PIQL: A Performance Insightful Query Language

Michael Armbrust, Stephen Tu, Armando Fox, Michael J. Franklin, David A. Patterson
Nick Lanham, Beth Trushkowsky, Jesse Trutna

EECS Department
University of California, Berkeley

{marmbrus, sltu, fox, franklin, pattrsn, nickl, trush, jtrutna}@cs.berkeley.edu

ABSTRACT

Large-scale websites are increasingly moving from relational databases to distributed key-value stores for high request rate, low latency workloads. Often this move is motivated not only by key-value stores' ability to scale simply by adding more hardware, but also by the easy to understand predictable performance they provide for all operations. While this data model works well, lookups are only done by primary key. More complex queries require onerous, explicit index management and imperative data lookups by the developer. We demonstrate PIQL, a Performance Insightful Query Language that allows developers to express many of the queries found on these websites, while still providing strict bounds on the number of I/O operations for any query.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based services

General Terms

Languages, Performance

1. INTRODUCTION

Key-value stores have gained traction in both academia and industry as more developers transition to them from a traditional RDBMS. Of the five companies who run the top ten websites on the internet, three have announced publicly that they use or plan to use a key-value store for some portion of their site [1, 2, 9, 10, 13]. Additionally, many companies who still use traditional RDBMSs have implemented their own application-specific storage layers on top of them and use the database for simple operations [12].

One of the driving factors in this transition is the lack of performance transparency developers encounter when using a traditional RDBMS. It is easy for developers to write SQL statements that look simple but are actually very computationally intensive. In contrast, when developers are forced to write their own imperative code consisting of many simple get/put operations, it is more apparent when queries' performance will not scale as the site becomes more popular. Unfortunately, forcing developers to access data imperatively introduces its own set of problems. In addition to giving up physical data independence, developers must worry about optimization, index management, session guarantees and intra-query parallelization.

The known problems and additional complexity that comes with imperative programming of data-intensive applications has not deterred a growing community of developers who are rejecting query languages in favor of lower-level imperative interfaces [4]. We believe that this choice is an unfortunate one and is a classic case of "throwing the baby out with the bathwater" – depriving developers of the many benefits of high-level query languages. To address this problem, we have developed PIQL, a Performance Insightful Query Language[1]. PIQL is designed specifically for the development of large-scale data-intensive web sites and applications and is based on the following principles:

- It is based on a subset of SQL that enables accurate performance predication
- All queries must specified ahead of time and validated against a developer-specified service level objective (SLO)
- It has direct support for manipulating the entities and relationships that are commonly used in web applications
- Its implementation is based on automatically selected indexes
- It can be adapted to run on many existing key-value stores

In our demo we will show how PIQL enables the creation of compelling applications without requiring developers to worry about scaling considerations. We will demonstrate a sample application called SCADr, a clone of the popular internet site www.twitter.com. SCADr was built using PIQL in only a few days. We will show the site running on hundreds of machines on Amazon's EC2 under artificial load.

Attendees will be able to interact with the site a few different ways: First, as end users they can visit the public site at http://scadr.knowsql.org either from the provided computers or through their own mobile devices. We will have

[1]A more detailed description of the PIQL language can be found in the proceedings of the ACM Symposium on Cloud Computing (SOCC) 2010 [7].

a screen that shows metrics from the actual running system such as the current latency for queries in the system. Second, they can inspect provided code snippets, in order to get a feel for how developers actually write programs using the PIQL language. Finally, attendees will be able to experiment with the system through the interactive developer console by adding data, running existing queries, and proposing additional queries for new functionality.

2. PIQL ENTITIES AND RELATIONSHIPS

Much like the popular Active Record, an object relational mapper (ORM) from the RubyOnRails web framework, the DDL of PIQL is focused on the specification of *entities* and *relationships* [5].

```
ENTITY user
{
  string name,
  string password,
  string hometown,
  string profileData
  PRIMARY(name)
}

ENTITY thought
{
  int timestamp,
  string thought,
  FOREIGN KEY owner REF user
  PRIMARY(owner, timestamp)
}

ENTITY subscription
{
  bool approved,
  FOREIGN KEY owner REF user MAX 5000,
  FOREIGN KEY target REF user,
  PRIMARY(owner, target)
}
```

Figure 1: The entites and relationships for the SCADr Web application.

Entity Sets are typed collections of attributes, analogous to relations in a traditional relational database. An important distinction, however, is that PIQL allows only the atomic update of a single entity, instead of the general update mechanism present in SQL. Figure 1 shows an example of the syntax for entities of the sample application described in Section 5.

Relationships are defined by specifying a foreign key that points to another entity type. This is different from SQL, where the join predicates can be defined at query time. The reason for requiring this explicit specification is to allow the user to place *cardinality constraints* on specific relationships. These constraints can be specified in the form of a integer constant, 5000 in the above example. If no bound is stated then the relationship is assumed to be unbounded. Figure 1 shows the relationships of the SCADr application.

3. QUERIES IN PIQL 1.0

PIQL queries are similar to SQL, but have a few notable differences chosen to allow the calculations of strict bounds on the number of operations performed by each query: First, since all queries are specified ahead of time, they must be named and can take parameters, much like stored procedures in an RDBMS. These parameters come in the form `[ordinal:name]`, where the ordinal is used for placement in the argument list of the generated function. Second, each query returns only a collection of a single type of entity. Finally, joins can only be done on pre-specified relationships. The general syntax for all queries is as follows:

```
QUERY name
FETCH entity
  [OF joined-entity alias BY relationship] ...
WHERE predicates
[{PAGINATE perpage | LIMIT count}]
```

The following are examples of this syntax from the SCADr application. The first query looks up a user by the primary key, username, while the second looks up a set of users with a given hometown, returning a variable number of results with a maximum of ten.

```
QUERY userByName
FETCH user
WHERE user.name = [1:name]

QUERY userByHometown
FETCH user
WHERE user.hometown = [1:hometown]
LIMIT [2:count] MAX 10
```

A third, more complicated query returns a paginated list, 10 at a time, of the most recent thoughts of all the approved subscriptions owned by the current user.

```
QUERY thoughtstream
FETCH thought
  OF user friend BY owner
  OF subscription BY target
  OF user me BY owner
WHERE me.username=[1:username] AND approved = true
ORDER BY timestamp
PAGINATE 10
```

This query first locates the current user by the primary key `me.username=[1:username]`. Then, it finds all the subscriptions of the current user by using the owner relationship between user and subscription, `subscription OF user BY owner`. It then locates all of the targets of these subscriptions, `user OF subscription BY target`. These subscriptions are filtered to only include those which have been approved, `approved = true`. Next, we find the target user of the approved subscriptions, `user OF subscription BY target` Finally, all of the thoughts of these users are located (`thought OF user BY owner`), sorted by time (`ORDER BY timestamp`), and returned ten at a time (`PAGINATE 10`).

As was mentioned before, many of the deviations from traditional SQL result from our desire to provide guaranteed performance bounds for all queries. This is similar to a related language GQL, which is Google AppEngine's interface to the underlying BigTable storage system. GQL is very similar to SQL and has a number of performance-based restrictions [3]. PIQL extends this model by allowing joins of limited cardinality, providing performance estimates based on machine learning models, and allowing use on systems other than those internal to Google. As a result, the system will analyze all of the queries in a given application spec and reject any query either has an unbounded intermediate step or an unbounded final result. An example of a query that would have an unbounded final result would be the `userByHometown` query from the previous subsection if there were no `LIMIT` clause. This is due to the fact that hometown is not a primary key, and as such any number of users could have the same town listed. An example of an unbounded intermediate step would be the thoughtstream query, if we had not created the cardinality limitation of 5000 but had instead chosen `MANY`. In either of these cases the compiler described in the next section would have thrown a compilation error.

4. IMPLEMENTATION

4.1 Compiler

The PIQL compiler takes in an application spec and produces a library jar that can be used from any JVM based language to interact with a cluster of SCADS Storage Nodes [6]. This jar could be used from any one of the many popular web frameworks that run on the JVM. For example, the SCADr demo was written using the Play Framework [8]. For each entity, a class is created that allows the creation and updating of the values for one specific instance of an entity type. Getter and setter methods allow users to modify attributes, and a save method attempts to update the entity along with reliably updating any asynchronously maintained indexes needed to correctly answer queries efficiently. Each query is compiled into a function with arguments for each of the parameters of the query.

While we currently only support bytecode generation, it should be relatively easy to output code for another popular web development language. In fact, there is currently an effort led by some undergraduates at Berkeley to add support for PIQL to RubyOnRails. Additionally, the underlying storage system could be swapped our for any key-value store that supports a few simple operations (get, put, test and set, get by key prefix, etc.) with only minor code changes.

The compiler uses a simple heuristic-based optimizer to choose indexes that will need to be created, and generate execution plans for all of the queries. Queries that use the paginate operator will return a special object that can be used to retrieve a list of the current results, as well as to access the next page. This object is easy to serialize and deserialize so that it can either be placed in the web framework's session store or used as arguments to the next URL invocation. It acts as a client-side cursor allowing the system to retrieve the next round of results starting exactly where the last page off and not back at the beginning. This feature allows us to not only retrieve an arbitrary number of "next pages" with constant performance, but also to provide a more consistent user interface in the case of insertions while paging.

4.2 Performance guarantees

Another important feature of the compiler is to ensure that the queries will perform within the user specified SLO even in the worst case and in spite of user-base growth. Two key factors make such guarantees possible. First, we rely on

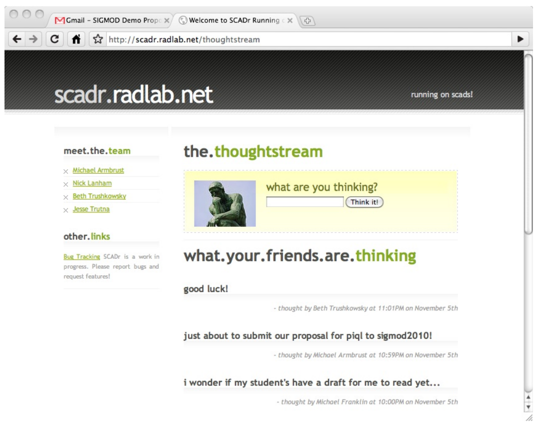
```
TopK(10) => 10
Sort(descending:timestamp) => 10 * 5000
IndexJoin("ent_thought",
    thought:owner == subscription:owner,
    descending: timestamp) => 10 * 5000
Select("approved" == true) => 5000
IndexLookup("ent_subscription",
    subscription:owner == [username]) => 5000

Total Reads = 5000 + 10 * 5000
Other Work = 5000 +
  (10 * 5000) * log(10 * 5000) + 10
```

Figure 2: Example performance information from the interactive query console about the thoughtstream query.

the fact that the underlying data store is designed to provide predictable performance by scaling both up and down based on machine learning models of expected load, as described in [6]. Combining this predictability with the required bounds imposed by the query language, we can derive the maximum number of simple operations that will need to be performed by any query. This allows developers to avoid the type of unexpected scaling catastrophes that plague and sometimes destroy sites as they become more popular [11].

4.3 ORM Convenience Features

The compiler also contains some syntactic sugar to make the query interface more usable. One example is the *implicit this parameter*. While all of the example queries use named parameters, a query can optionally contain a single `[this]` parameter, which has the value of the primary key of the object that the query was called on. Any query that contains `[this]` will be compiled as an instance method of an entity class instead of as a static method. This would, for instance, make it possible to call the thoughtstream query using the syntax `user1.thoughtstream` instead of the more verbose `Queries.thoughtstream(user1.username)`



Bug Tracking SCADr is a work in progress. Please report bugs and request features!

good luck!

- thought by Beth Trushkowsky at 11:01PM on November 5th

just about to submit our proposal for piql to sigmod2010!

- thought by Michael Armbrust at 10:59PM on November 5th

i wonder if my student's have a draft for me to read yet...

- thought by Michael Franklin at 10:00PM on November 5th

Figure 3: A screen shot of the SCADr Application.

5. DEMO

Our demo will consist of a real application, SCADr, running on hundreds of nodes on Amazon's Elastic Compute

Cloud. Users of SCADr add tidbits of information in the form of thoughts to the system. These thoughts are broadcast to all users who have subscribed to that user's thought-stream and have been approved.

Artificial load will be applied by a number of clients to simulate many users on the system, and a local client will show that the response time of the system stays within the accepted SLO. Conference attendees will be able to access the site from a provided laptop, or from their own mobile devices. In the event of limited connectivity (both internet and cellular) we will run a scaled down version of the site on a local laptop.

Attendees will also be able to propose new ad-hoc queries to the system running on EC2. If the query is bounded, the system will provide an annotated query plan explaining the maximum number of operations that will be performed during execution. Figure 2 shows an example annotated query plan. Additionally, if all of the indexes needed to answer the query are available the system will return the answer.

Finally, we will also provide attendees with the ability to experiment with the query language by modifying the existing spec with new entities, added queries, or changed relationships. The system will provide updated performance expectations based on the specified changes.

6. REFERENCES

[1] Alexa top 500 global sites. Available from: `http://www.alexa.com/topsites`.

[2] Baidu sponsors hypertable [online]. Available from: `http://www.hypertable.org/sponsors.html`.

[3] GQL. `http://code.google.com/appengine/docs/python/datastore/gqlreference.htm%l`.

[4] Nosqleast conference [online]. Available from: `https://nosqleast.com/2009/`.

[5] Ruby on rails api: Activerecord. Available from: `http://api.rubyonrails.org/classes/ActiveRecord/Base.html`.

[6] Armbrust, M., et al. Scads: Scale-independent storage for social computing applications. In *CIDR* (2009), www.cidrdb.org.

[7] Armbrust, M., Lanham, N., Tu, S., Fox, A., Franklin, M., and Patterson, D. A. Piql: A performance insightful query language for interactive applications. First Annual ACM Symposium on Cloud Computing (SOCC).

[8] Bort, G. The play web framework. Available from: `http://www.playframework.org/`.

[9] Chang, F., Dean, J., et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 1–26.

[10] Lakshman, A., and Malik, P. Cassandra: A structured storage system on a p2p network. Presented at SIGMOD 2008.

[11] Rivlin, G. Wallflower at the web party. *The New York Times* (October 15 2006).

[12] Sobel, J. High performance at massive scale. Talk at HPTS 2009.

[13] Srivastava, U. Pnuts - platform for nimble universal table storage. Talk, October 2007.