# SQL Query to Trigger Translation: A Novel Transparent Consistency Technique for Cache Augmented SQL Systems

*Shahram Ghandeharizadeh, Jason Yap*

Database Laboratory Technical Report 2014-08

Computer Science Department, USC

Los Angeles, California 90089-0781

May 23, 2014

**Abstract**

Organizations enhance the velocity of simple operations that read and write a small amount of data from big data by extending a SQL system with a key-value store (KVS). The resulting system is suitable for workloads that issue simple operations and exhibit a high read to write ratio, e.g., interactive social networking actions. A popular distributed in-memory KVS is memcached in use by organizations such as Facebook and YouTube.

This study presents SQL query to trigger translation (SQLTrig) as a novel transparent consistency technique that maintains the key-value pairs of the KVS consistent with the tabular data in the relational database management system (RDBMS). SQLTrig provides physical data independence, hiding the representation of data (either as rows of a table or key-value pairs) from the application developers and their software. Software developers are provided with the SQL query language and observe the performance enhancements of a KVS without authoring additional software. This simplifies software complexity to expedite its development life cycle. We present performance results to show SQLTrig provides a performance comparable to a non-transparent consistency technique where a developer extends the application software to maintain key-value pairs consistent with the tabular data.

## A Introduction

Consider the following simple concept. A novel JDBC driver for a SQL solution that enhances overall performance of an application by a factor of two or more. It does not require a re-write of the application software or a re-design of its database. Instead, it enhances performance by looking up the result of SQL queries instead of processing them. It is ideal for applications that exhibit a high read to write ratio, e.g., social networking [18]. This is because query result look up is faster and more efficient than query processing [26]. This idea (and not the concept) is in use by many popular Internet destinations including Facebook, Wikipedia, YouTube and others. In [18], Facebook describes how it employs a distributed in-memory key-value store, KVS, named memcached in combination with the MySQL relational database management system (RDBMS) to implement its interactive social networking actions. However, they require developer provided software to maintain the key-value pairs consistent with the tabular data in the presence of updates.

The simple concept of a JDBC driver that embodies a KVS is not available today. This is due to a lack of *physical data independence* that hides the details of the tabular data in the RDBMS and the corresponding key-value pairs in the KVS from the application developers and their software. Software developers must perform global reasoning on what RDBMS changes impact which key-value pairs and how updates to the RDBMS should propagate to the KVS. Physical data independence eliminates this reasoning to reduce the complexity of application software to expedite their development life cycle: implementation, testing, and debugging. An abler of physical data independence is a transparent KVS consistency technique. Another is a technique that prevents undesirable race conditions that cause the KVS and the RDBMS to diverge and produce stale data, e.g., Gumball technique [20] and Leases [18, 23]. While both are important, the former has received little attention and is the focus of this study.

This study introduces a novel transparent KVS consistency technique named SQL Query To Trigger translation, *SQLTrig* for short. It is embodied by a wrapper[1] that provides the JDBC interface of an RDBMS. It intercepts each SQL query instance issued by an application and looks up its result set in the KVS. If it is found then its result set is returned to the application for processing. Otherwise, it executes the query instance using the RDBMS to obtain its result set. Next, it computes the query template for this instance. (Figure 1 illustrates the difference between a query template and a query instance, see Section C for details.) If this is a repeated query template with cached query instances then SQLTrig proceeds to cache the query instance and its result set as a key-value pair in the KVS. Otherwise, it authors one or more triggers for this query template and registers them with the RDBMS. Updates to the tabular data in the RDBMS invoke the triggers to invalidate (delete) the corresponding query instances (keys) whose result sets (values) have changed and might be KVS resident.

While registration of triggers is an expensive operation, it is performed once per query template. Typically, an application consists of tens to a few hundred query templates and billions (if not trillions) of query instances. SQLTrig authors and registers triggers per query template once it encounters an instance of the query template for the very first time. Subsequent instances of the query template do not cause SQLTrig to either re-author or re-register triggers.

SQLTrig supports the following SQL query types with no change to either the RDBMS or the KVS server infrastructure: simple aggregates such as count and sum, simple exact-match predicates, join predicates, and arbitrary combinations of these predicates using the logical *and* connective. When the KVS infrastructure is

[1]In addition to SQLTrig, the wrapper includes the JDBC driver of an RDBMS and the client component of a KVS, see Section D.1.

extended with additional index structures, SQLTrig supports range selection predicates and predicates using the logical *or* connective. SQLTrig does not support complex queries (e.g., OLAP decision support style queries) such as non-equi join predicates and nested SQL queries at the time of this writing.

We designed and implemented SQLTrig based on the following principles. First, given the complexity of today's RDBMS software [33, 34], SQLTrig requires no software changes to its target RDBMS as long as it implements triggers. Most, if not almost all, RDBMSs implement triggers. Experimental results of Section E show use of triggers has minimal impact on system performance. Second, SQLTrig deletes (invalidates) key-value pairs impacted by an RDBMS update command. An alternative is to either refresh [12] (re-compute and insert) or incrementally update [25] the impacted keys. These alternatives are more complex and constitutes our future research direction, see Section G.

## B Overview

SQLTrig is designed for applications with workloads that read and write a small amount of data from big data, e.g., interactive social networking actions. An application consumes most if not the entire result set produced by the RDBMS. Hence, building an index on a result set is not useful. Operations with these characteristics are termed *simple* [34, 10].

A query instance is a simple read operation and an application may issue a very large number (billions if not trillions) of them. Example simple operations from the social networking applications are detailed in Section E.1 and include queries such as show the profile of a member, list pending friend invitations of a member, and others.

SQLTrig is not a substitute for materialized views [24] used to process *complex* operations in support of On Line Analytical Processing (OLAP) for decision support. These applications may author a few (tens of) materialized views consisting of a large number of rows. Queries retrieve a few rows from the views and these views are indexed to facilitate efficient retrieval of the referenced rows. Moreover, views are updated incrementally due to the expense of computing the view. The periodic technique to update views may produce a substantial amount of stale data. This and other limitations of materialized views for simple operations are discussed in [7], motivating the use of a KVS and its key-value pairs for simple operations. Of course, one may use a KVS and the proposed SQLTrig to maintain simple operations that query materialized view.

This study assumes the KVS is maintained consistent with the RDBMS using an invalidation technique[2]. With this technique, the KVS deletes a query (key) whose result-set (value) is impacted by an RDBMS update. A subsequent KVS look up for the query observes a miss, issues one or more RDBMS queries to

[2]Alternatives to invalidation are either refresh [12] or incremental update [25] and constitute our future research direction, see Section G.

```
SELECT m.userid, m.email, m.profileImage
FROM   Members m, Frds f
WHERE  f.frdID1=1 and m.userid=f.frdID2
```

1.a) Query instance

```
SELECT m.userid, m.email, m.profileImage
FROM   Members m, Frds f
WHERE  f.frdID1=? and m.userid=f.frdID2
```

1.b) Query template

Figure 1: A query instance to retrieve the friends of Member with userid=1 and its corresponding query template.

compute a new key-value pair, and inserts this key-value pair in the KVS. To avoid race conditions that cause the KVS to produce stale data, we assume the use of either the Gumball technique [20] or Leases [18].

The query to trigger translation is the key contribution of this paper and described in the next section. Section D presents a prototype of SQLTrig using an industrial strength RDBMS and a modified Twemcache server version 2.5.3. Section E presents performance results from this prototype using a social networking benchmark named BG [6]. Obtained results show SQLTrig and its use of RDBMS triggers provides comparable performance with an implementation that requires the application software to invalidate key-value pairs in the presence of RDBMS updates. We present related work in Section F. Brief future research directions are outlined in Section G.

## C SQL Query to Trigger Translation

The input to the SQLTrig's query to translator is a query *instance* issued by an application. The translator consists of the following two components: 1) A query template generator, QTGen: The input to this component is the query instance and its output is a query template, and 2) A trigger generator, TrigGen: Its input is a query template and its output is a set of triggers. In the following, we start by describing what is a trigger. Next, we describe QTGen and TrigGen in turn.

A trigger is a procedure registered for execution with the RDBMS. It is specified on a table, say R, to execute when a row is either inserted in R, deleted from R, or updated. With an insert (delete), the trigger is provided with the record to be inserted (deleted). With an update, the trigger is provided with both the new and old version of the record to be updated. A trigger defined on a Table R may not query Table R because this table is in the process of being updated. TrigGen respects this constraint for all its authored triggers.

Conceptually, TrigGen authors software to identify the query template as a trigger. This software includes additional logic to instantiate the query template everytime the trigger executes. It uses the inserted/deleted/updated row to compute those query instances whose result sets have changed and to invalidate (delete) them from the KVS.

SQLTrig assumes triggers execute synchronously, returning an error code when it fails to delete a key-value pair, e.g., due to intermittent network connectivity. At run time, such failures cause the RDBMS transaction to abort, leaving the KVS and the RDBMS consistent with one another.

To compute the query template of a query instance, QTGen parses the SQL query to identify its selection predicates. These predicates appears in the qualification list (where clause) of the query and might be connected using Boolean logic (and, or, not). A predicate may compare an attribute of a table (Member.id) with a constant (say 47645) using a comparison operator ($=$, $\leq$, $\geq$, $<$, $>$, $\neq$). QTGen replaces a constant with a wild card to compute the query template. Figure 1 shows a query instance and its corresponding query template produced by QTGen.

To translate a parsed query template into triggers, TrigGen identifies the following six types of SQL queries with a "where" clause consisting of:

1. One exact-match selection[3] predicate: TrigGen authors triggers that produce the query instance whose result has changed. See Section C.1 for details.

2. Several exact-match selection predicates connected using the logical *and*: Identical to the discussion of queries with one exact-match selection predicate, see Section C.1 for details.

3. One or more join predicates and one or more exact-match selection[4] predicates connected using the logical *and*: TrigGen authors triggers that generate the query instance (key) whose result (value) has changed. The authored trigger may query one or more of the tables in the "from" clause of the query. A trigger does not query its own table that is in the process of being modified. See Section C.2 for details.

4. One or more range selection predicate connected using the logical *and*: TrigGen requires the KVS to maintain an R-Tree to identify the different query instances (keys) whose results (values) reside in the cache. The dimensionality of the R-Tree is dictated by the number of unique attributes referenced by the different range selection predicates. TrigGen authors triggers that produce a multi-dimensional value that probes the R-Tree to identify the impacted query instances (keys) that are subsequently deleted by the KVS.

5. One or more selection and join predicates connected using the logical *or*: TrigGen uses Boolean logic to break the original query instance into query fragments, each with a distinct where clause resembling one of the previous four cases. Conceptually, the union of the result of the query fragments computes the result of the original query. TrigGen requires the KVS to maintain a hash table that maps each query fragment to the original query (key). Next, it authors triggers to generate a query fragment based on the provided four classifications. When an RDBMS update invokes a trigger, it invokes a KVS method that consumes the query fragment to probe the hash table to identify the key (query instance with the logical *or*) whose value (result set) must be invalidated.

[3]An exact match is a comparison of an indexed tuple variable with a constant using an equality predicate, e.g., userid="654".

[4]Recall that queries purely with join predicates are not appropriate for use with SQLTrig because they are decision support style queries. SQLTrig targets query instances that are selective and large in number.

6. One of the previous five cases with the “select” clause of the query using an aggregate such as “count” or “sum”: TrigGen employs the translation process of the above classification with one difference. Triggers are authored intelligently based on the aggregate. For example, a query that counts the number of rows should not be invalidated if one record of its referenced table is updated.

Below, we provide details of how TrigGen supports each class of queries in turn.

## C.1 Exact match selection predicates

Consider the following query with a qualification list consisting of one exact-match selection predicate:

SELECT attr$_1$, attr$_2$, ..., attr$_n$
FROM R
WHERE attr$_{n+1}$=$C_1$

Its relational algebra equivalent is: $\pi_{attr_1,...,attr_n}(\sigma_{attr_{n+1}=C_1}(R))$. The translation process to generate triggers is as follows. With either an insertion or deletion of a row $r$, the trigger is authored to embody the query template and replace the wild card with the value of attr$_{n+1}$ of the impacted row r, i.e., r.attr$_{n+1}$. The resulting query instance is the key whose value has changed. The KVS deletes this key.

With an update, TrigGen authors the trigger to execute before update to a row $r$ of Table R. Thus, the trigger may access the attribute value of the old and new version of row $r$. If r.attr$_{n+1}$ is being modified from $C_1$ to $C_{new}$ then the result of two different query instances have changed. TrigGen authors an ”If($C_1$ != $C_{new}$)/Else” statement to detect this by comparing the old ($C_1$) with the new value ($C_{new}$). When these two values are not equal, additional code is provided to generate two query instances by replacing the wild card of the query template with two different values: old and new values of r.attr$_{n+1}$, i.e., $C_1$ and $C_{new}$. The KVS deletes both keys.

When r.attr$_{n+1}$ is not modified, at least one of the attributes in the projection list, i.e., r.attr$_1$, r.attr$_2$, ..., r.attr$_n$, must be modified in order for the trigger to identify an impacted query (key). This is constructed by replacing the wild card of the query template with the value of r.attr$_{n+1}$.

In its most general form, the query’s qualification list (where clause) may consist of $k$ predicates connected using the logical *and*, attr$_{n+1}$=$C_1$ AND attr$_{n+2}$=$C_2$ AND ... AND attr$_{n+k}$ = $C_k$. Extension of the insert and delete trigger authoring process is trivial: $k$ wild cards of the query template are replaced with the respective attribute values (attr$_{n+1}$ to attr$_{n+k}$) of the impacted row $r$. With an update, every time the value of one or more of the $k$ attributes (attr$_{n+1}$ to attr$_{n+k}$) of a row $r$ changes then two query instances are identified for invalidation. They are constructed by replacing the wild card of the query template with the old and new value of the $k$ attributes of the impacted row $r$. KVS deletes these keys.

With a query instance converted into its algebraic equivalent, TrigGen performs simple string manipulations (change to uppercase and removal of extra spaces) and sorts the attribute names referenced by its project (and select operator) prior to constructing a query template and authoring triggers. This ensures the same query that is slightly different and issued by different methods of an application produce identical templates and triggers.

## C.2 Equi-join predicates with one or more exact-match selection predicates

To describe TrigGen's authoring of triggers for queries with a join predicate, consider the following query:

SELECT R.attr$_1$, R.attr$_2$, ..., R.attr$_n$
FROM R, S
WHERE S.$attr_j$=C1 and R.$attr_{n+1}$=S.$attr_i$

where tables R and S might be Members and Frds tables and C1 is the value 1 in Figure 1.a. Next, TrigGen authors two sets of triggers, one for Table $R$ and the other for Table $S$. The set of triggers is different for each table due the presence of the exact-match selection predicate referencing Table S. Both compute the query instance (key) whose result (value) has changed and should be invalidated. Below, we describe authoring of triggers for each table in turn. Subsequently, we generalize the discussion for complex "where" clauses consisting of an arbitrary number of join and exact-match selection predicates.

TrigGen authors triggers that handle insert and delete of a row $s$ from the table referenced by a selection predicate (S) as follows. It uses the query template and replaces its wild card with the value of the attribute referenced by the selection predicate, $s._{attr_j}$. With an update of row $s$, it authors the trigger to replace the wild card with the old and the new value of $s._{attr_j}$, identifying two query instances (keys) whose results (values) have changed. The KVS deletes these keys.

With the table that participates in the join clause and is not referenced by the selection predicate, Table R, TrigGen authors triggers that handle insert and deletion of a row $r$ as follows. It authors code to perform an exact-match look up of table $S$ by transforming the equi-join predicate to an exact-match lookup: S.$attr_i$= r.$attr_{n+1}$. Note that r.$attr_{n+1}$ is a constant as $r$ is a specific row of Table R (in the process of being inserted or deleted). For each matching record $s$, the authored trigger employs the value of $s.attr_i$ to replace as the value of the wild card in the query template. This query instance (key) should be invalidated because its result (value) has changed.

Figure 2 shows the pseudo-code for how TrigGen processes a "where" clause consisting of an arbitrary number of equi-join and exact-match selection predicates. It groups predicates based on whether they are equi-join or exact-match selection predicates. Next, it merges the exact match predicate that reference the same table into one. Subsequently, it generates triggers for those tables referenced by the exact match predicates, Step 5. Finally, it generates triggers for each table referenced by the join predicate, Step 6.

```
1.  Let T = query template of the query instance
2.  Let {P} = Selection predicates
3.  Let {J} = Join predicates
4.  Combine those selection predicates in {P} referencing the same table into one
5.  For each selection predicate p in {P} do:
    5.1.  Author code to use the value of impacted row of table referenced by p
    5.2.  Let {Q} = {P} - p
    5.3.  For each q in {Q} do
          5.2.1.  Author code to use all elements of {J} to lookup the
                  value of attribute referenced by q, q.attr
    5.4.  Author code to use the values computed in the for loop with {Q}
          to replace the wild cards in T
    5.5.  Author code to invalidate the resulting query instance
6.  For each Table S referenced by a join predicate j in {J} do:
    6.1.  Let {Q} = {J} - j
    6.2.  For each q in {Q} do
          6.2.1.  Let {R} denote the table referenced by q
          6.2.1.  For each table in {R}
                  6.2.1.1. Author code that for every deleted/inserted row r,
                           its joining attribute value is used to look up the
                           record in the other table
                  6.2.1.2. Author code to use the resulting row to join with other
                           tables in {Q} - q to compute the value of attributes
                           used in the selection predicates {P}
                  6.2.1.3. Author code to use the computed value to instantiate
                           the wild cards of the query template
                  6.2.1.4. Author code to invalidate the resulting query instnace
```

Figure 2: Pseudo code for processing join predicates.

## C.3 Range predicates

TrigGen constructs one R-Tree for each query template whose where clause references a range selection predicate. A dimension of the R-Tree corresponds to a referenced attribute. A query instance is a $k$ dimensional polygon in the R-Tree (corresponding to its query template) and whose results are used to compute a key-value pair.

The R-Tree is maintained by the KVS and TrigGen authors triggers to generate a $k$ dimensional value. These probe the R-Tree to identify matching polygons. Each polygon identifies queries (keys) whose result sets (values) have changed. The KVS deletes these keys.

Consider the following query instance with a range predicate referencing two different attributes of Table R:

SELECT attr$_1$, attr$_2$, ..., attr$_n$
FROM R
WHERE (attr$_{n+1} > C_1$ AND attr$_{n+1} < C_2$) AND
(attr$_{n+2} > C_3$ and attr$_{n+2} < C_4$)

The selection operator references two different attributes of $R$. TrigGen directs the KVS to create and maintain a two dimensional R-Tree with each dimension corresponding to a unique attribute, attr$_{n+1}$ and attr$_{n+2}$. The two ranges specified by the query, $\{(C_1,C_2), (C_3,C_4)\}$, are termed internal keys, IntKeys. They define a rectangle that is inserted in the R-Tree to identify this query instance (key).

Conceptually, TrigGen authors triggers by changing each range predicate to an exact match predicate and employs discussions of Section C.1 with the following modification: The trigger tags its produced value with a literal (say "Range") that designates it for a range predicate, table name (R), and the referenced column names (attr$_{n+1}$, attr$_{n+2}$). Thus, an insert of a row $r$, causes the trigger to delete the two dimensional value $\{$r.attr$_{n+1}$, r.attr$_{n+2}\}$ concatenated with the aforementioned tokens. TrigGen parses this to detect a unique R-Tree. The KVS uses the two dimensional value to look up this R-Tree for the rectangles containing it. Each such polygon identifies one or more queries (keys) whose result sets (values) are no longer valid. The KVS deletes these keys.

TrigGen supports query templates with alternative range comparison operators ($\leq$, $\geq$, $<$, and $>$) by constructing R-Trees that maintain either open or closed intervals for a dimension. With our example query and others similar to it, the authored triggers do not differentiate between the different comparison operators, producing a two dimensional value always.

TrigGen currently supports only value driven range queries, where an attribute in a range selection predicate is compared with a constant. In other words, it does not support range predicates such as R.sal $<$ R.age $\times$ 100.

## C.4 Logical "or" Connectivity

With queries whose "where" clause uses the logical *or* connectivity, TrigGen employs the distributivity property[5] of propositional logic to construct several sub-queries. The "where" clause of each sub-query uses the logical *and* connectivity and is different for each sub-query. Logically, the union of the results of these sub-queries computes the same result as the original query.

TrigGen requires the KVS to maintain a hash table that associates each sub-query instance (an internal key, IntKey) with the original query instance (key). This hash table is identified by a unique name and is specific to this query template, i.e., it is popuated by the large number of instances of this query template. Subsequently, TrigGen employs the discussions of the previous sections to translate each query into a set of triggers with one difference. The trigger generates both a sub-query instance (IntKey) and the name of the hash table for its query template. The KVS uses the hash table name along with the sub-query instance (IntKey) to identify the query instance (key) whose result set (value) must be invalidated.

As an example, consider the following query:

```
SELECT userid
FROM   Friends
WHERE  status='2' AND
       ( userid='869' OR friendid='869' )
```

[5]Distribution of conjunction (and) over disjunction (or).

Using the distributivity property of propositional logic, TrigGen constructs the following two sub-queries:

1. $\pi_{userid}(\sigma_{(status='2')\ and\ (userid=869)}(Friends))$, and
2. $\pi_{userid}(\sigma_{(status='2')\ and\ (friendid=869)}(Friends))$.

TrigGen directs the KVS to maintain a hash table with a unique name, say X, that maps these two query instances (IntKeys) to the original query (key). Next, it uses the discussions of Section C.1 to author triggers for each sub-query. Once activated, these triggers identify a sub-query string along with the mapping table X. The KVS probes mapping table X with the sub-query string (IntKey) to identify the original query (key) whose result set (value) has been invalidated. In a final step, the KVS deletes this key.

### C.5 Simple Aggregates

Aggregates such as count are a common query with social networking applications. An example query is one that counts the number of friends for a given user:

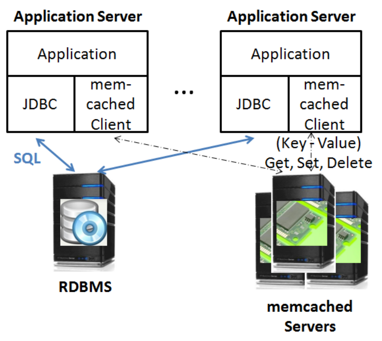
```
SELECT count(f.friendid)
FROM   Friends f
WHERE  f.userid='869'
```

TrigGen authors triggers by re-writing their target list to eliminate the aggregate. Subsequently, it uses the discussions of the previous 3 sections to author triggers. For example, with the example query, "count(f.friendid)" is replaced with "f.friendid". With "count(*)", the "*" is replaced with the primary key of the referenced table. TrigGen does recognize the presence of an aggregate and, once the triggers are generated, restores the target list of the query (key) generated to its original aggregate. This ensures the trigger produces the correct key for invalidation.

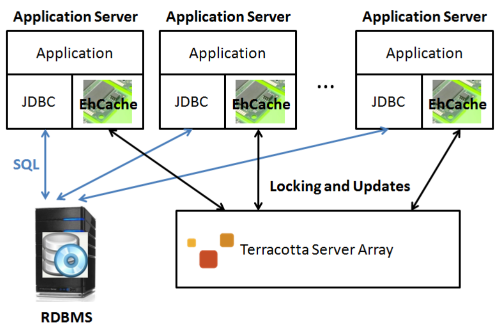
With aggregates that have no where clause, e.g., the sum of all values in a column, TrigGen associates KVS key-value pairs with the name of the reference table and the columns of interest. It authors triggers to generate the table name concatenated with the referenced columns as its output. This invalidates key-value pairs with any change involving those column values on record inserts, deletes and updates. The count aggregate with no qualification list is a special case where the key-value pair is associated with the table name and is invalidated at the granularity of a table change. However, only inserts and deletes generate query instances (keys) as updates do not affect the number of rows.

## D An Implementation

This section provides a high level description of the alternative caching architectures that may use utilize the translator of Section C. Next, it describes a specific implementation shown in Figure 4.



3.a) Client-Server (CS) architecture



3.b) Shared Address Space (SAS) architecture

Figure 3: Alternative CADBMS architectures.

An implementation may supports two different architectures, see Figure 3. With the Shared Address Space (SAS) architecture, the KVS is a library that implements SQLTrig to act as an intermediary between the application and the RDBMS [27, 13, 17, 37]. It provides the optimum read service time by staging content in the same process as the application, eliminating the overhead of serializing key-value pairs, and inter-process and inter-processor communication. It also requires each invalidation to be propagated to all cache instances. Examples of the SAS architecture include Terracotta Ehcache [35] and JBoss Cache [9]. None include a transparent caching technique such as SQLTrig and require developer provided software to maintain key-value pairs consistent with the tabular data in the RDBMS.

With a Client-Server architecture, CS, the cache manager consists of a client and a server component that communicate via message passing [1, 2, 16]. Typically, key-value pairs are partitioned across the KVS server instances. Hence, a key-value invalidation impacts one server instance. The service time of reads with this architecture is worse than SAS because the client component incurs the overhead of communicating with the server that might be running on a different node [21]. An example system is the widely used memcached [30, 18].

A discussion of the two architectures and their tradeoffs is a digression from the main focus of this paper (i.e., SQLTrig) because it must consider issues such as scalability, elasticity, and data availability in the presence of RDBMS and KVS failures. (See [21, 7] for a quantitative comparison of the two architectures.) Instead, we present SQLTrig assuming a simple CS architecture consisting of an industrial strength RDBMS named[6] SQL-X and a modified version of Twitter memcached (Twemcache) Version 2.5.3 [3] that implements SQLTrig. The latter is named the SQLTrig server and its implementation details are provided in Section D.2. The client component is named *SQLTrig client.* It incorporates the JDBC driver of SQL-X and the Whalin memcached client version 2.6.1 [36], see Figure 4. It is described in the next section.

[6]Due to licensing restrictions, we cannot disclose its identity and name it SQL-X.

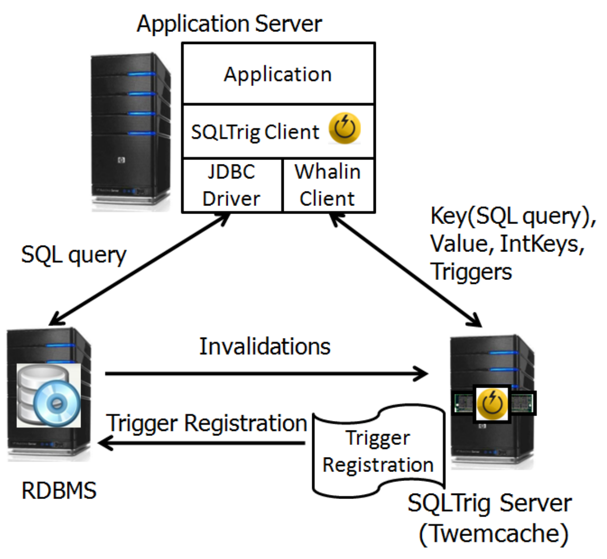


Figure 4: Architecture of a single node SQLTrig implementation and its components.

## D.1 SQLTrig Client

The SQLTrig client is a wrapper that provides the JDBC interface of SQL-X to the application. It employs the JDBC driver of SQL-X to issue queries to SQL-X and the Whalin memcached client version 2.6.1 [36] to issue commands to the SQLTrig Server (extended Twitter Memcached). The client is previewed to all RDBMS queries and update commands issued by the application including those commands that define transaction boundaries. To respect the consistency guarantees implemented by the developer, SQLTrig does not materialize key-value pairs pertaining to queries issued as a part of a multi-statement transaction. With single-statement queries that fall into one of the categories described in Section C, the client looks up the result set (value) of the query (key) in the SQLTrig server (using its Whalin client). If the server provides a value, the client deserializes it into an instance of result set, and provides it to the application for further processing. Moreover, the client constructs the query template of the instance and records the fact that this query template observed a cache hit. (As explained below, this information is used to avoid repeated authoring of triggers.)

When the server reports a cache miss, the client issues the query to the RDBMS to obtain its result set. Next, it converts the query instance to a query template using QTGen, see Section C. If this query template is one with an instance that observed a cache hit then the client (a) skips authoring of triggers for this query instance and (b) inserts the query instance (key) and its serialized result set (value) in the server using the regular set command of Twemcache. Otherwise, the client proceeds to use TrigGen to author triggers for the query template, {Trigs}. If the query includes a range predicate or uses *or* connectivity, there will also be intermediate keys, {IntKeys}. With these, it invokes the SQLT-SETKeys($k_i$, $v_i$, {Trigs},

$\{$IntKeys$\}$). Otherwise, it invokes SQLT-SET($k_i$, $v_i$, $\{$Trigs$\}$). With both $v_i$ is the serialized result set obtained from SQL-X. Both SQLT-SETIK and SQLT-SET invoke their counterparts on the SQLTrig server, see Sections D.2.

The technique used to serialize and deserialize (marshall) the result of SQL queries impacts the overall system performance. We use a custom serialization technique for result sets. This custom implementation is more than four times more compact and more then three times faster than the generic serialization provided by the Java programming language. See Section H for details.

## D.2 SQLTrig Server

The SQLTrig server is implemented using the C++ language and is an extends version of the Twitter memcached 2.5.3, Twemcache [3]. The extensions implement the indexing in support of two new commands: SQLT-SET($k_i$, $v_i$, $\{$Trigs$\}$) and SQLT-SETIK($k_i$, $v_i$, $\{$Trigs$\}$, $\{$IntKeys$\}$). Both are per SQLTrig client specification, see Section D.1.

The SQLTrig server maintains a hash table of the triggers that have been registered with the RDBMS successfully. When the client issues either SQLT-SET or SQLT-SETIK command, the SQLTrig server determines if each trigger in the set $\{$Trigs$\}$ is found in the hash table of the registered triggers. If this is the case, with SQLT-SET, it proceeds to insert the provided $k_i$-$v_i$ in the cache. With SQLT-SETIK, it associates each internal key $IntKey_i$ with the provided key $k_i$ prior to inserting $k_i$-$v_i$. This is in support of queries using either a range selection predicate (see Section C.3), the logical *or* connectivity (see Section C.4), or both.

SQLTrig performs the following two steps in turn. First, for each $IntKey_i$ in the set $\{$IntKey$\}$, it registers $IntKey_i$-$k_i$ with the appropriate index that must be maintained by the KVS, see discussions of Section C.3 and C.4. Next, it inserts $k_i$-$v_i$ into the KVS.

If a trigger in the set $\{$Trigs$\}$ is not found in this hash table, SQLTrig places the trigger in a registration queue and returns without inserting $k_i$-$v_i$ in the KVS, i.e., discards $k_i$-$v_i$ and the provided $\{$IntKeys$\}$. A background trigger registration thread consumes elements of the trigger queue and issues commands to a Trigger Registeration (TR) process to register each trigger with the RDBMS. TR is written using the Java program and uses the JDBC driver of SQL-X to register triggers. Once it registers a trigger successfully, it returns control to the background thread of the SQLTrig server. This thread inserts the trigger in the hash table of registered triggers and proceeds to the next trigger in its queue.

In addition to the delete command of the Twemcache server, the SQLTrig server implements SQLT-DELIK($\{$IntKeys$\}$). The latter is used by those triggers authored for queries using either a range predicate or the logical *or* connectivity. It provides context for the SQLTrig server to lookup each intermediate key using the appropriate index structure to identify the query instance whose results has changed (and delete it). Note that the query instance may or may not reside in the SQLTrig server.

| BG Action | Read Only | 0.1% Write | 1% Write | 10% Write |
|---|---|---|---|---|
| View Profile | 40% | 40% | 40% | 35% |
| List Friends | 5% | 5% | 5% | 5% |
| View Friends Requests | 5% | 5% | 5% | 5% |
| View Top-K Resources | 40% | 40% | 40% | 35% |
| View Comments on Resource | 10% | 9.9% | 9% | 10% |
| Invite Friend | 0 | 0.02% | 0.2% | 2% |
| Accept Friend Request | 0 | 0.02% | 0.2% | 2% |
| Reject Friend Request | 0 | 0.03% | 0.3% | 3% |
| Thaw Friendship | 0 | 0.03% | 0.3% | 3% |

Table 1: Four mixes of social networking actions.

Section C.3 describes R-Trees as an index in support of range queries beacuse they are a good fit and conceptually simple to explain. However, our implementation uses interval trees which is a one-dimnensional R-Tree. With $N$ range selection predicates referencing different attributes, we use one interval tree that points to another interval tree in sequence. The last interval tree points to the query result set. The alphabetical sorting of attribute names dictate which interval tree indexes the other. This implementation is more efficient than R-Tree because R-Trees are a persistent data structure. An in-memory interval tree is sufficient for our purposes because the SQLTrig server (Twemcache) is an in-memory key-value store.

## E An Evaluation

This section employs the BG [6] benchmark to compare the performance of an industrial strength relational database management system named SQL-X with itself and when extended with SQLTrig, see Section D. As a comparison yard stick, we present performance of SQL-X with Twemcache and developer provided software to maintain the key-value pairs consistent with the tabular data. Below, we provide an overview of the BG benchmark. Subsequently, Section E.2 characterizes the queries (keys) and result sets (values) generated by this benchmark. Section E.3 presents the social action rating (SoAR) of the alternative configurations using BG. These results demonstrate SQLTrig enhances the performance of SQL-X by more than two folds while providing physical data independence. This section concludes with an analysis of queries issued by three other benchmarks: RUBiS [11], RUBBoS [31] and TPC-W [15].

### E.1 BG Social Networking Benchmark

BG [6] is a benchmark to quantify performance of a data store for interactive social networking actions and sessions. These actions and sessions either read or write a very small amount of the entire data set. In addition to response time and throughput, BG quantifies the amount of unpredictable data produced by a data store. This metric refers to either stale, inconsistent, or invalid data produced by a data store. This is

Members($\underline{userid}, username, pw, firstname, lastname, job, gender, jdate, ldate, address, email, tel, profileImage,$ $thumbnailImage, \#PendingFriends, \#Friends, \#Resources$)

Friends($\underline{\widehat{frdID}1, \widehat{frdID}2}$)

PdgFrds($\underline{\widehat{inviterID}, \widehat{inviteeID}}$)

Resource($\underline{rid}, \widehat{creatorid}, \widehat{walluserid}, type, body, doc, priority$)

Manipulation($\underline{mid}, modifierid, \widehat{rid}, \widehat{creatorid}, timestamp, type, content$)

Figure 5: SQL-X database design with no images. The underlined attribute(s) denote the primary key of a table. Attributes with a hat denote the indexed attributes.

particularly useful because it enabled us to experimentally verify SQLTrig produces no stale data.

BG computes a Social Action Rating (SoAR) of a data store based on a pre-specified service level agreement (SLA) by manipulating the number of threads (i.e., emulated members) that perform actions simultaneously. SoAR is the maximum system throughput (actions per second) that satisfies the SLA. All SoAR ratings in this paper are established with the following SLA: 95% of requests observe a response time of 100 milliseconds or faster with no unpredictable (stale) data.

Table 1 shows the interactive *actions* of BG which are common to many social networking sites [6]. This table shows the four different workloads that we explore in this study. A read-only workload that performs no write actions and three different mix of read and write actions with the percentage of write actions varying from 0.1% to 10%. The workload of social networking applications is dominated (> 99%) by read actions [4, 18].

All results reported below were obtained using eight nodes with the following specifications: Windows Server 2003 R2 Enterprise x64 bit Edition Service Pack 1, Intel® Core™ i7-2600 CPU 3.4 GHz, 16 GB RAM, Seagate 7200 RPM 1.5TB disk. The BG clients executes on six nodes preventing the benchmarking infrastructure from becoming the bottleneck. Two different nodes host the RDBMS and the KVS (either Twemcache server or SQLTrig server). These nodes communicate using a 1 Gigabit Ethernet switch.

## E.2 BG Actions

Table 1 shows the different BG actions and their queries. We discuss these in turn. The *View Profile* action of BG consists of four SQL queries that retrieve (1) the profile attributes of a member such as her first name, last name, picture, etc., (2) her number of friends, (3) her number of pending friend invitations, and (4) her number of resources. Per schema of Figure 5, the “where” clause of the first SQL query is an exact-match selection predicate referencing the Members table. As shown in Figure 5, the 3 simple analytics are represented as attributes. The attribute values of a row (i.e., a member) are kept up-to-date using stored procedures that implement a write actions such as Thaw Friendship and Invite Friend [7].

| Workload | SQL-X | ADC | SQLTrig | |
|---|---|---|---|---|
| | | | No Compression | Compression |
| Read Only | 25,411<br>RDBMS CPU | 63,292<br>KVS NICs | 62,450<br>KVS NICs | 81,829<br>KVS NICs |
| 0.1% Write | 21,584<br>RDBMS CPU | 61,032<br>KVS NICs | 62,594<br>KVS NICs | 80,549<br>KVS NICs |
| 1% Write | 13,227<br>RDBMS Disk | 22,418<br>RDBMS Disk | 21,763<br>RDBMS Disk | 21,319<br>RDBMS Disk |
| 10% Write | 10,055<br>RDBMS Disk | 14,404<br>RDBMS Disk | 12,004<br>RDBMS Disk | 11,869<br>RDBMS Disk |

Table 2: SoAR, actions per second, with 95% of requests observing a response time of 100 milliseconds or faster with no stale data.

The SQL query that implements *List Friends* requires an equi-join between Friends and Members table with an exact-match look up using the userid of the member whose friends is being listed. The SQL query for View Friend Request is similar and uses PdgFrds instead of Friends.

*View Top-5 Resources* is implemented using an SQL query that employs a range predicate on the priority attribute of the Resource table. The SQL query that implements *View Comments on Resource* consists of an exact-match selection predicate using the rid attribute of the Resource table.

Actions that write to the RDBMS (such as *Invite Friend*, and *Thaw Friendship*, see Table 1) invoke SQLTrig's authored triggers to invalidate the impacted key-value pairs. A subsequent reference for these key-value pairs observes a KVS miss, is redirected to the RDBMS for processing, and a new key-value pair is inserted in the KVS. Below, we provide a conceptual description of the SQL queries (keys) whose results (values) are invalidated by the different BG write actions.

When Member B invokes Invite Friend action to extend an invitation to Member A, the SQLTrig authored triggers invalidate two keys: the profile of A (as this member's number of pending friend invitation is incremented/updated) and A's list of pending friend requests. When A invokes BG's Reject Friend Request, its RDBMS update invokes the SQLTrig authored triggers to invalidate the same two keys as the Invite Friend action. When A invokes Accept Friend Request to become friends with $B$, the RDBMS updates invoke SQLTrig authored triggers to invalidate 5 keys: profile of A, profile of B, A's list of friends, B's list of friends, and A's list of pending friend invitations. Finally, when $A$ thaws friendship with $B$, the triggers invalidate four keys: profile of A, profile of B, A's list of friends, and B's list of friends. Note that each key is an SQL query instance with a result set as its value.

## E.3 Social Action Rating

This section reports on the Social Action Rating (SoAR) of the following three different configurations: 1) An industrial strength RDBMS named SQL-X by itself, 2) SQL-X configured with SQLTrig per discussions of Section D, and 3) SQL-X configured with Twemcache and maintained consistent using developer provided software, named application developer consistency, ADC. Details of ADC are as follows. It represents query instances and their results as key-value pairs. It extends the write actions of BG by identifying impacted query instances (keys) whose results (values) have changed and issues Twemcache delete calls for these keys.

SQLTrig's authored triggers compute the same set of keys as those deleted by ADC. Hence, ADC is comparable to SQLTrig with one key difference: the application (i.e., node hosting the BG benchmark clients generating requests) issues the delete calls directly to the KVS and there are no authored/registered triggers. One may use the performance observed with ADC as a measuring yard stick to quantify the overhead of the triggers authored by SQLTrig and executed by the RDBMS in the presence of updates.

Below, we compare the SoAR of the three alternatives with a social graph consisting of 100,000 members. Each member has 100 friends and 100 resources. The workload specifies a skewed distribution of access, emulating members as socialites using a Zipfian distribution with exponent 0.27. This means approximately 70% of requests reference 20% of the data. The database is twenty Gigabytes in size and does not fit in the the 16 Gigabyte memory of the server hosting SQL-X.

Table 2 shows the SoAR of the three systems with different workloads. Amongst the presented workloads, the 1% mix is most representative of a social networking site such as Facebook [5, 18]. With SQLTrig, we show system performance with and without the use of compression. In all experiments, a resource of the server hosting either the KVS or the RDBMS limits the observed performance. (Recall that the RDBMS and the KVS are hosted on different servers, see Figure 4.) This limiting resource is identified for each experiment and is as follows:

- RDBMS CPU: The CPU cores of the server hosting the RDBMS becomes 100% utilized.
- RDBMS Disk: A queue forms on the disk drive of the server hosting the RDBMS.
- KVS Network: Three 1-Gigabit per second network interface cards of the KVS server become 100% utilized. These three cards are bonded to provide a 3-Gigabit per second network connection.

With a read-only workload, the CPU cores of the server hosting SQL-X becomes fully utilized and limits its SoAR at 25,411 actions per second. Using the KVS with both ADC and SQLTrig without compression, the system SoAR improves to 63,000 actions per second with the network interface cards of the server hosting the KVS becoming fully utilized. Compression trades the available CPU cycles of the KVS server to reduce the size of the payload transmitted across the network (see Table 3), enhancing SoAR of SQLTrig to 81,829 actions per second. This is three folds higher than SQL-X all by itself.

The observed performance with a very low mix of write actions (0.1%) is similar to the read-only workload. As we increase the percentage of the write actions to 1% and higher, a queue forms on the disk of the server hosting SQL-X, causing the SoAR of all systems to drop. Compression no longer enhances

| BG read action | Uncompressed | Compressed |
|---|---|---|
| View Profile | 1,197 | 703 |
| List Friends | 82,836 | 50,740 |
| View Friends Req (Empty) | 338 | 111 |
| View Top-5 Resources | 1,865 | 1,198 |
| View Comments on Resource | 185 | 91 |

Table 3: Size (in bytes) of payload produced by different BG read actions.

| Benchmark | Query Templates | % Supported |
|---|---|---|
| TPC-W | 32 | 87.5% |
| RUBiS | 45 | 84.4% |
| RUBBoS | 35 | 77.1% |

Table 4: SQLTrig support for SQL queries issued by three other benchmarks.

the SoAR of SQLTrig as the networking cards of the KVS are no longer fully utilized. Moreover, the gap between SQL-X and ADC/SQLTrig closes as the KVS hit rate drops due to invalidations.

In general, SQLTrig and ADC provide comparable performance up to 1% mix of write actions. With 10% mix of write actions, SQLTrig provides a lower SoAR due to the overhead of the authored triggers querying the RDBMS to identify the impacted key-value pairs. With ADC, the human developer identifies the impacted keys (at the application level) without issuing RDBMS queries. This enables ADC to outperform SQLTrig.

### E.4 RUBiS, RUBBoS, TPC-W

We analyzed how effective SQLTrig is in generating triggers for queries issued by other benchmarks, namely, RUBiS [11], RUBBoS [31] and TPC-W [15]. The second column of Table 4 shows the number of unique query templates that constitute each benchmark. The last column of this table shows the percentage of query templates that SQLTrig supports by authoring triggers. With TPC-W, approximately 13% of its templates use a nested selection predicate and SQLTrig does not generate triggers for these at the time of this writing. With RUBiS, SQLTrig authored triggers for 84% of templates. The remaining templates use a range predicate with the function call to the current time, NOW(), such as “SELECT ... FROM ... WHERE EndDate $\geq$ NOW()”. The result of this query is dependent on the time of the wall clock and SQLTrig does not support its caching as it is constantly changing.

With RUBBoS, those templates with a where clause that use the term ”LIKE” are not cached. This clause is an approximate string match predicate for use with varchar attribute values. An example is a query such as: “SELECT ... FROM ... WHERE title LIKE ’?%’”. SQLTrig cannot cache the result of this query because it is unable to author triggers that identify a cached query instance based on the changed data.

## F Related Work

A key-value pair shares similarities with a materialized view, MV, of a RDBMS. Both speedup an application. However, their target applications are different. While MVs enhance the performance of applications that manipulate a large amount of data such as decision support applications and their On-Line Analytical Processing (OLAP) tools, key-value pairs enhance the performance of interactive applications that retrieve a small amount of data from big data. With the latter, MVs have been shown to perform poorly [7].

SQLTrig authored triggers notify the KVS of a query result change. This resembles the query change notification mechanism of RDBMSs such as Oracle 11g and Microsoft SQL Server 2005 and its later editions. We explored the use of this mechanism to invalidate key-value pairs in [22]. This study shows today's industrial strength RDBMSs either do not support notification for many (billions) query instances or slow down updates dramatically, providing service times in the order of minutes. SQLTrig is novel because it distinguishes between query templates and its instances, see Section C. It limits the number of authored triggers based on query templates which is typically in the order of tens to a few hundred for a given application. One may incorporate SQLTrig's translation process into an RDBMS to enable it to provide query change notification mechanism efficiently, see Section G.

Early transparent cache consistency techniques invalidated cached entries at the granularity of either table change or combination of table and column change [1]. These are suitable with web sites that disseminate information (e.g., stock market ticker prices [28], results of Olympic events [13]) where a table is the basis of thousands of cached entries. They become inefficient with applications such as social networking where each row of a table is the basis of a different cached entry and there are many (billions of) rows and corresponding cache entries. With these techniques, an update to a row would invalidate many (billions of) cached key-value pairs even though only a single entry should be invalidated.

A transparent consistency technique for a proxy-based cooperative query result caching system named Ferdinand is presented in [19]. This off-line technique computes a pairing of SQL query templates and the SQL update templates that impact the query result sets using query-update independence analysis [29]. This pairing is used at run-time when an update command is issued to publish query result change notification to cache servers. SQLTrig is different as it does not have an off-line process. Moreover, it does not analyze the update commands. Instead, it authors triggers that are invoked by update commands. These triggers employ an impacted row to construct the query instances (keys) whose result sets (values) have changed. It is important to note that SQLTrig is intended for a data center setting while Ferdinand was designed for proxy caches deployed in a geographically distributed setting. SQLTrig requires the KVS to provide sophisticated indexing techniques in order to invalidate key-value pairs efficiently.

TxCache [32] is a transparent caching framework that supports transactions with snap shot isolation. It targets RDBMSs that supports multi-version concurrency control [8], e.g., PostgreSQL, and extends them with additional software to produce invalidation tags in the presence of updates. A generated tag is based on a query whose results is used to generate a cached key-value pair. The tag is for one attribute value of a table (TABLE:KEY). This works when the workload of an application consists of simple exact-match selection predicates. Details of how this technique works for queries with range and join predicates are not clear and its presented evaluation avoided join queries due to the severe performance impact. SQLTrig is different because it does not require a change to its RDBMS. Thus, it can be used with all SQL RDBMSs that support triggers.

CacheGenie [25] employs an Object-Relational Mapping (ORM) framework such as Django to generate the SQL queries, object instances stored in the cache, and RDBMS triggers to invalidate cached objects. It can perform this for a subset of query patterns generated by the ORM. SQLTrig is different in two ways. First, SQLTrig generates triggers based on the issued SQL queries and not an ORM description. Thus, SQLTrig is applicable for use with both ORM[7] and non-ORM frameworks. Second, CacheGenie incrementally updates cached key-value pairs and suffers from dirty reads while SQLTrig invalidates key-value pairs and does not incur dirty reads.

## G Future Research

SQLTrig is a transparent consistency technique designed to provides physical data independence to reduce the complexity of applications and expedite their development life cycle. It targets those applications that perform simple operations that read and write a small amount of data.

The intuition behind SQLTrig is that the relational data model (hence RDBMSs) define table variables that are populated with values. An SQL query retrieves a set of records by specifying attribute values of interest. Similarly, an SQL insert/delete/update command adds/removes/modifies attribute values of one or more rows. The transformations of Section C author software on the fly to identify the query template of a query instance whose result is cached. This software reconstructs a query instance by substituting the wild card of its query template with the attribute values of a row that is being inserted/deleted/modified. This row, both its old and new versions, is identified by the trigger mechanism of the RDBMS when processing an SQL insert/delete/update. SQLTrig deletes these query instances (keys) from the KVS to propagate the RDBMS updates transparently, implementing physical data independence.

Currently, we are extending SQLTrig in several ways. First, we are developing a more formal definition of the transformation rules of Section C used to author software for different SQL query types. This formalism may enable us to develop a proof of correctness for SQLTrig with RDBMSs. Moreover, it may provide intuition into alternative implementations that may not require the use of triggers or the indexes of

[7]With an ORM framework, one simply replaces the JDBC driver of the run-time system with SQLTrig's client that provides a JDBC interface.

Section C. This may pave the way into how one may use SQLTrig with those data stores that do not support the relational data model, i.e., NoSQL data stores [10]. This is useful because we have experimental results showing query result lookup enhances the performance of a NoSQL solution such as MongoDB, see [7] for details. Extensions of SQLTrig to these data stores is an exciting future research direction.

Second, we are implementing SQLTrig in a multi-node architecture, analyzing how the internal keys (IntKeys) should be partitioned to enable an architecture to scale to a large number of nodes. Finally, we are also exploring alternatives to invalidation such as either refreshing [12] or incrementally updating [25] a key-value pair [23].

## References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In *ICDE*, 2005.

[2] C. Amza, G. Soundararajan, and E. Cecchet. Transparent Caching with Strong Consistency in Dynamic Content Web Sites. In *Supercomputing*, ICS ’05, pages 264–273, New York, NY, USA, 2005. ACM.

[3] C. Aniszczyk. Caching with Twemcache, http://engineering.twitter.com/2012/07/caching-with-twemcache.html.

[4] T. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. *ACM SIGMOD*, June 2013.

[5] T. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *SIGMOD*, pages 1185–1196, 2013.

[6] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR*, January 2013.

[7] S. Barahmand, S. Ghandeharizadeh, and J. Yap. A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. *CIKM*, 2013.

[8] P. Bernstein and N. Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Transactions on Database Systems*, 8:465–483, February 1983.

[9] JBoss Cache. JBoss Cache, http://www.jboss.org/jbosscache.

[10] R. Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[11] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *OOPSLA*, pages 246–261, 2002.

[12] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *ACM/IEEE SC*, November 1998.

[13] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.

[15] Transaction Processing Performance Council. TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp.

[16] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, D. VanderMeer, K. Ramamritham, and D. Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In *VLDB*, pages 667–670, 2001.

[17] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[18] R. Nishtala et. al. Scaling Memcache at Facebook. *NSDI*, 2013.

[19] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable Query Result Caching for Web Applications. In *VLDB*, August 2008.

[20] S. Ghandeharizadeh and J. Yap. Gumball: A Race Condition Prevention Technique for Cache Augmented SQL Database Management Systems. In *ACM SIGMOD DBSocial Workshop*, 2012.

[21] S. Ghandeharizadeh and J. Yap. Cache Augmented Database Management Systems. In *ACM SIGMOD DBSocial Workshop*, June 2013.

[22] S. Ghandeharizadeh, J. Yap, and S. Barahmand. COSAR-CQN: An Application Transparent Approach to Cache Consistency. In *International Conference On Software Engineering and Data Engineering*, 2012.

[23] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. Technical report, University of Southern California, Database Laborary Technical Report 2014-06, http://dblab.usc.edu/users/papers/IQTechReport.pdf, 2014.

[24] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[25] P. Gupta, N. Zeldovich, and S. Madden. A Trigger-Based Middleware Cache for ORMs. In *Middleware*, 2011.

[26] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, pages 981–992, 2008.

[27] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 49–60, 1997.

[28] A. Labrinidis and N. Roussopoulos. Exploring the Tradeoff Between Performance and Data Freshness in Database-Driven Web Servers. *The VLDB Journal*, 2004.

[29] A. Y. Levy and Y. Sagiv. Queries Independent of Updates. In *VLDB*, 1993.

[30] memcached. Memcached, http://www.memcached.org/.

[31] ObjectWeb. RUBBoS: Bulletin Board Benchmark, http://jmob.ow2.org/rubbos.html.

[32] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI*. USENIX, October 2010.

[33] M. I. Seltzer. Beyond Relational Databases. *Commun. ACM*, 51(7):52–58, 2008.

[34] M. Stonebraker and R. Cattell. 10 Rules for Scalable Performance in Simple Operation Datastores. *Communications of the ACM*, 54, June 2011.

[35] Terracotta. Ehcache, http://ehcache.org/documentation/overview.html.

[36] G. Whalin, X. Wang, and M. Li. Whalin memcached Client Version 2.6.1, http://github.com/gwhalin/Memcached-Java-Client/releases/tag/release_2.6.1.

[37] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. In *VLDB*, pages 188–199, 2000.

## H Appendix A: Marshalling of Result Sets by the SQLTrig Client

The SQLTrig client must serialize and deserialize (marshall) the result set (value) of a SQL query instance (key) to store and retrieve the key-value pair from the KVS. Ideally, a marshalling technique must be fast, efficient and produce the most compact serialized representation. With the Java programming language, this can be done by marshalling a serializable version of the JDBC ResultSet class. Since the general ResultSet class is not serializable, it has to be converted into an object that does support serialization. One such method

| | SQLTrig Marshalling | | Generic Java Marshalling | |
|---|---|---|---|---|
| | No Compression | With Compression | No Compression | With Compression |
| Average Size (bytes) | 1,536 | 972 | 7,671 | 3,787 |
| Avg Latency ($\mu$s) | 102 | 117 | 317 | 875 |

Table 5: Marshalling of YCSB Workload C ResultSet with SQLTrig and Java.

is to employ the CachedRowSet implementation[8] (by Sun, now Oracle) to generate a serializable instance of the query ResultSet class. This instance is populated with a ResultSet obtained by executing a query. Next, this instance is serialized into an array of bytes using the Java writeObject call. The resulting array of bytes is stored as the value portion of a key-value pair in the KVS. It might be compressed to minimize the memory footprint and network transmission time. When unmarshalling this array of bytes after reading it from the SQLTrig server, a corresponding Java readObject call is used to rebuild the original CachedRowSet instance. The Java marshalling and unmarshalling of objects are expensive because they are designed to handle arbitrarily complex classes. To avoid this overhead, we implemented our own marshalling of the ResultSet. It outperforms the Java marshalling technique because it is aware of the specific structure of the ResultSet object. It retrieves its number of columns and rows and stores them as the first eight bytes of an array. Subsequently, it stores the meta-data information for a column (name, length, table name, type) and its values for every row, producing a column store representation. Today, with variable length columns such as varchar, we store its data as a series of {length, value} pair. An alternative representation would be to store all {length} values followed by {value} of the columns. This would most likely produce a more compact representation when compressing our serialized representation.

We used the YCSB benchmark [14] (Workload C) to compare the generic Java marshalling technique with our implementation. YCSB is configured with one table consisting of ten string columns. Each column is 100 bytes long. The target query retrieves all columns of a single row. First row of Table 5 shows the average size of the resulting object with both SQLTrig's marshalling technique and the generic Java marshalling technique. Our marshalling technique results in representations that are 3 to 4 times smaller in both compressed and uncompressed format. Moreover, the service[9] time to both generate and compress[10] the value is faster with our implementation, see the second row of Table 5. Section E.2 compares Java marshalling with our marshalling technique for a social networking benchmark, demonstrating that the

[8]A commercial RDBMS software vendor may provide its own implementation of CachedRowSet as a part of its JDBC driver, e.g., OracleCachedRowSet. One may use this instead of the generic implementation.

[9]In this experiment, the RDBMS, cache server, and the client are hosted on the same PC. While there are inter-process communications, there are no inter-processor communications.

[10]Compression enables a more scalable infrastructure because it frees shared resources such as the cache space and the network bandwidth.

trends shown in Table 5 continue to hold true.