PIQL: A Performance Insightful Query Language For Interactive Applications

Michael Armbrust
Nick Lanham
Stephen Tu
Armando Fox
Michael Franklin
David A. Patterson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-8
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-8.html

January 25, 2010

PIQL: A Performance Insightful Query Language For Interactive Applications

Michael Armbrust, Nick Lanham, Stephen Tu,
Armando Fox, Michael J. Franklin, David A. Patterson
University of California, Berkeley
{marmbrus, nickl, sltu, fox, franklin, pattrsn}@cs.berkeley.edu

ABSTRACT

Large-scale, user-facing applications 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. For complex queries this approach often requires onerous explicit index management and imperative data lookup by the developer. We propose PIQL, a Performance Insightful Query Language that allows developers to express many of the queries found on these websites, while still providing strict performance guarantees.

1. INTRODUCTION

There are a growing number of user-facing, large-scale, data-intensive applications. In addition to huge data volumes and high request rates, these applications are characterized by a need for interactive response times due to the well documented effects of latency on user behavior [10]. Even small amounts of latency can measurably affect the behavior of users, and applications that are unable to keep up with their growing user base will quickly lose it to their competition [9].

The desire to rapidly add features and deploy new versions of the application make the traditional RDBMS an attractive option for developers of these applications. Features like the independence from the physical and logical layout of the data result in more maintainable code and SQL is sufficiently expressive for answering virtually any query against the data they store. Unfortunately, a consequence of this expressive power is performance opacity, which can make it very difficult to reason about the latency of a given query, especially as the data grows by orders of magnitude.

As a result, many in this space have abandoned the RDBMS and instead write their applications in terms of low-level and performance-transparent operations like `get()` and `put()` against a key/value store. Examples of key/value stores include Facebook's Cassandra, Amazon's Dynamo, and Google's BigTable [3, 5, 7]. Each of these systems was designed specifically to provide high-percentile performance service level objectives while storing ever increasing amounts of data, simply by adding more machines. Thinking in terms of reliable, low level actions enables developers to easily understand the performance and scaling behavior of their queries, but it also means they must sometimes write complex imperative functions. Additionally, for performance reasons, they are often forced to directly utilize and sometimes maintain indexes, losing physical data independence that would be provided by an RDBMS.

Even those sites that opt to store data in an RDBMS have typically created a proprietary layer on top of many federated databases. These libraries use the database in a limited way, often as a (less scalable) key/value store. Real world examples of this pattern include Facebook which maintains a PHP library to federate a large number MySQL instances and memcached nodes, and they specifically do not do any complex queries inside of the database [11].

Because of these performance opacity and scalability issues, application developers have been forced to code with low level tools that lack many of the features of higher-level declarative systems. PIQL (pronounced 'pickle') seeks to solve this problem by providing developers with a *performance predictable* language subset of SQL. It provides developers many of the benefits of using a traditional RDBMS, such as the ability to express their queries declaratively, automatic data parallelism, physical data independence, and automatic index selection and maintenance, all while still maintaining the *soft real-time guarantees* on application performance that come from the underlying key/value store. Features of the system include:

- It is engineered specifically to run on top of existing performance predictable key/value stores.
- Guaranteed bounds on the number of operations that will be performed for any insert, update or query.
- Compile time feedback on worst-case performance for all queries in a given application.
- Language features that allow unbounded amounts of data to be efficiently traversed, e.g. pagination.
- Support for a heterogeneous schema providing the ability to do rolling updates without affecting performance of the running system.
- Automatic selection and maintenance of indexes needed to provide performance guarantees.
- The option to trade strong consistency for performance or availability.

2. BACKGROUND

2.1 Alternative Approaches

As was mentioned in the introduction, existing approaches for supporting large-scale interactive applications primarily focus on complex developer written abstraction layers and/or imperative programs. The complexity of this approach is exemplified by the data model of the Cassandra system developed at Facebook and its example usage, inbox search [7]. The Cassandra data model is defined as a four or five dimensional hash table where *keys* point to *rows* which contain *super columns* which contain *column families* which contain *columns* which consist of values with an optional timestamp. In order to allow users to search for messages that contain a certain word, the developer must manually insert a value for each word in each message that is received by a user. Values inserted are of the following form: $row \rightarrow userid$, $supercolumn \rightarrow word$, $column \rightarrow messageTimestamp$, $value \rightarrow messageId$. [1]

In contrast to the complexity of the Cassandra data model, the inbox search example could be expressed in PIQL simply as:

```
FETCH message
  OF user BY recipient
WHERE user = [this] AND
  message.text CONTAINS [1: word]
ORDER BY timestamp
```

There are also a number of projects like PIG and HIVE that strive to provide a declarative language for manipulating data that is not stored in a traditional RDBMS. However, these are focused primarily at batch analytics and not interactive applications. These and other related systems are discussed in Section 9.

2.2 Motivating Example Application

To illustrate the PIQL language we will use a sample application SCADr[2], which is a simplified clone of the popular micro-blogging site Twitter. SCADr allows users to share their thoughts, in the form of small chunks of text, with other users who are interested in them.

2.3 Assumptions

PIQL was designed with a number of assumptions about the target applications and the underlying data store. First, we assume that the simple operations of the underlying key/value store will operate with predictable performance even as more data is added or as the request rate increases. This is possible either by over-provisioning, or for the cost conscious, dynamically scaling the system up and down using utility computing and the methods described in [2]. Additionally, while in practice many sites operate their key/value store with a majority of the data hosted in main memory, this is primarily for latency and cost per IOP reasons. We can still provide soft real-time guarantees, albeit with higher latency, on top of a properly provisioned disk based store.

[1]The difficulty of mapping other queries on to this model is evidenced by the number of questions on the mailing list and the prevalence of blog posts chronicling other query implementations.

[2]SCADr is a combination of the idea that users will use it to "scatter" their thoughts to the Internet and the larger project name "SCADS".

The second set of assumptions concern the applications that will be using PIQL.

The language was inspired by applications like eBay, Facebook and Twitter, and as such we assume that all queries have soft real-time requirements on the order of milliseconds, due to the aforementioned effects of page load latency on user behavior [10]. While there are classes of analytic queries that are very interesting and have different requirements, these are not the focus of the PIQL language./ Additionally, while the current implementation of PIQL relies on trading strong consistency for performance, this is only an option rather than a requirement of the language. In Section 5 we discuss the possible trade-offs in greater detail. In the next section we will discuss the data manipulation language (DML) of PIQL and how the optimizer calculates bounds on the number of low-level operations that any insert, update, or query will need to perform.

3. DML WITH BOUNDED OPERATIONS

3.1 Insert and Update

In PIQL, data is stored in sets of typed *entities*, which are analogous to relations. However, the model for updating them is significantly different. First, PIQL will only guarantee the atomicity of the update or creation of a single entity. While it may be possible to relax this constraint slightly, there are no plans to support a more general bulk method like SQL's `UPDATE` statement. When an entity is updated, the system will ensure that any indexes that refer to that entity are also updated. This model has the advantage of ensuring that the number of writes that must be performed for any single update or insert can be at most one plus the number of indexes for that entity.

Examples of the entities that are used in the SCADr sample application are shown in Figure 1.

3.2 Queries

All queries in a PIQL application are specified as templates ahead of time. The primary benefit of this approach is that it allows the compiler to compute the number of operations that will be required in the worst case for each query and provide this feedback to the developer at compile time instead of runtime. While there is an interactive mode of PIQL programming that allows ad-hoc queries, it cannot provide compile-time performance feedback or create indexes and is thus not intended for production use.

Queries can be parameterized, effectively allowing many different queries to be executed at runtime. The compiler is still responsible for determining the upper bound on the number of low-level operations executed for each query, regardless of what values are passed in as parameters at runtime. Take a simple query that looks up the profile of a user, given a user name.

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

Calculating the bound on this query is easy, as in this case the fields that are present in the predicate are a superset of the fields that compose the primary key of the user entity. As a result we know that this query will always perform a single get request, and will return either one or zero results.

```
ENTITY user                ENTITY thought                   ENTITY subscription
{                          {                                {
  string name,               int timestamp,                   bool approved
  string password,           string thought                   PRIMARY(owner, target)
  string email,              PRIMARY(owner, timestamp)      }
  string hometown          }
  PRIMARY(name)
}
```

Figure 1: The entities for the SCADr Web application.

Another slightly more complicated example would look up a user by their hometown:

```
QUERY userByHometown
FETCH user
WHERE user.hometown = [1:hometown]
LIMIT [1:count] MAX 100
```

Here, it is unclear how many users will have the same hometown, as there are no cardinality restrictions imposed by the schema. As a result, in PIQL the `LIMIT` clause is mandatory when the primary key is not present, and enforced by the compiler. This query would be satisfied by a bounded range get on an (automatically created) index of users by their hometown. We can be assured that there will be only one such `range_get()`, and it will return at most 100 items.

3.2.1 Joins

The PIQL language also allows equi-joins against pre-specified relationships between entities. For example, if we wanted to return a list of the most recent thoughts owned by a particular user, we could define the following `RELATIONSHIP` and `QUERY`:

```
RELATIONSHIP owner FROM user TO MANY thought

QUERY userThoughts
FETCH thought of
  user by owner
WHERE user.name = [1:username]
ORDER BY timestamp
LIMIT [2:count] MAX 100
```

This query would be satisfied by a bounded range get on an index of thoughts by owner and timestamp, in this case the primary index for thought entities. Thus we know that it will perform at most one range get and return at most 100 entities.

3.2.2 Pagination

Sometimes a simple limit clause can be too restrictive. For example, when looking at a list of thoughts from a given user stored by time, users may want the ability to see thoughts that are older than the most recent hundred. PIQL makes this possible while still keeping our soft real-time promise by the introduction of the `PAGINATE` operator. `PAGINATE` can be used in place of the LIMIT clause and will cause the system to return an easily serializable iterator. This keeps track of start key offsets as successive sets of thoughts are returned and makes it possible to "pick up where you left off" simply by retrieving and deserializing the iterator from the user's session. Note that, if we instead provided a numeric offset with the limit, it would take $O(n^2)$ reads from the underlying store to page over n thoughts from a given user.

3.2.3 Developer Specified Cardinality Constraints

Even more complicated joins can be expressed in a performance-safe way, with the addition of developer-specified join cardinalities. Such restrictions are already present in many systems for performance reasons; for example Facebook users were at one point restricted to having at most 5000 friends. By allowing the compiler/optimizer to be aware of such cardinality restrictions, we can make our language more expressive while still maintaining the soft real-time promise of the compiler. An example would be a query that returns the most recent thoughts of all the users that a given user is subscribed to and where that subscription has been approved:

```
RELATIONSHIP owner FROM user TO 5000 subscription
RELATIONSHIP target FROM user TO MANY subscription
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
LIMIT [2:count] MAX 100
```

In spite of the fact that this query is fairly complicated, the PIQL compiler is still able to calculate an upper bound on the number of operations that will be required to execute this query. The query plan will first perform a `range_get` on an index of all subscriptions that are owned by the user '`me`'. Next, it will scatter/gather the `count` most recent thoughts for each user that is a target of the returned `subscription` list. These can then be merge sorted and the top `count` returned to the application. As a result, even in the worst case this query will perform no more than $1 + 5000$ operations.

3.2.4 Join Indexes

Finally, there are some queries that can only be implemented in PIQL using a join index. One example is a query that returns a list of the most recent thoughts with a given tag:

```
ENTITY tag {
  string name
  PRIMARY(target, name)
}

RELATIONSHIP target FROM thought TO 10 tag

QUERY recentByTag
FETCH thought
  OF tag BY target
WHERE name = [1:tagname]
ORDER BY timestamp
LIMIT [2:count] MAX 100
```

In order to satisfy this query while still maintaining the soft real-time promise the system will need an index over both the name of the tag, and the timestamp. Otherwise the system may need to read an arbitrary number of thoughts with a given tag before finding the most recent ones. Additionally, due to the bound of ten tags per thought introduced by the relationship specified above, PIQL can guarantee than any updates to the timestamp of a thought will result in a bounded number of update operations.

4. DDL WITH BOUNDED OPERATIONS

A challenging aspect of interactive web applications is their rapid rate of change. New applications, versions, and features are pushed into production rapidly, and upgrades are often done in a rolling fashion. As a result the PIQL system was designed to not only allow DDL operations to be performed without sacrificing performance, but also to allow heterogeneous version of the schema to be running concurrently on the same cluster. This is done through use of a self-describing record format and the option of phased deployment of new schemas. While there is a common concern regarding performance of such record layouts, encodings like Google Protocol Buffers have shown that they can be used for performance critical applications [4].

The tagged record format means that when entities are stored we tag each field with the name of the field that is being stored. This allows the system to have a heterogeneous schema, enabling us to update stored entities one at a time as fields are added or removed.

Phased deployment of a new schema occurs in the following order:

1. The application is updated to stop using any deprecated fields or queries.
2. A new version of the PIQL spec is compiled containing any new fields or queries and removing any deprecated ones.
3. The new spec is rolled out to the cluster, potentially starting with a small subset of the machines.
4. Indexes and entities are updated to the new spec either as they are accessed and modified or using some kind of rate-limited sweep.
5. The application code is changed to utilize the new fields and queries.
6. The new version of the application is rolled out either gradually (such that only some users are directed to the machines running the new version) or all at once.

This strategy not only allows for consistent performance during the upgrade, but also allows efficient rollback in case any issues are encountered at any phase. Additionally, a trade off between the speed of the creation of new indexes and fields and the performance of the system allows developers to balance the time-to-release with current performance.

5. CONSISTENCY

Many of the installations that the PIQL language is targeting are willing to trade strong consistency for availability and performance of certain operations. This is often done in response to incredibly high request rates (Facebook serves 200 billion page views a month each with many database queries and 3.9 trillion feed actions per day [11]), or the desire to deploy an application across multiple datacenters for latency and availability reasons. While this is not a requirement of using PIQL, there are a number of places where we plan to allow such consistency/performance trade-offs to be made.

The first example occurs during entity creation and update. The current PIQL implementation provides serializability of writes to a single entity using a test/set operation to all replicas in a consistent order. It is conceivable, however, that one might want to use a quorum write to allow for higher tolerance to failures or slow-downs of a small number of replicas. Additionally, if strong isolation is required, two-phase commit could be used to update all of them synchronously.

There are also trade-offs to be made when updating index entries. While the current system updates indexes asynchronously using a write-ahead log for atomicity, two phase commit could be used here too. Conversely, the write-ahead log could be turned off if the application can tolerate occasional index inconsistencies.

There are also a number of trade-offs when executing queries. First, read policies abstract the underlying operations on a replicated distributed store from the higher-level relational operators. For example, when displaying a user profile, it may be acceptable to read only from the closest replica, with the understanding that it may be stale. However, when authenticating a user it might be desirable to read from all replicas of a given entity to ensure that a stale password can not be used.

There are also consistency interactions that span both read and write policies. For example, consider the thoughtstream query from the previous section and an eventually consistent index of the subscription entity sorted by owner and approval status. Reading only from the index and not looking at the actual subscription would improve performance, but would return potentially stale results. However, if we instead maintain this index synchronously (shifting cost to the write operation) or verify the index by reading the actual subscription, this condition could be alleviated.

6. DATA LOCK-IN

Due to the fact that the PIQL execution engine is designed to run on top of an existing key/value store, there is less of a concern with data lock-in. Other applications can access and modify the data in the underlying store as long as they obey any contracts regarding index consistency. This means that analytic queries could be written in another language like PIG and executed by Map/Reduce [8].

7. IMPLEMENTATION

Our prototype implementation of PIQL is written in Scala and provides the aforementioned functionality for at least one point in the consistency-performance trade-off space. Figure 2 shows an overview of the architecture. A developer writing a web application would run the PIQL compiler on their application spec (Entities, Relationships, and Queries). The compiler produces a Jar that contains classes for each entity and functions for each query. The developer can then use this Jar file to communicate with a cluster of underlying SCADS storage nodes. For example, to store and then recall

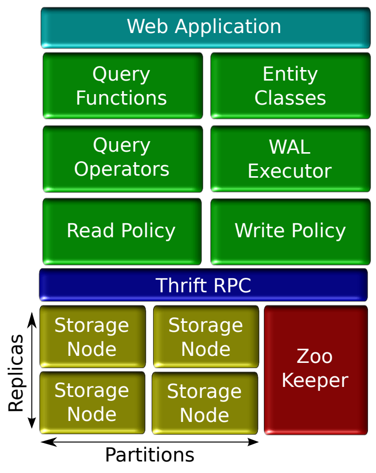

Figure 2: The architecture of the PIQL system.

a user from the system, a developer would simply write:

```
//Creation
val u = new user
u.name("marmbrus")
u.save

//Retrieval
val u = Queries.userByName("marmbrus")
```

We will now describe the most import components of the PIQL system.

7.1 Read Path

The query functions are composed of operators, much like queries in a traditional database. These operators, however, are customized to operate over a generic key/value store. The operators fall in two categories: *Remote Operators* perform actions on the underlying key/value stores. They include operations like load and entity by primary key, lookup ranges of an index, or perform index joins. These operators are guaranteed to always return a bounded set. *Local Operators* like *sort* and *selection* are very similar to their counterparts in an RDBMS, but for performance predictability reasons they are only allowed to operate on bounded sets.

7.2 Write Path

The write path is invoked when a developer calls `save` on a SCADS entity. It is responsible for atomically, but eventually (unless we are adjusting consistency), updating the value stored for all replicas of the entity and also any indexes that point to the entity. This is done by a simple write ahead log that is present at each application that has included the PIQL generated jar.

7.3 SCADS Store

The SCADS store is a modular system composed of two main pieces: the ZooKeeper and set of storage nodes. ZooKeeper is a distributed lock manager from Yahoo Research that maintains a list of which key ranges are assigned to which nodes. The SCADS Storage node is built on top of the BerkeleyDB Java Engine and provides operations like get, get range, put, and atomic test/set. It could be easily replaced with any other storage system that supports the same operations.

8. PERFORMANCE OVERVIEW

In order to validate the above performance claims we ran our system with successively larger numbers of users and machines. We used the SCADr schema from Section 3 and a generator that created 10,000 users per server, with twenty thoughts, and uniformly random subscriptions to the thoughts of ten other users. After boot-strapping the system with artificially generated users, we simulated application servers with one load client per server that repeatedly called the thoughtstream query for five minutes. We measured both the total throughput of the system and the latency of individual queries. All experiments were run on Amazon's EC2 using small instances (1.7 GB of memory, 1 EC2 Compute Unit or 1 virtual core with 1 EC2 Compute Unit)

As the number of machines/users increased from 20 to 100 and 200,000 to 1,000,000 respectively, the throughput increases nearly linearly as expected. In addition, Figure 3 shows that the latency remains virtually constant as promised.

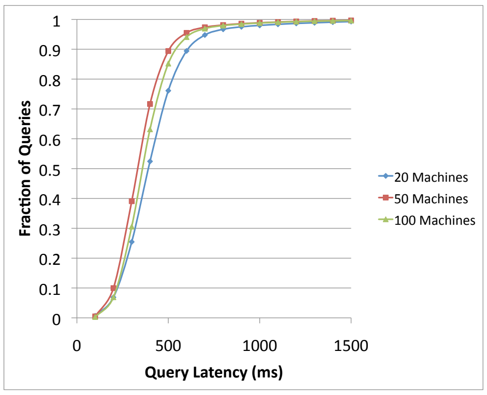

Figure 3: This CDF shows that as the system scales in both the number of users and the number of machines query latency remains relatively constant and 90% of queries are still answered in less than 600ms.

Another benefit of our compile time approach, which requires all queries to be specified ahead of time, is the option for multi-query optimization and index suggestion. As an example we ran two experiments, one where the primary key for thoughts was (timestamp, owner) and another where it was (owner, timestamp) as suggested by the optimizer. This allows the primary index to be used to answer the thoughtstream query instead of another secondary one. In addition to decreasing the cost of inserts and updated to thought entities, it also has the added benefit of converting the series of index dereferencing random reads to a single sequential one. As can be seen in Figure 4, this optimization improves that aggregate query throughput by over 70%. We believe this is only one of many multi-query optimization opportunities available.

9. RELATED WORK

Other projects in both the distributed systems community and the database community have tried to tackle similar workloads.

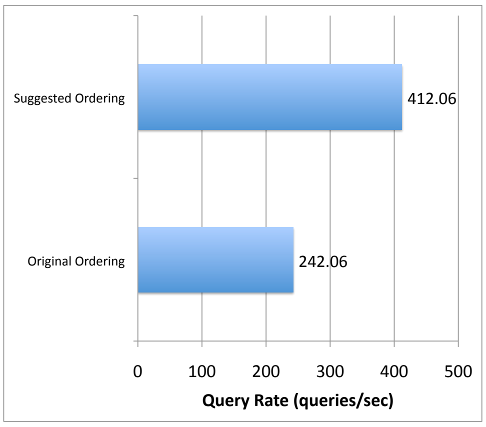

Figure 4: Implementing the suggested ordering by the optimizer yields over a 70% increase in query throughput.

GQL is the language that Google AppEngine provides to developers as a higher level abstraction to the underlying BigTable store [1]. Many of their language decisions were motivated by a desire to bound the cost of each query. PIQL improves on this by significantly expanding the types of queries that can be executed with the addition of relationships and cardinality constraints, which make the safe expression of joins possible Another benefit of the PIQL system is the modularity, allowing a much wider space of performance and consistency to be explored, as well as the option of running on a wider range of underlying stores.

As was mentioned earlier, systems like Hive and Pig are designed to provide a high-level declarative language to an underlying distribute system. However, neither of these projects was focused on interactive workloads.

VoltDB, and its research counterpart H-Store, are relational MPP database management systems specifically designed for large-scale OLTP applications [6]. They provide developers feedback to help them partition their data such that many of the queries can run in "fast mode" on single machine. It however also has a "slow mode" where more traditional techniques are used to execute queries across machines while maintaining ACID. PIQL on the other hand only has a "fast mode" and has a compile time step to validate the performance of the application before deployment, along with the option of relaxing consistency for performance.

10. FUTURE WORK

We plan to expand our system by seeing what functionality can be added to PIQL while still providing predictable performance. While it is possible to write many of the queries that would be needed to implement most popular sites that currently exist, there are a few omissions, the most notable being the lack of aggregates. However, we believe that there are techniques that could be used to implement many aggregate queries while still providing the same soft-real time promises.

Another place we plan to improve on the existing prototype is implementing some of the consistency performance trade-offs discussed in Section 5. The challenge here is twofold. We hope to not only understand exactly what the consequences are to both performance and consistency as you implement different options, but also how different options interact. There is also the challenge of how to expose the options to a developer in a manner that they can understand.

Additionally, there are a number of performance improvements that we hope to make. Most notably there are many experimental artifacts as a result of the synchronous RPC system we are currently using. It is our hope that switching to an asynchronous system will allow us to do more experiments with intra-query parallelism, significantly lowering the per-query latency.

11. CONCLUSION

Developers are increasingly abandoning the RDBMS and all of the benefits thereof, including the ability to express their queries declaratively, automatic data parallelism, physical data independence, and automatic index maintenance, in favor of performance predictable key/value store. We have described the PIQL system which provides developers of large-scale data-intensive applications many of these benefits while maintaining the soft real-time guarantees of the underlying key/value store. In addition to compile time performance feedback on worst case performance, we have shown how the system gives developers the ability to rapidly add features and roll-out new versions of the application without disturbing the performance of the running system.

12. REFERENCES

[1] Gql [online]. Available from: `http://code.google.com/appengine/docs/python/datastore/gqlreference.htm%l`.

[2] Armbrust, M., Fox, A., Patterson, D. A., Lanham, N., Trushkowsky, B., Trutna, J., and Oh, H. Scads: Scale-independent storage for social computing applications. In *CIDR* (2009), www.crdrdb.org.

[3] Chang, F., et al. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI* (2006), USENIX Association, pp. 205–218.

[4] Dean, J., and Ghemawat, S. Mapreduce: a flexible data processing tool. *Commun. ACM 53*, 1 (2010), 72–77.

[5] DeCandia, G., et al. Dynamo: amazon's highly available key-value store. In *SOSP* (2007), T. C. Bressoud and M. F. Kaashoek, Eds., ACM, pp. 205–220.

[6] Kallman, R., et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB 1*, 2 (2008), 1496–1499.

[7] Lakshman, A., and Malik, P. Cassandra: structured storage system on a p2p network. In *PODC* (2009), S. Tirthapura and L. Alvisi, Eds., ACM, p. 5.

[8] Olston, C., et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference* (2008), J. T.-L. Wang, Ed., ACM, pp. 1099–1110.

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

[10] Schurman, E., and Brutlag, J. Performance related changes and their user impact. Presented at Velocity Web Performance and Operations Conference, June 2009.

[11] Sobel, J., and Rothschild, J. High performance at massive scale. Presented at HPTS, October 2009.