

The Case For PIQL: A Performance Insightful Query Language

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

University of California, Berkeley
{marmbrus, nickl, sltu, fox, franklin, pattns}@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 queries found on these websites while still providing strict bounds on the number of I/O operations that will be performed.

Categories and Subject Descriptors

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

General Terms

Languages, Performance

1. INTRODUCTION

Large-scale, user-facing applications are storing and processing data at an unprecedented scale. Facebook, for example, serves over 200 billion page views a month, each with many queries to the database [15]. The conflict between existing technology and the requirements of these applications has led to the nascent NoSQL movement, which currently consists of a few open-source and proprietary storage systems, many blog posts, and even entire conferences [10, 7, 5, 11, 9, 2]. This movement is characterized by its intentional abandonment or ignorance of the work of the traditional database community. In fact, Ousterhout and others recently went so far as to claim that, at least in the context of datacenter storage, "SQL is dead" and that the "relational query model [is] tied to high latency" [13].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0036-0/10/06 ...\$10.00.

We believe one of the primary motivations for the developers making this transition is a desire for performance predictability. Their requirements are based on the well documented effects of latency on user behavior and thus the ability of their sites to succeed financially [14]. These developers have found it easier to meet their service level objectives (SLOs) using low-level and performance-transparent operations directly against a key/value store, such as `get()` and `put()`, instead of a higher-level declarative language.

Unfortunately, the NoSQL approach is not without its own problems. It sacrifices several benefits, such integrated support for backup/recovery and other manageability concerns, which can be easily dismissed as implementation artifacts of early projects. However, these new storage systems are sometimes based on questionable architectural decisions. Specifically, the idea that programmers should write imperative code to access data and maintain indexes means they lose important features like physical data independence and automatic query optimization.

PIQL (pronounced 'pickle') seeks to challenge the belief that declarative query languages are somehow less scalable than hand-coded imperative code by providing developers with a *performance predictable* language subset of SQL. PIQL 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 guaranteeing strict bounds on the number and size of I/O operations that must be performed for any query. Coupled with a distributed key/values store's ability to provide high-quantile reliable performance per operation, these bounds allow PIQL to provide the low-latency queries that its motivating applications demand. Features of the initial PIQL implementation 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 heterogeneous schemas 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.

Having described the motivation for the PIQL language we will now discuss its initial syntax, the methods it uses to guarantee bound on the number of I/O operations for all queries, and present some initial performance results.

2. BACKGROUND

2.1 Alternative Approaches

Existing approaches for supporting large-scale interactive applications focus primarily on complex abstraction layers written by the application’s developer and/or on 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 [10]. The Cassandra data model is defined as a four or five dimensional hash table whose *keys* point to *rows* which contain *super columns* which contain *column families* which contain *columns* which consist of values with an optional timestamp. Take, for example, a feature that allows users to search for messages that contain a certain word. The developer must manually create an inverted index over the contents on the message by inserting a value for each word in each message that is received by a user.

In contrast to the complexity of the inbox search implementation based on the Cassandra data model and its lack of a query language, this same example could be expressed in PIQL as:

```

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

```

Additional approaches include GQL, 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 our PIQL implementation is 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.

Finally, while efforts such as Pig strive to provide a declarative language for manipulating data that is not stored in a traditional RDBMs they are focused primarily on batch analytics and not interactive applications [12].

2.2 Assumptions

PIQL was designed with two primary 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. DeCandia and other showed that this can be done in practice up to the 99.9th percentile [7]. Additionally, for the cost

conscious, it should be possible to dynamically scale the system up and down using utility computing and the methods described in [3]. While 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 bound the number of I/O operations, albeit with higher latency, on top of a properly provisioned disk based store.

The second set of assumptions concerns the applications that will be using PIQL. The language was inspired by applications such as eBay, Facebook, and Twitter, and as such we assume that all queries have soft real-time completion requirements on the order of milliseconds due to the aforementioned effects of page load latency on user behavior [14]. While there are classes of analytic queries that are interesting and have different requirements, these are not currently the focus of the PIQL language. Additionally, while the current implementation of PIQL relies on trading strong consistency for performance, this is an option rather than a requirement of the language. In Section 5.4 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. THE PIQL 1.0 LANGUAGE

In PIQL, data is stored in strongly-typed *entity sets*, which are analogous to relations in a traditional RDBMS. To illustrate the PIQL language, we will use the sample application SCADr[4], 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. Figure 1 shows examples of the data definition language (DDL) to create the entities that are used in the SCADr sample application. In the rest of this section, we will describe the data manipulation language (DML) of PIQL and describe how we ensure a bounded number of I/O operations for each query.

3.1 Insert and Update

The model for inserting and updating entities is slightly different than the SQL model. PIQL will only guarantee the eventual 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 restricted 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.

3.2 Simple Queries

All queries in a PIQL application are pre-specified in a manner analogous to stored procedures. This approach 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.

```

ENTITY user
{
  string name,
  string password,
  string email,
  string hometown
  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 entities for the SCADr Web application.

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. Consider 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 comprise 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.

A slightly more complicated example would look up a users 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 its presence is 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.

3.3 Joins

The PIQL language also allows equi-joins against pre-specified relationships between entities. For example, we might want to return a list of the most recent thoughts owned by a particular user:

```

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.

3.4 Pagination

Sometimes a simple limit clause is too restrictive. For ex-

ample, 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 bounded I/O promise through the `PAGINATE` operator, which can be used in place of the `LIMIT` clause and causes the system to return an easily serializable iterator. This iterator 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.

3.5 Developer Specified Cardinality Constraints

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 at most 5000 friends. By allowing the compiler/optimizer to be aware of such cardinality restrictions, we can make PIQL more expressive while still maintaining the bounded I/O 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:

```

QUERY thoughtstream
FETCH thought
  OF user AS friend BY owner
  OF subscription BY target
  OF user AS me BY owner
WHERE me.username=[1:username] AND approved = true
ORDER BY timestamp
LIMIT [2:count] MAX 100

```

Although 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.

3.5.1 Join Indexes

Finally, there are some queries that can only be implemented in PIQL using a join index. Figure 2 shows a query that returns a list of the most recent thoughts with a given tag.

In order to satisfy this query while still maintaining the bounded I/O promise, the system will need an index over

both the name of the tag and the timestamp. Otherwise, the system might 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 in Figure 2, PIQL can guarantee that any updates to the timestamp of a thought will result in a bounded number of update operations.

```
ENTITY tag {
  string name,
  FOREIGN KEY target REF thought MAX 10
  PRIMARY(target, name)
}

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

Figure 2: An additional entity and query that allow thoughts to be tagged with specific terms, and recent thoughts to be queried by term.

4. SCHEMA CHANGES 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 not only to allow DDL operations to be performed on a running system without sacrificing performance, but also to allow heterogeneous versions of the schema to be running concurrently on the same cluster. We accommodate this heterogeneity in the schema through the use of a self-describing record format and the option of phased deployment of new schemas. While in the past there may have been concern regarding the performance of such record layouts, encodings like Google Protocol Buffers show they can be used for performance critical applications [6].

Phased deployment of a new schema occurs in this order:

1. The application is optionally updated if it is currently using any deprecated fields or queries.
2. A new version of the PIQL schema 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 set 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

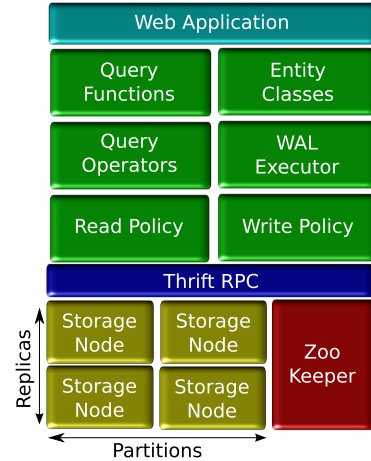


Figure 3: The architecture of the PIQL system.

any issues are encountered at any phase. Additionally, the 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. IMPLEMENTATION

Our prototype implementation of PIQL is written in Scala and it provides the aforementioned functionality for one point in the consistency-performance trade-off space. Figure 3 shows an overview of the architecture. A developer writing a web application would run the PIQL compiler on the application spec (Entities and Queries). The compiler produces a jar file 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 [3]. For example, to store and then recall 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 important components of the PIQL implementation.

5.1 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 an 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.

5.2 Write Path

The write path is invoked when a developer calls `save` on a SCADS entity. It is responsible for atomically, but eventually, updating the value stored for all replicas and indexes of the entity. The atomicity is ensured through the use of a simple write ahead log that is present at each application that has included the PIQL generated jar.

5.3 SCADS Store

The SCADS [3] store is a modular system composed of ZooKeeper [8] and set of storage 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, such as Cassandra [10].

5.4 Consistency

The relaxation of consistency is often done in response to incredibly high request rates or in response to the desire to deploy an application across multiple datacenters for latency and availability reasons. While relaxed consistency 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 replicas and indexes synchronously.

There is 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 also be used here. Conversely, the write-ahead log could be turned off if the application can tolerate occasional index inconsistencies.

There are several trade-offs when executing queries. Read policies abstract 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 concern could be alleviated.

While the current version of PIQL only implements some of the choices presented in this section we intend to use it to not only determine exactly what the consequences are to both performance and consistency as you implement different options, but also how different options interact.

6. PERFORMANCE OVERVIEW

In order to test the robustness of our ideas, we ran the SCADr application developed using PIQL on a variety of clusters 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)

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 4 shows that the latency remains virtually constant as promised.

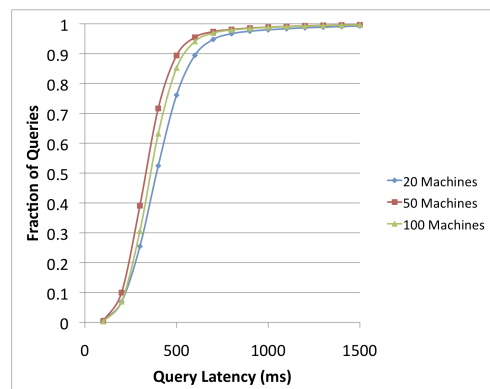


Figure 4: This CDF shows that as the system scales in both the number of users and the number of machines query latency remains relatively constant and 90% of PIQL queries are still answered in less than 600ms.

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 in which the primary key for thoughts was (timestamp, owner) and another was (owner, timestamp) as suggested by the optimizer. This optimization 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 updates to thought entities, it also has the added benefit of converting the series of index dereferencing random reads to a single sequential one. Initial performance tests show that this optimization improves that aggregate query throughput by over 70%. We believe this is only one of many multi-query optimization opportunities available.

7. CONCLUSION

Developers are increasingly abandoning the RDBMS and all of its benefits, including the ability to express queries declaratively, automatic data parallelism, physical data independence, and automatic index maintenance, in favor of performance predictable key/value stores. We have described

the PIQL language and an initial implementation, which provides developers of large-scale data-intensive applications many of these benefits while maintaining the predictable performance 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.

We plan to expand our system by investigating 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 bounds on the number of I/O operations.

Another place we plan to improve is implementing the consistency performance trade-offs discussed in Section 5.4.

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

Finally, we believe targeting existing key/value stores can potentially allay fears of data lock-in that are common with a traditional monolithic RDBMS. Other applications should be able to access and modify the data in the underlying store as long as they obey contracts regarding index consistency. This flexibility means that analytic queries could be written in another language like Pig Latin and executed by Map/Reduce [12] without first dumping the data from the production system. However, the mechanisms to allow sharing of the underlying storage engine while still providing performance isolation for applications with interactive requirements is open research.

8. ACKNOWLEDGMENTS

We would like to thank the following people, along with the reviewers, for feedback that significantly improved content of this paper: Peter Bodík, Henry Cook, Tim Kraska, Ariel Rabkin, Beth Trushkowsky, and Jesse Trutna. This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California’s MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

9. REFERENCES

- [1] GQL [online]. Available from: <http://code.google.com/appengine/docs/python/datastore/gqlreference.html>.
- [2] no:sql(east). Available from: <https://nosqleast.com/2009/>.
- [3] ARMBRUST, M., ET AL. SCADS: Scale-independent storage for social computing applications. In *CIDR* (January 2009), www.cidrdb.org.
- [4] ARMBRUST, M., ET AL. PIQL: A performance insightful query language for interactive applications. In *SIGMOD Demo Session* (June 2010), ACM.
- [5] CHANG, F., ET AL. Bigtable: A distributed storage system for structured data. In *OSDI* (2006), USENIX Association, pp. 205–218.
- [6] DEAN, J., AND GHEMAWAT, S. Mapreduce: a flexible data processing tool. *Commun. ACM* 53, 1 (2010), 72–77.
- [7] 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.
- [8] JUNQUEIRA, F. P., AND REED, B. C. The life and times of a zookeeper. In *PODC ’09: Proceedings of the 28th ACM symposium on Principles of distributed computing* (New York, NY, USA, 2009), ACM, pp. 4–4.
- [9] KELLOG, D. My thoughts on the NoSQL database “tea party” post. Available from: <http://www.kellblog.com/2010/03/09/my-thoughts-on-the-nosql-database-tea-party-post/>.
- [10] 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.
- [11] MONASH, C. NoSQL? Available from: <http://www.dbms2.com/2009/07/01/nosql-sql-alternative/>.
- [12] 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.
- [13] OUSTERHOUT, J. RAMCloud: Scalable high-performance storage entirely in DRAM. In *HPTS* (2009).
- [14] SCHURMAN, E., AND BRUTLAG, J. Performance related changes and their user impact. Presented at Velocity Web Performance and Operations Conference, June 2009.
- [15] SOBEL, J., AND ROTHSCCHILD, J. High performance at massive scale. Presented at HPTS, October 2009.