. 每个动态的缓存条目都会创建一个默认的存在时间清除器, 在预定义的时间间隔后会清除相应的数据条目, 但是过期时间的设定是一条经验规

① 收稿时间:2016-03-21;收到修改稿时间:2016-04-08 [doi:10.15888/j.cnki.csa.005450]

则, 需要开发人员对数据的准确性需求和对数据过时程度的容忍度有足够的了解, 然后才能做出合适的决策. IBM 的 WebSphere eXtreme Scale[5]实现了基于轮询的缓存一致性策略, 由缓存定期查询数据库以确定自上次加载以来数据是否发生了变更, 已在后台数据库中更新的条目失效或者使用新的数据更新缓存. 但是, 这些定期查询会给后台数据库带来较大的负载压力, 轮询机制也会消耗额外的 CPU 资源.

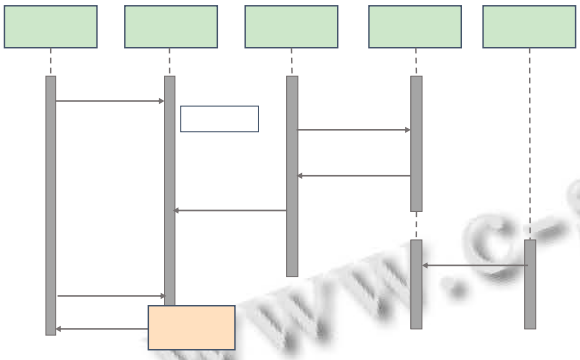

图 1 过时缓存问题

针对现有缓存方案的特点以及缓存一致性策略存在的问题, 本文提出一种基于变化数据捕获机制[6] (Change Data Capture, CDC)的缓存一致性策略, 主要包括以下三个方面内容:

(1) 变化数据捕获机制. 设计实现关系型数据库的数据变化捕获机制, 实时监听后台数据库的数据变化, 捕获数据更新并传送到分布式缓存, 支持基于触发器和基于日志两种方式.

(2) 数据模型自动转换方法. 实现关系型数据模型向 key-value 数据模型的自动转换, 将数据库表中的行数据自动转换为分布式缓存的对象数据.

(3) SQL 翻译引擎. 将变化数据捕获机制捕获到的 SQL 更新操作翻译为缓存的 key-value 操作, 从而将更新同步到缓存.

2 相关工作

文献[7,8]较为全面的介绍了变化数据捕获技术, 主要有基于表记录、复制、基于触发器、基于日志等多种方法, 并比较了不同方法的条件、优点、缺点和适用场合. 文献[9]和文献[10]分别介绍了基于触发器和基于日志的变化数据捕获技术, 但是文献[9]面向的是 ETLM 过程的数据仓库, 文献[10]面向实时商业智能系统.

在缓存一致性策略方面, 文献[11]介绍了同步进制实现分布式缓存的强一致性, 每次更新数据时, 会同步更新所有缓存结点, 然后返回. 这种机制适用于以读请求为主的应用场景, 而当数据更新操作频繁时, 强一致性的同步机制会显著增加系统响应时间. 文献[12]实现了在缓存增强的 SQL 系统上的缓存强一致性, 并提出了 IQ 框架, 同时满足 ACID 属性, 但是分布式缓存受到 CAP 理论的约束, 该方法并不能够在分布式缓存中适用; 文献[13]将基于触发器的数据捕获技术应用到缓存, 保障缓存的一致性, 提出了一种面向 ORM 框架的缓存中间件, 是数据捕获技术的一次成功应用. 但是该文献重点在于缓存的 ORM 访问方式, 对数据捕获技术在缓存中的应用讨论得并不够细致.

上述相关工作中, 有将 CDC 应用到数据仓库和智能系统的, 但是仍缺少将 CDC 技术与分布式缓存的集成; 有的实现了缓存的强一致性, 但是存在场景限制, 只适合读请求为主的场景; 有的提出创新的缓存一致性框架, 但并不适合分布式缓存. 本文重点关注将变化数据捕获技术应用到分布式缓存中需解决的模型转换, SQL 翻译等问题, 并用两种方式实现变化数据捕获.

3 面向分布式缓存的变化数据捕获机制

针对现有缓存方案的特点以及缓存一致性策略存在的问题, 本文提出一种基于变化数据捕获机制的缓存一致性策略, 通过在源系统中添加触发器和获取关系型数据库日志两种方式实现变化数据捕获机制, 然后实现关系型数据模型向 key-value 数据模型的自动转换, 最后解析 SQL, 更新缓存, 实现数据从关系型数据库到分布式缓存的自动同步.

3.1 变化数据捕获机制

变化数据捕获技术是基于对数据源改变部分的数据识别、数据获取和数据传送技术来实现的, 在数据源数据发生变化时, 将实时捕获变更的数据并同步更新到分布式缓存中, 从而保障分布式缓存的一致性. 本文实现基于触发器和基于日志两种变化数据捕获机制, 并对比其优缺点.

3.1.1 基于触发器的变化数据捕获

通过在关系型数据库中设置触发器并设计一张日志表与源数据表相关联, 当源数据表数据发生变化时, 通过触发器机制自动记录数据变化到日志表. 同时, 监听线程实时监听日志表信息, 获取最新的数据变化, 并通过数据传送渠道将变更数据更新到缓存.

该方法的重点在于触发器的设计. 触发器一共有三种类型, 分别对应数据库的插入, 删除, 更新操作, 任何数据表的变更操作, 都会触发触发器, 从而在触发器日志表产生记录. 图 2 是针对数据库 test 中的 orders 表插入操作时创建的触发器.

```
DROP TRIGGER IF EXISTS `test`.`orders_i`;
delimiter ///
    CREATE TRIGGER `test`.`orders_i` AFTER insert ON
    `test`.`orders`
      FOR EACH ROW BEGIN
          INSERT INTO `test`.`TRIGGERLOG` (`CTIME`,
                `TYPE_CODE`, `SCHEMA_NAME`,
                `TABLE_NAME`, `NEW_PRIMARYKEY0`,
                `CLIENT_NAME`) VALUES (now(), 1, test,
                'orders', NEW.`orderId`, user());
          END;          ///
delimiter ;
```

图 2 触发器实例

3.1.2 基于日志的变化数据捕获

关系型数据库都管理着一个事务日志, 其中记录了对数据库内容和元数据所做的更改. 基于日志的变化数据捕获, 以关系型数据库的事务日志为基础, 并对其进行实时监控, 一旦源数据库发生数据变化, 就进行实时捕获, 对需要实时同步的数据进行捕获, 如图 3 所示.

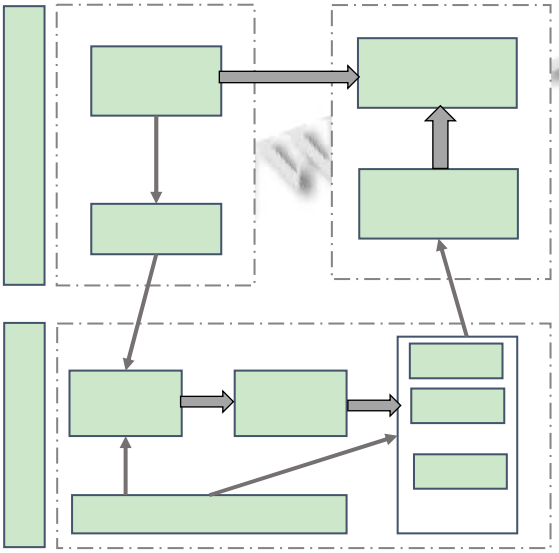

图 3 基于日志的变化数据捕获

基于日志的变化数据捕获需要访问关系型数据库的事务日志, 并解析日志产生对应的更新 SQL 操作. 但是事务日志通常以二进制的形式存在, 如果没有官方文档, 我们很难理解事务日志的内容. 本文利用开源工具 open-replicator[14]实现基于日志的变化数据捕获, open-replicator 可以高效地解析 Mysql 的二进制日志, 并实时产生监听事件.

3.1.3 变化数据捕获机制比较

表 1 从应用条件、编程代价、性能, 移植性等方面比较了基于触发器的变化数据捕获机制和基于日志的变化数据捕获机制.

表 1 两种不同变化数据捕获机制对比

比较内容	基于触发器	基于日志
应用条件	支持触发器	有事务日志
编程代价	中等(设计触发器)	高(需解析特定数据库的日志)
移植性	较强(触发器语法较为通用)	较低(不同厂商日志格式不一样)
性能	较低(触发器给数据库带来额外性能负担)	较高(读取日志即可, 对数据库性能基本不影响)

3.2 数据模型转换方法

分布式缓存以 key/value 形式存储数据, 有利于缓存节点的横向扩展, 其中 key 和 value 均为数据对象, 而在关系型数据库中, 数据以表的形式进行存储, 因此要实现数据库向分布式缓存数据的自动同步, 首先要实现关系型数据模型向 key/value 数据模型的自动转换, 图 4 是数据模型转换实例.

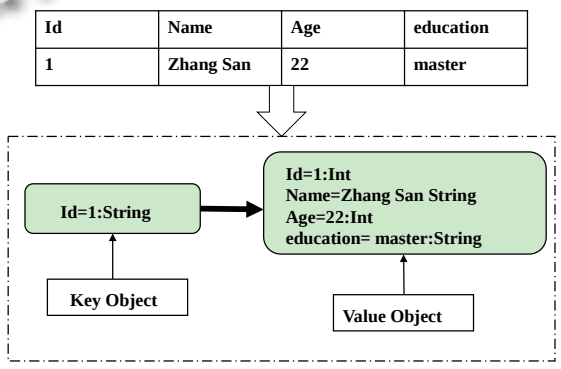

图 4 数据模型转换实例

(1) 分布式缓存 key 的生成方法

生成分布式缓存的 key, 首先要考虑两个问题: ① key 的唯一性: 由于分布式缓存一般以 Map 的结构存

在, key 的唯一性是 Map 存储数据的必要条件; ②key 的易用性: 分布式缓存中, 会计算 key 的哈希值从而确定数据的具体分布, 一个合理的 key 需保障哈希值的易计算, 同时, 分布式缓存的数据读取都以 key 为基础, 如果 key 的生成过于复杂, 在数据读取时都会带来不必要的性能开销. 为了保证 key 的唯一性和易用性, 本文中的 key 都以字符串的形式存在, 对于数据库中的单一主键情形, 直接选取数据库表中的主键作为数据对象的 key, 对于数据库表中的多主键情况, 将多列主键通过特殊间隔符组合拼接, 生成单一的 key, 对数据库表中没有明确指定主键的情况, 采用主键自增方式, 为每个 value 对象维护一个整数自增变量作为对应的 key 值.

(2) value 的生成方法

生成分布式缓存的 value, 重点需考虑的是 value 的通用性. 由于关系型数据库中不同表的数据含有不同的结构, 如何将不同表数据统一生成 Map 中的 value, 是本文重点考虑的问题. 本文将一张数据库表映射转换成一个 Map, 数据库表中的一条记录对应 Map 中的一组 key/value 键值对. 通过数据库表的元信息为每个表动态生成一个数据对象类, 类中属性的类型和关系型数据库属性的类型一一对应, 所有数据对象类含有同一父类, 这样, 为数据库表的每条记录生成的每个对象实例, 都能存储到同一结构的 Map 中.

(3) Map 的索引

关系型数据库中的索引信息是提供高效数据访问的正确手段. 本文为了将关系型数据库的索引信息同步到分布式缓存, 专门设计了索引管理器. 在数据同步到缓存之前, 首先会利用数据库表的元信息创建对应的分布式 Map, 同时会提取数据库列索引信息, 在 Map 中加入对应的索引. 在基于索引的查询过程中, 加入索引的 Map 可以快速寻址到对应的 value 对象.

3.3 SQL 翻译引擎

通过变化数据捕获机制获得的监听事件往往以 SQL 的形式存在, 而分布式缓存的操作是基于 key/value 存储的操作, SQL 翻译引擎负责将 SQL 翻译成基于 key/value 存储的可执行序列.

对数据库的更新主要来源于 Insert, Delete, Update 三类语句, 本文主要翻译这三类语句. SQL 作为一种结构化查询语言, 具有复杂的语法和完备的事务能力, 使用 key/value 存储结构难以完整兼容所有 SQL, 并且对于复杂的嵌套、统计等语法效率会很低下. 本文目前支持的 SQL 语法如下:

(1) INSERT INTO <表名>[(<属性列 1>[, <属性列 2>…])] VALUES (<常量 1>[, <常量 2>…]);

(2) DELETE FROM <表名> [WHERE <条件>];

(3) UPDATE <表名> SET <列名>=<表达式>[, <列名>=<表达式>…] [WHERE <条件>];

其中, 表达式支持常量的任意算术运算组成的表达式, 条件支持 BETWEEN, IN, LIKE, EXISTS, IS NULL, 布尔条件表达式, 等式表达式等.

为了在 key-value 存储系统上执行 SQL, 本文定义了 3 种谓词: 基本谓词(basic_predicate)、关系谓词(relation_predicate)、执行谓词(execute_predicate), 如表 2 所示. 基本谓词表示查询条件, 如大于, like, exists 等, 每个基本谓词是一系列基本 k-v 操作的封装, 可直接在分布式缓存中执行; 关系谓词表示多个谓词之间的与/或关系; 执行谓词分成插入谓词(insert_predicate)、更新谓词(update_predicate)、删除谓词(delete_predicate), 表示三种不同的基于键值对的数据更新操作.

表 2　谓词定义

谓词	定义
basic_predicate	<attribute> =, !=, <, <=, >, >= <value> <attribute> [NOT] LIKE <value> <attribute> [NOT] IN (val1, val2,…) ……
relation_predicate	<expression> AND <exp> OR <exp> …
execute_predicate	insert_predicate, update_predicate, delete_predicate
insert_predicate	(table_name, attribute_one, value_one, attribute_two, value_two, ...)
update_predicate	(table_name, valueSet, attribute_one, value_one, attribute_two...)
delete_predicate	(table_name, valueSet)

在增删改三类数据更新请求中, 更新语句的解析与执行相对来说较为复杂, 算法 1 描述了更新语句的解析执行过程. 首先, SQL 解析后会生成一颗抽象语法树, 同时生成关系谓词列表(relation_predicate_list), 然后从关系谓词列表中顺序提取基本谓词, 依据基本谓词可从分布式缓存中获取对应的 Value 对象: map[table].getValue(basic_predicate), 再依据关系谓词中的逻辑关系和基本谓词获得的 Value 对象, 获得更

新语句过程中的查询结果，更新谓词会对该查询结果对应的数据进行更新.

算法 1　更新语句执行算法

```
Input: updateSentence;
Output: updateRows;
Begin
    syntaxTree ← parse(updateSentence);
    tableName ← syntaxTree.getTable();
    updateItem ← syntaxTree.getItem();
    relationList ← syntaxTree.getPredicate();
    valueSet ← NULL
    while !relationList.isEmpty()
        relation ← relationList.next();
        basicPredicate ← relation.getPredicate();
        value ← getValue(basicPredicate);
        valueSet←updateValueSet (relation, value,
valueSet);
    executePredicat←newUpdatePredicate(tableNa
me, updateItem, valueSet);
    updateRows ← executePredicate.execute();
End
```

4　实验与分析

本文从两个方面来对比基于触发器的变化数据捕获机制和基于日志的变化数据捕获机制，一是比较不同的变化数据捕获机制对数据库性能的影响程度，二是比较不同的变化数据捕获机制在保证分布式缓存一致性时，不一致窗口的大小.

4.1 实验设计

本文使用 TPC-W[15]基准中的关键业务对比验证基于触发器的变化数据捕获和基于日志的变化数据捕获. TPC-W 是一款以真实电子商务应用为用例的测试基准，可以模拟用户访问电子商务图书网站时的查询、购买等行为，包括根据查询书籍，用户注册，订单管理等. 本文选取 TPC-W 基准中与订单管理相关的关键 SQL 语句来进行测试，观察变化数据捕获的性能特点，选取的 SQL 语句为：INSERT INTO order_line (ol_id, ol_o_id, ol_i_id, ol_qty, ol_discount, ol_comments) VALUES (?, ?, ?, ?, ?, ?), , 记为 SQLX，SQLX 的参数使用随机值.

实验的负载发生端使用 YCSB[16]性能测试工具，数据库采用 MySQL，对比测试不同变化数据捕获技术应用到分布式缓存时，给数据库带来的性能影响和缓存不一致窗口的大小. 实验环境的软硬件配置如表 3 所示.

表 3　实验环境软硬件配置

硬件	Inter(R) Core™ i5-4300M CPU@2.60GHzs 8GB Memroy
操作系统	Windows 8.1 x64
数据库	MySQL 5.1

4.2 实验结果与分析

(1) 比较两种变化数据捕获机制下的数据库性能

本次实验中，每个线程执行 SQLX 10000 次，随机向数据库插入 10000 条记录，通过改变线程数量，从而改变对数据库的压力，分别测试线程数为 1,2,5,10 时，每个线程向数据库插入 10000 条记录所需要的时间，即数据库的响应时间. 实验结果如图 5 所示，其中，横轴代表线程的数量，纵轴代表每个线程插入 10000 条记录的平均时间，即数据库的响应时间. 实验结果表明，随着线程数的增大，数据库的响应时间逐渐增大. 但是，由于触发器对数据库性能的影响，在基于触发器的变化数据捕获下，数据库的响应时间增大更明显，基于日志的变化数据捕获下的数据库性能是基于触发器的变化数据捕获下数据库性能的 11 到 21 倍.

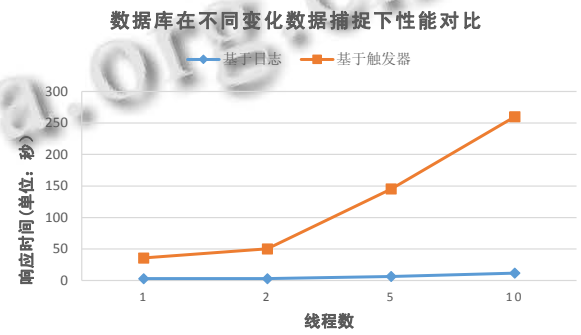

图 5　不同变化数据捕获机制下数据库性能对比

(2) 比较两种变化数据捕获机制下的缓存一致性

根据 CAP 理论，在一个分布式系统中，一致性、可用性和分区容忍性三者不可得兼. 本文中分布式缓存的一致性不是强一致性，数据库数据更新和缓存数据同步更新之间存在一定的不一致窗口，不一致窗口的大小是衡量系统性能的一个重要指标.

本次实验中, 单线程执行 SQLX 10000 次, 随机向数据库插入 10000 条记录, 通过改变两次 SQL 执行之间的间隔时间来改变数据的更新频率, SQL 间隔执行间隔时间越长, 数据更新频率越低. 分别测试 SQL 执行间隔时间为 0,1,2,5,10 毫秒时, 缓存捕获到最新数据和数据库插入数据间的不一致时间窗口. 实验结果如图 6 所示, 其中, 横轴代表两次 SQL 执行的时间间隔, 纵轴代表缓存与数据库之间的数据不一致时间窗口. 从图 6 可以看到如下现象: (1)数据更新频率越低, 缓存的一致性效果越好. 由图可知, 随着间隔时间的增大, 也即数据更新频率减小, 对变化数据捕获模块的压力相对也减小, 从而在更短的响应时间内将数据同步到缓存. (2)基于日志的变化数据捕获相比基于触发器的变化数据捕获有更好的一致性效果. 由图可知, 在不同的时间间隔下, 基于日志的变化数据捕获的不一致窗口都要明显小于基于触发器的变化数据捕获.

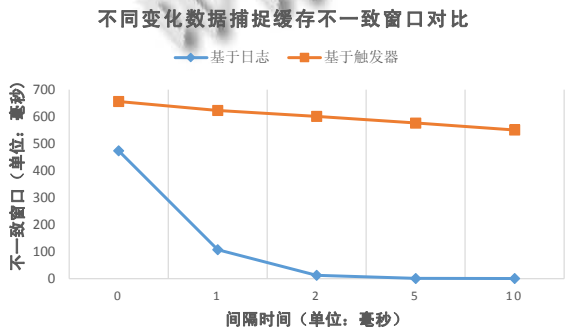

图 6　不同变化数据捕获机制下缓存一致性对比

5　总结与展望

本文首先分析了第三方应用不能感知分布式缓存时存在的过时缓存问题, 然后提出了一种基于数据捕获机制的缓存一致性策略, 主要包括面向分布式缓存的两种变化数据捕获机制, 数据模型的自动转换方法和 SQL 翻译引擎. 通过实验对比验证了基于日志的变化数据捕获技术在数据库性能和缓存一致性效果方面的性能优势, 但是基于触发器的变化数据捕获技术有其通用性, 易用性, 编程代价低等优势.

本文中基于变化数据捕获机制的缓存一致性策略的设计与实现并不完善, 需要在未来工作中针对以下方面进行研究与改进: (1)由于关系型数据模型的复杂性, 关系型数据的外键约束等尚不支持; (2)运行时动态对数据表结构的修改并不能实时映射到分布式缓存数据结构.

参考文献

1 Dreibholz T, Rathgeb E P. On the performance of reliable server pooling systems. The IEEE Conference on Local Computer Networks, 2005. 30th Anniversary. IEEE. 2005. 200–208.

2 Memcached. http://memcached.org/. [2016-03-29].

3 Redis. http://redis.io/. [2016-03-29].

4 Hazelcast. http://hazelcast.org/. [2016-03-29].

5 WebSphere Extreme Scale. http://www-03.ibm.com/software/products/en/websphere-extreme-scale/. [2016-03-29].

6 Eccles M. Pragmatic Development of Service Based Real-Time Change Data Capture[Thesis]. Aston University, 2013.

7 徐富亮,周祖德.变化数据捕获技术研究.武汉理工大学学报:信息与管理工程版,2009,31(5):740–743.

8 林子雨,杨冬青,宋国杰,等.实时主动数据仓库中的变化数据捕获研究综述.计算机研究与发展,2007,44(z3):447–451.

9 Rocha RLA, Cardoso LF, de Souza JM. Performance tests in data warehousing ETLM process for detection of changes in data origin. Data Warehousing and Knowledge Discovery. Springer Berlin Heidelberg, 2003: 129–139.

10 Shi JG, Bao YB, Leng FL, et al. Study on log-based change data capture and handling mechanism in real-time data warehouse. 2008 International Conference on Computer Science and Software Engineering. IEEE. 2008, 4. 478–481.

11 Amza C, Soundararajan G, Cecchet E. Transparent caching with strong consistency in dynamic content web sites. International Conference on Supercomputing. 2005. 264–273.

12 Ghandeharizadeh S, Yap J, Nguyen H. Strong consistency in cache augmented SQL systems. Proc. of the 15th International Middleware Conference. ACM. 2014. 181–192.

13 Gupta P, Zeldovich N, Madden S. A trigger-based middleware cache for ORMs. Acm/ifip/usenix International Conference on MIDDLEWARE. Springer-Verlag. 2011. 329−349.

14 Open-replicator. https://github.com/whitesock/open- replicator. [2016-03-29].

15 TPC-W. http://www.tpc.org/tpcw/. [2016-03-29].

16 YCSB. https://github.com/brianfrankcooper/YCSB. [2016-03-29].