Cloud-side Execution of Database Queries for Mobile Applications

Robert Pettersen, Steffen Viken Valvåg, Åge Kvalnes and Dag Johansen
Department of Computer Science, University of Tromsø, The Arctic University of Norway, Tromsø, Norway

Keywords: Mobile, Cloud, Performance.

Abstract: We demonstrate a practical way to reduce latency for mobile .NET applications that interact with cloud database services. We provide a programming abstraction for location-independent code, which has the potential to execute either locally or at a satellite execution environment in the cloud, in close proximity to the database service. This preserves a programmatic style of database access, and maintains a simple deployment model, but allows applications to offload latency-sensitive code to the cloud. Our evaluation shows that this approach can significantly improve the response time for applications that execute dependent queries, and that the required cloud-side resources are modest.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Use of cloud-provided services is integral to the operation of modern distributed and mobile applications. In particular, cloud databases simplify application logic by serving as highly available repositories for critical state. For improved scalability and availability these databases are commonly NoSQL, with limited support for tabular relations and transactions and with a more relaxed consistency model than a conventional relational database. Queries are issued through a programmatic interface, rather than a domain-specific, high-level query language.

This promotes a usage pattern where multiple, consecutively-issued queries implement a single logical transaction. For example, an atomic update can be implemented as a read of the old value, followed by a conditional write of the new value, with the predicate that the old value remains unchanged. Or a collection of related records can be retrieved in multiple steps, by manually following foreign key references, rather than using higher-level features like joins and subqueries.

When the database is hosted in the cloud, issuing a sequence of dependent queries entails multiple round-trips of communication, and network latency becomes an important concern. For example, we have measured a latency of $50-350$ms for accessing the Amazon DynamoDB (DeCandia et al., 2007) cloud database from a mobile device (Pettersen et al., 2014), whereas a study covering 260 global vantage points reports an average round-trip time (RTT) of 74ms for accessing Amazon EC2 instances (Li et al., 2010). Issuing a sequence of queries to the cloud can result in unwanted delays that are perceptible by users.

This paper demonstrates a practical way to significantly reduce completion-latency when mobile applications execute dependent queries against a cloud database. Our approach is to provide a programming abstraction—*satellite execution*—for location-independent code, which has the potential to execute either locally on a device, or be offloaded to the cloud. If an application experiences high latency, or needs to issue a long sequence of database queries, the latency-sensitive code can be offloaded to the cloud and executed in close proximity to the database service. This ensures low-latency database access on demand, while preserving the programmatic style of database access.

Our system, called Dapper, significantly extends and integrates the functionality of two previous systems: Rusta (Valvåg et al., 2013) and Jovaku (Pettersen et al., 2014). Rusta is a platform for developing cloud applications that can utilize client-side storage and processing capacity, while the Jovaku system provides a distributed infrastructure for caching of cloud database data through the ubiquitous DNS service.

A goal with Rusta was to express computations in a location-independent way, allowing for opportune execution in the cloud or at client-side devices. This was accomplished by expressing computations in the Scala programming language and using built-in closure features to create transferable execution contexts. Dapper uses .NET reflection to achieve the

Pettersen R., Viken Valvåg S., Kvalnes Å. and Johansen D..
Cloud-side Execution of Database Queries for Mobile Applications.
DOI: 10.5220/0005451605860594
In *Proceedings of the 5th International Conference on Cloud Computing and Services Science* (CLOSER-2015), pages 586-594
ISBN: 978-989-758-104-5
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

same, thereby approaching the problem of transferability in a more general manner; any part of the execution context of a program written in any .NET supported language can be made transferable.

Jovaku's application-transparent interfacing with Amazon's DynamoDB through DNS was in part made possible by a cloud-side relay-node. Software on this node bridged DNS with DynamoDB by turning DNS requests into database queries. Dapper not only supports use of DNS for caching of data on behalf of a mobile application, but also transforms and extends the Jovaku cloud-side node into a platform for hosting and executing offloaded .NET code.

To illustrate the benefits of our approach, we quantify latency savings when cloud database queries are executed from the cloud rather than at the client-side device. We examine communication traces of popular phone applications to determine the practicality of our approach, and we measure the performance of the Dapper cloud-side platform to assess added costs.

The rest of the paper is structured as follows. Section 2 elaborates on the background and context of our work, motivating our general approach. Section 3 describes Dapper, and the programming abstractions that enable cloud-side execution of queries. Section 4 contains our performance evaluation, with measurements of typical reductions in latency, and the maximum query processing throughput that can be achieved in various configurations. Section 5 discusses related work, and Section 6 concludes.

2 BACKGROUND

The desire to reduce latency for mobile applications tends to encourage a split application architecture, where parts of the application logic executes on the device, and other parts execute in the cloud. Higher-level operations such as submitting a comment or generating a news feed are delegated as a whole to the cloud, to avoid multiple round-trips of communication.

The split between frontend and backend also has a tangential benefit: it allows a variety of frontends, often tailored for different devices, to access the same backend service. For example, an on-line chess service will typically offer both a web-based frontend, as well as clients for various mobile devices and platforms. Users should be able to switch seamlessly between client devices, e.g. moving from their laptop to their phone, so the state of on-going games must be maintained by the backend. This requires frequent communication with the cloud to retrieve and update application state.

Many frameworks and platforms aim to ease the development of mobile applications that are factored into separate backend and frontend components. One example is Parse (Parse, 2015), which provides a backend-as-a-service solution that offers backend cloud storage, as well as the ability to deploy application modules that execute in the cloud, close to the data. One common downside of these approaches is that the device-specific and cloud-specific parts of the application are deployed independently, through different channels. This increases the risk of breakage, when old versions deployed on devices interact with the newest version deployed in the cloud.

We approach the problem differently; rather than explicitly deploying parts of applications in the cloud, we empower applications to offload latency-sensitive code on demand, in a dynamic manner. Offloaded code will execute in close proximity to the backend storage service, where latency is low. Thus, we address the main motivating concern—improving application responsiveness as experienced by users—without dictating a static deployment model for applications.

A key idea underlying this work is to move computations closer to the data that they touch, which is a well-known technique that finds diverse applications. When processing streams of data, the demand for network bandwidth can be reduced by filtering streams closer to the source, pushing computations upstream. When processing stored data, similar gains can be made by scheduling computations to execute locally on the storage nodes, using functional programming models like MapReduce (Dean and Ghemawat, 2004) for location independence.

Our experience from mobile agents (Johansen et al., 2001; Johansen et al., 1999) and MapReduce-style distributed data processing have inspired some key aspects of this work. As in Cogset (Valvåg et al., 2013), we promote a functional programming model using the visitor pattern, where latency-sensitive code has the ability to *visit* the backend storage service as desired. In this case, a visitor also resembles a mobile agent; although restricted to moves back and forth between a client device and the cloud, it retains the defining ability to carry state.

3 DAPPER

Instead of statically partitioning applications into client- and cloud-side components, Dapper enables individual objects to move dynamically between the client and the cloud. This is accomplished by ex-

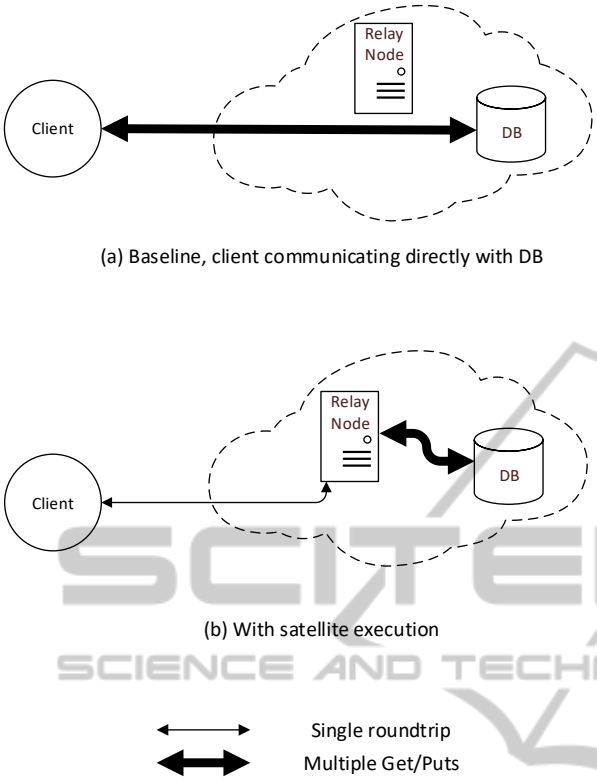


Figure 1: How *satellite execution* is applied to eliminate extraneous round-trips of communication between a client and the cloud, reducing latency.

tending the mobile platform with a *satellite execution* environment hosted on a cloud-side relay-node. Dapper implements the relay-node and provides programming abstractions for an application to temporarily execute an object at the satellite.

The decision to deploy an object for satellite execution is taken at run-time. Deployment involves moving an object's code (i.e., its class) and its current state. Incurred state changes while executing remotely are included when the object is moved back to the client. Objects can move repeatedly between the client and the cloud, for example in response to changes in application environment or state.

In this work, we demonstrate how satellite execution can reduce completion-latency for cloud database queries. Queries are expressed as objects that interact programmatically with the database. Through satellite execution the objects are deployed in close proximity to the targeted cloud database. This approach preserves the advantages of a programmatic database interface; for example, objects can perform computations, transformations, cryptographic operations, and any other manipulations of parameters and intermediate results that may be required when performing a sequence of queries. Figure 1 illustrates our approach,

```
public interface IContext
{
    Task<object> Get(string key);

    Task<List<object>> GetMany(string key);

    Task<bool> Put(string key, object value);
}
```

Figure 2: Dapper interface to key/value databases or stores.

showing how multiple round-trips between a client and the cloud can be replaced with a single round-trip to the relay-node and multiple low-latency intra-cloud interactions with the database.

Our implementation targets Amazon DynamoDB, a popular NoSQL cloud database service, but it can easily be adapted to work with any similar services. The relay-node is an Amazon EC2 instance in the same availability zone as the DynamoDB service, running an unmodified Windows Server 2012 image. The relay-node software is written in C# and uses an asynchronous programming model to efficiently handle a large number of clients and database connections.

Amazon's official C# API is used to perform DynamoDB operations. But Dapper exposes this API to the programmer indirectly, through a database context object providing general database operations, implementing the IContext interface shown in Figure 2. This indirection separates application logic from the particular database, promoting customization flexibility. For example, an application can be run fully client-side by providing a context object that binds to a client-side database.

To be eligible for satellite execution, a class must implement the ISatellite interface. Figure 3 shows an example implementing a bag-of-queries capable of satellite execution, given a database context. Queries are added to the bag by invoking AddQuery(); the queries are aggregated in the _queryList field. Execute() issues the queries via the database context object and stores results in the _responseList field. The client can retrieve query results by invoking GetResponses().

An object is moved for execution at a relay-node when the application invokes the Dapper run-time's ExecuteAt() function, shown in Figure 4, specifying the object and the particular satellite execution environment. ExecuteAt() transfers the object, in a serialized state, to the relay-node, where the object is deserialized and its Execute() function is invoked. After the Execute() function completes, the object is moved back to the client. Dapper exposes .NET task-based asynchronous programming primitives, as shown in

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
[Serializable]
public class QueryBag : ISerializable, ISatellite
{
    private List<string> _responseList;
    private List<string> _queryList;

    public async Task Execute(IContext ctx)
    {
        foreach (var query in _queryList)
        {
            var queryResponse =
                await ctx.GetMany(query);
            if (_responseList == null)
                _responseList = new List<string>();

            _responseList.AddRange(queryResponse);
        }
    }

    public void AddQuery(string query)
    {
        if (_queryList == null)
            _queryList = new List<string>();

        _queryList.Add(query);
    }

    public List<string> GetResponses()
    {
        return _responseList;
    }
}
```

Figure 3: Implementation of a bag-of-queries that can execute remotely in the cloud via satellite execution.

```
async Task<object> ExecuteAt(object obj,
                             Node location = null)
```

Figure 4: Interface for requesting remote execution.

figures 3 and 4, for the application to determine completion of remote execution and for the remote environment to efficiently handle actual execution.

Dapper employs several techniques to reduce the amount of data communicated between the client and the relay-node. One technique is to cache previously received assemblies at the relay-node. Thus, if instances of the same class are moved repeatedly, the corresponding byte codes need only be communicated once. Assembly-versioning determines the validity of a cached assembly. Another optional optimization is to communicate only changed state back from the remote environment—when remote execution completes, Dapper determines the difference in object state before and after execution and communicates that difference back to the client for object reconstruction. This is implemented using a custom serialization protocol; default serialization offers more convenience and is employed unless otherwise specified.

A user typically assigns different levels of trust to applications hosted on the same mobile device. For example, the user could entrust one application with access to data such as a contact list, but deny that access to another application. It is important for satellite execution not to weaken enforcement of this differentiated trust. For example, if the relay-node provides no isolation between execution environments, code executed on behalf of one application could potentially access code and data belonging to another application, thereby compromising trust assigned by the user at the mobile device.

Dapper relies on .NET application domains (Application Domains, 2015) to create separate and isolated execution domains for received assemblies at the relay-node. Each of these domains is configured with a whitelist of library-assemblies that are available to the hosted assembly. Also, some library-assemblies are made partially available subject to call-interception and approval.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4 EVALUATION

Our relay-node was hosted on two types of Amazon EC2 instances in our experiments. The first type was t1.micro, equipped with 613 MB memory and a single-core 64-bit vCPU operating at 1.85 GHz. The second type was t2.medium, equipped with 4 GB memory and a two-core vCPU operating at 2.50 GHz. Both types of instances were running Microsoft Windows Server 2012 R2. We used Amazon's DynamoDB as the cloud-side database, instantiated in the same availability zone as our relay-node.

Dapper runs on a variety of Microsoft Windows platforms, including phone, store, and desktop. We used two different client-side platforms for the experiments: (1) a phone with 2 GB memory and a four-core QualCom Snapdragon 800 2.2 GHz CPU and (2) a desktop machine with 64 GB memory and a four-core Intel Xeon E5-1620 3.7 GHz CPU. The phone ran Windows Phone 8.1 and communicated over 4G, whereas the desktop machine ran Windows 10 and was connected to a LAN.

We first report on a black-box examination of the cloud communication patterns of some popular mobile device applications. Here we sought to discover patterns consistent with sequences of dependent queries, with the motivation that satellite execution could be used in place of such interactions. We configured our phone platform to communicate through an access point instrumented to capture all ingress and egress packets. We then inspected packet traces look-

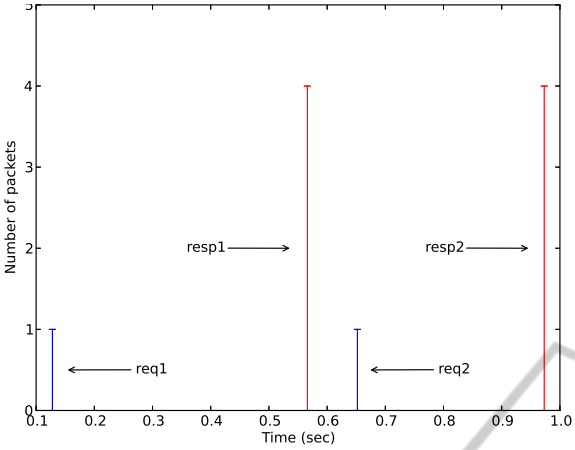


Figure 5: Example communication pattern between mobile device and cloud assumed to be of a request/reply type.

Table 1: Summary of cloud interactions during phone application startup.

Application	# request/reply	# connections
Social networking	1	1
	2	1
Instant messaging	1	4
	2	3
	7	1
Short messaging	1	7
	2	3
	4	1
	6	1
Picture exchange	1	1
	2	2

ing for what appeared as consecutive request/reply cloud interactions without intervening user actions. The particular pattern we looked for is exemplified in Figure 5, which shows two interactions assumed to be of a request/reply type.

Our findings for cloud interactions during startup of four popular applications are summarized in Table 1. We observed that the applications communicate over a number of separate network connections, ranging from 2 for the social networking application to 12 for the short messaging application. Most of these connections are to different services within the same cloud, but some are external, typically in support of content distribution such as Akamai (Nygren et al., 2010). The number of assumed request/reply interactions varied across applications and connections, with the instant- and short messaging applications respectively having as many as 7 and 6 consecutive interactions. These findings suggest satellite execution could be effective if applied in these popular applications.

We continue with an experiment that quantifies latency when a client issues cloud database queries directly and when utilizing the cloud-side relay-node. For this we used the bag-of-queries implementation outlined in Figure 3 to issue queries to the cloud database. Latency when the bag contained between 1 and 5 queries is shown in Figure 6. The figure presents results, averaged over 1000 runs, for both phone and desktop with the relay node hosted on a t1.micro instance. As shown, there are significant latency savings when the bag contains more than one query. This is because latency between the relay-node and the database is low, and the round-trip latency between the client and the cloud—approximately 64 ms for desktop and 105 ms for phone—overshadows the low cost of serializing and transferring the query bag.

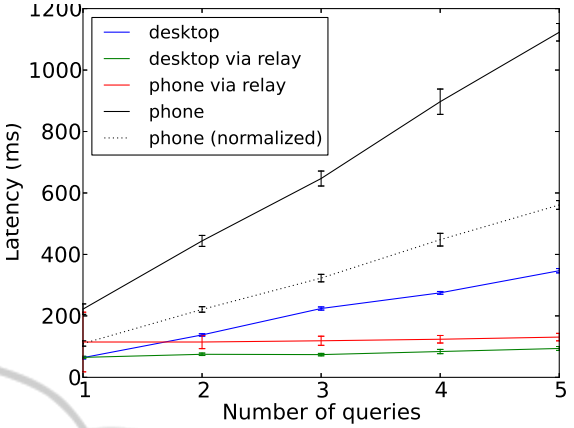
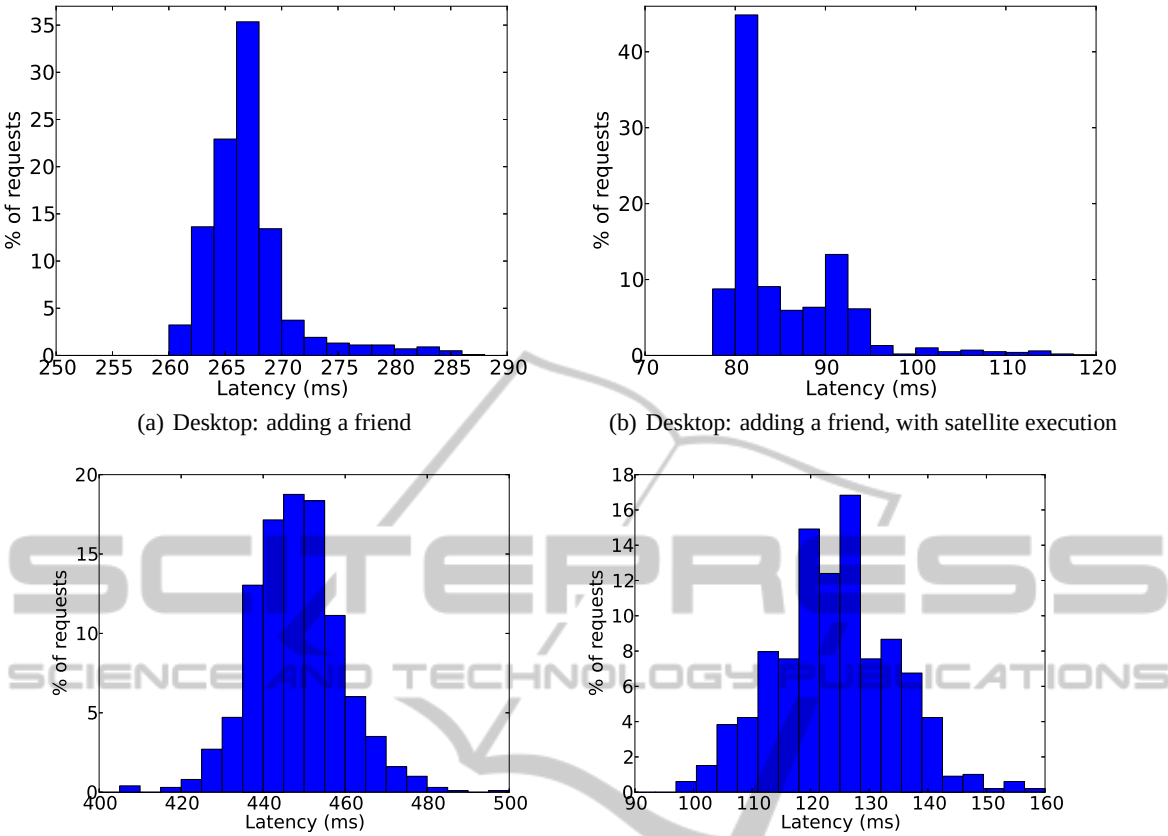


Figure 6: Increasing number of queries executed with and without satellite execution.

The DynamoDB library uses the HTTP 100-continue feature when interacting with the cloud database. Use of this feature adds a communication round-trip to database interaction, needlessly inflating latency, as described in (Pettersen et al., 2014). We therefore used platform interfaces to disable this HTTP feature on desktop. Similar interfaces do not exist on Windows Phone, however. The results in Figure 6 consequently include one additional round-trip latency for phone, compared to desktop. To better convey the latency difference between phone and desktop, the figure also includes results where one round-trip latency has been subtracted from phone. Even after this normalization, phone has significantly higher latency than desktop, demonstrating the relative importance of our satellite execution technique for the mobile platform.

The data on popular applications in Table 1 only indicates that latency savings are possible; determining the degree to which the interaction could exploit satellite execution would require access to application source code. To approximate the savings that could be experienced in a deployed application we reconstruct



(a) Desktop: adding a friend

(b) Desktop: adding a friend, with satellite execution

(c) Mobile: adding a friend

(d) Mobile: adding a friend, with satellite execution

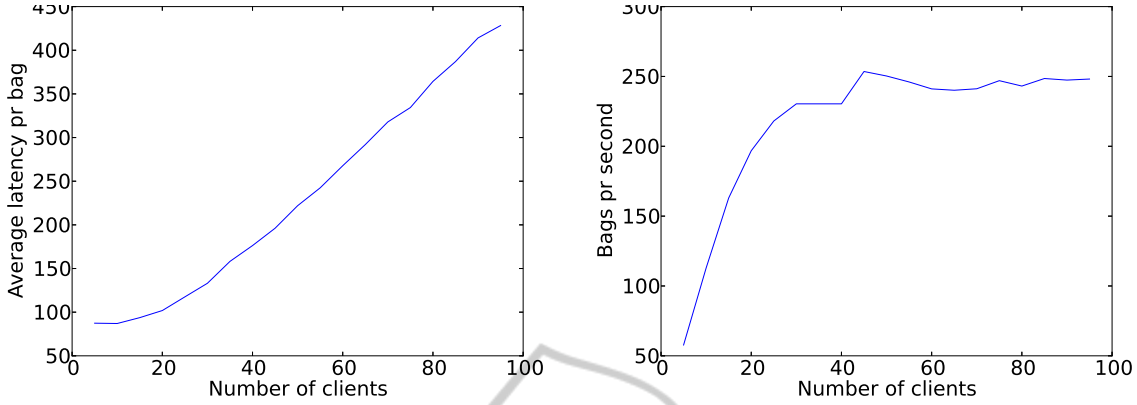
Figure 7: Latencies when adding a friend to a social network, with and without Dapper's satellite execution.

a scenario where a friend connection is established in the MSRBook, a social networking application based on Deuteronomy (Levandoski et al., 2011). The addition of a friend in this network involves friend and news feed updates for both concerned parties, for a total of 4 queries. Equivalent queries were placed in our bag-of-queries and we ran the friend-add action 1000 times on both the desktop and the mobile platform, with and without satellite execution. Figure 7 illustrates latency savings. Savings due to satellite execution are pronounced; on desktop latency drops from around 265 ms to approximately 100 ms, while it drops on mobile from around 450 ms to approximately 125 ms.

On a mobile device such as a smartphone, a person uses around 24 different applications every month (Nielsen, 2014). Even the modest resource allocations available to the Amazon t1.micro instance used in our experiments are likely to be ample for a relay-node dedicated to a single mobile device. But if the relay-node functionality was a service offered by the cloud database provider, in a fashion similar to the Parse application module service (Parse, 2015), the relay-node would likely be shared among many mobile devices and its capacity would be an issue. We therefore last consider an experiment where the relay-node serves an increasing number of mobile devices.

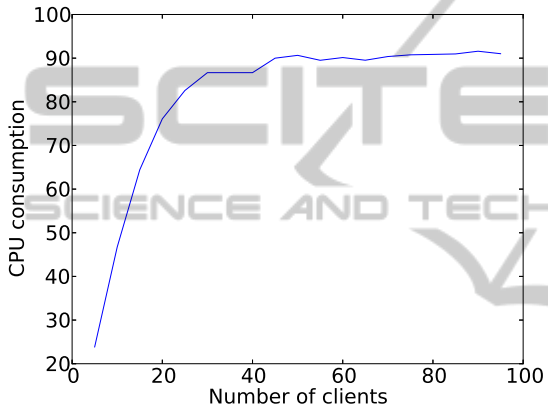
In the experiment we configured each client (i.e. mobile device) with a 4-query bag at the relay-node. Queries in these bags were repeatedly executed, ensuring high contention for relay-node resources. We then increased the number of clients, in an attempt to reveal relay-node capacity. Results for the t1.micro instance are shown in Figures 8(a)–(c). From the figures we observe that the t1 instance is capable of completing around 50 bags per second before performance tapers off. This maximum performance is likely due to CPU becoming a bottleneck, as indicated by the data in Figure 8(c). This is corroborated by t2.medium instance performance, which is shown in Figures 8(d)–(f). The t2.medium instance has approximately twice the CPU capacity of the t1.micro instance, and also completes bags at twice the rate of the t1.micro instance.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

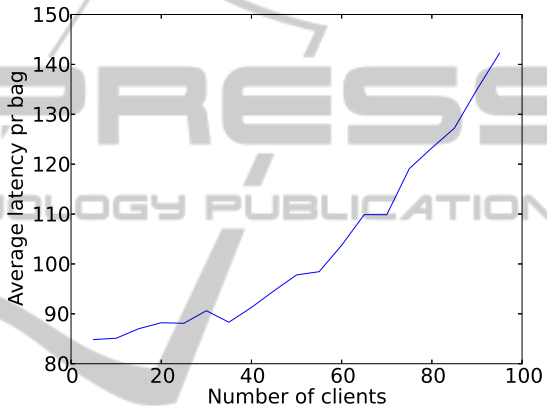


(a) Bag completion-time on t1.micro

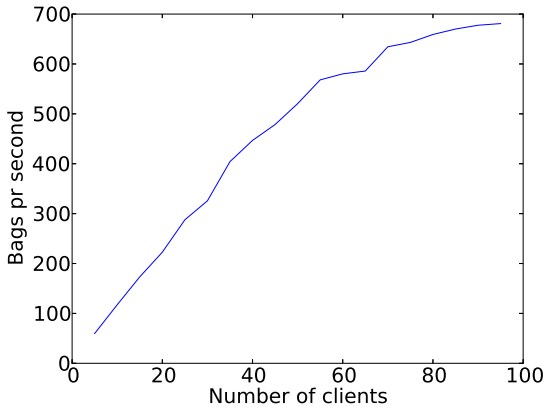
(b) Bags per second on t1.micro



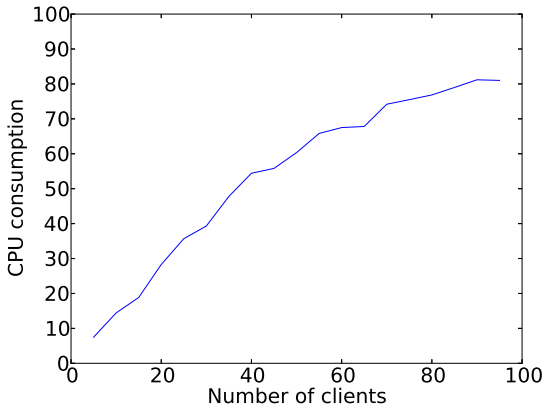
(c) CPU consumption on t1.micro



(d) Bag completion-time on t2.medium



(e) Bags per second on t2.medium



(f) CPU consumption on t2.medium

Figure 8: t1.micro and t2.medium relay-node performance

5 RELATED WORK

The complexity of developing and deploying applications that span a variety of mobile devices, personal computers, and cloud services, has been recognized as a new challenge. Users expect applications and their state to follow them across devices, and to realize this functionality, one or more cloud service must usually be involved in the background. Sapphire (Zhang et al., 2014) is a recent and comprehensive system that approaches this problem by making deployment more configurable and customizable,

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

separating the deployment logic from the application logic. The aim is to allow deployment decisions to be changed, without major associated code changes. Applications are factored into collections of location-independent objects, communicating through remote procedure calls. Like Sapphire, Dapper provides a location-independent programming abstraction, but preserves a monolithic application structure, which allows the application to be installed in its entirety on a single device through a regular distribution channel like an app store. Code is then transferred on demand from the device to the cloud, as objects move to the cloud to enjoy low-latency execution. The decision to visit the cloud or stay on the local device can be made dynamically, at run time.

With Dapper, we introduce relay-nodes in the cloud as an architectural tier between the cloud and mobile devices. Similar middle tiers have been proposed for example with Cloudlets (Satyanarayanan, 2013), and are implemented in code-offloading systems like COMET (Gordon et al., 2012), MAUI (Cuervo et al., 2010), and CloneCloud (Chun et al., 2011). However, the goal of these systems is often to augment mobile devices with additional computing power, or to conserve energy (Tilevich and Kwon, 2014), so the added tier may be located close to the devices, on local server machines, or wherever cheap computing power is available. In contrast, our motivation is not to offload work, but to reduce the latency of accessing cloud services, and thus the new tier sits as close to the cloud services as possible.

Concretely, Dapper reduces latency by eliminating extraneous round-trips of communication to the cloud. An alternative way to achieve that is by having cloud databases support more expressive query languages, so that more sophisticated transactions can be submitted as a single operation. Indeed, relational databases with full SQL support are part of the offerings of major cloud providers like Amazon. However, the ability to access the database via a general-purpose programming language remains appealing for its generality and flexibility. This is a lesson learned from programming models like MapReduce (Dean and Ghemawat, 2004), Oivos (Valvåg and Johansen, 2008), and Cogset (Valvåg et al., 2013), where data is accessed programmatically through user-defined visitor functions that can integrate easily with legacy code and libraries. The programming model in Dapper follows a similar philosophy, with the difference that user-defined functions are visiting a database in the cloud rather than a partition of data in a cluster.

6 CONCLUSION

This work focuses on the general issue of latency as a concern for applications that interact with the cloud, and looks specifically at scenarios where multiple consecutive queries are issued to a database in the cloud. Intuitively, latency can be reduced by shortening communication distances, so our idea is to move the location where queries are issued closer to the database. Since cloud databases commonly have programmatic interfaces, we implement a general mechanism for code-offloading to support this pattern.

Having a relay-node in the cloud, located in close proximity to the database service, has already proven to be a useful technique for caching, and beneficial for read-mostly workloads (Pettersen et al., 2014). Here, we extend the relay-node with functionality for *satellite execution*, allowing code that has moved temporarily from a mobile device to execute in an environment with low-latency database access. This gives benefits for additional workloads, which may include updates.

The key characteristic that a workload must exhibit to benefit from our approach is dependencies between queries. For example, if the results from one query are used to shape the next query, there is a dependency between the two. So long as there is no need for user interaction, a whole sequence of dependent queries can be offloaded to the cloud. By eliminating extraneous round-trips of communication, this improves response times.

To estimate the potential for improvement in real applications, our evaluation examines the communication patterns of some popular applications through a black-box technique. This has yielded some indications that dependent queries occur in practice, since sequences of up to 7 requests were observed back-to-back over the same connection on startup. Looking at a concrete implementation of a social networking application from (Levandoski et al., 2011), we found specific examples. For example, a friend request results in 4 dependent queries; when offloaded to the cloud from a phone, the completion time of a friend request dropped from 450 ms to approximately 125 ms.

Our implementation handles the practicalities of transferring assemblies of .NET code, serializing and deserializing objects, and sandboxing code that executes on the relay-node. Our evaluation gives some data points on performance: a single Amazon t1.micro instance can serve hundreds of queries per second. One such instance can thus easily handle load imposed by a large number of applications. So, we can dramatically reduce latency without disrupting

application architectures and with minimal requirements for resources in the cloud.

REFERENCES

Application Domains (2015). http://msdn.microsoft.com/en-us/library/cxk374d9%28v=vs.90%29.aspx.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA. ACM.

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010). Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA. ACM.

Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th symposium on Operating Systems Design and Implementation*, OSDI '04, pages 137–150. USENIX Association.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220.

Gordon, M. S., Jamshidi, D. A., Mahlke, S., Mao, Z. M., and Chen, X. (2012). Comet: code offload by migrating execution transparently. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA. USENIX Association.

Johansen, D., Lauvset, K. J., van Renesse, R., Schneider, F. B., Sudmann, N. P., and Jacobsen, K. (2001). A TACOMA retrospective. *Software - Practice and Experience*, 32:605–619.

Johansen, D., Marzullo, K., and Lauvset, K. J. (1999). An approach towards an agent computing environment. In *ICDCS'99 Workshop on Middleware*.

Levandoski, J. J., Lomet, D. B., Mokbel, M. F., and Zhao, K. (2011). Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133. www.cidrdb.org.

Li, A., Yang, X., Kandula, S., and Zhang, M. (2010). CloudCmp: comparing public cloud providers. In *ACM SIGCOMM*, pages 1–14.

Nielsen (2014). http://www.nielsen.com/us/en/insights/news/2014/smartphones-so-many-apps–so-much-time.html.

Nygren, E., Sitaraman, R. K., and Sun, J. (2010). The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19.

Parse (2015). http://www.parse.com.

Pettersen, R., Valvåg, S. V., Kvalnes, A., and Johansen, D. (2014). Jovaku: Globally distributed caching for cloud database services using DNS. In *IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 127–135.

Satyanarayanan, M. (2013). Cloudlets: at the leading edge of cloud-mobile convergence. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, pages 1–2. ACM.

Tilevich, E. and Kwon, Y.-W. (2014). Cloud-based execution to improve mobile application energy efficiency. *Computer*, 47(1):75–77.

Valvåg, S. V., Johansen, D., and Kvalnes, A. (2013). Cogset: A high performance MapReduce engine. *Concurrency and Computation: Practice and Experience*, 25(1):2–23.

Valvåg, S. V. and Johansen, D. (2008). Oivos: Simple and efficient distributed data processing. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, HPCC '08, pages 113–122. IEEE Computer Society.

Valvåg, S. V., Johansen, D., and Kvalnes, A. (2013). Position paper: Elastic processing and storage at the edge of the cloud. In *Proceedings of the 2013 International Workshop on Hot Topics in Cloud Services*, HotTopiCS '13, pages 43–50, New York, NY, USA. ACM.

Zhang, I., Szekeres, A., Aken, D. V., Ackerman, I., Gribble, S. D., Krishnamurthy, A., and Levy, H. M. (2014). Customizable and extensible deployment for mobile/-cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO. USENIX Association.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS