Towards Principled Design Support for Scalable OLTP Workloads

Bin Liu Junichi Tatemura Hakan Hacıgümüş
NEC Laboratories America
10080 N. Wolfe Rd. SW3-350, Cupertino, CA 95014 USA
{binliu, tatemua, hakan}@sv.nec-labs.com

ABSTRACT

Supporting online transaction processing (OLTP) workload in a scalable and elastic fashion is a challenging task. With the advent of cloud-based systems, supporting entity group based consistency is a viable, scalable, and cost-effective option. This approach remains attractive in the presence of systems supporting the highest level of consistency, due to the relative high cost and performance degradation of the latter. In this paper, we briefly introduce our on-going work for assisting application developers to design OLTP workload for entity group based systems. The goal is providing a suite of user-friendly design tools for new-breed databases to achieve scalability and elasticity.

Categories and Subject Descriptors

H.2 [**H.2 DATABASE MANAGEMENT**]: Logical Design, Systems

General Terms

Algorithm, Design, Performance

1. INTRODUCTION

With the rapid emergence of Web 2.0 and mobile applications, traditional database technologies are increasingly challenged on the scalability front and the ease of application deployment [5, 9, 3]. Many applications do not require the full ACID guarantees of relational databases and rather prefer the simplicity of key-value stores in order to achieve fast time-to-market and low-latency. In the mean time, cloud has emerged as the synonym of highly scalable and reliable systems. A natural marriage between key-value stores and the cloud has been proven a very attractive solution, where application developers can easily have the best of both worlds.

The simplicity of key-value stores does come at a price when higher level consistency is required, which is especially true for OLTP workloads. Application programers have to spend extra time to handle corner cases, which can quickly become a laborious and error-prone task. Supporting some form of transactions in key-values stores in the cloud is thus very desirable. Megastore [5] (Google) and Microsharding [9] (NEC Labs) are examples of such efforts. Both systems use the concept of "entity groups" [8]. Intuitive examples of entity groups are individual email accounts or twitter profiles. Full ACID is guaranteed inside each entity group, while transactions across entity groups are expensive and should be limited. Megastore's success has proved that entity group provides a good balance of scalability and consistency, although it poses some extra challenges for application developers who are accustomed to relational databases.

In this summary paper, we briefly introduce our on-going work for user-friendly scalable OLTP workload design. We focus on the problem we are solving and outline solution approaches. The underlying consistency model is entity groups. The goal is, given a relational workload in SQL, to assist application developers design scalable workload in an easy to use fashion. A typical application scenario is migrating applications which use the relational model to cloud-based transactional key-value stores such as Megastore and Microsharding. The rest of the paper is organized as follows. We introduce necessary background and problem definition in Sec. 2, which is followed by the challenges and approach towards the goal in Sec. 3. We then introduce related work in Sec. 4 and conclude the paper in Sec. 5.

2. BACKGROUND AND PROBLEM DEFINITION

We now introduce the necessary background and problem definition.

2.1 Background

For illustration, we use the RUBiS benchmark [2], which implements core functionalities of an auction site. An excerpt of the schema of RUBiS is shown in Figure 1. We consider the following three sample transactions:

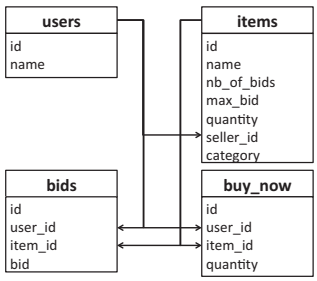

Figure 1: Rubis Benchmark Schema

EDBT 2012, March 26–30, 2012, Berlin, Germany.

- **Bidding** (T1): Insert a new tuple into *bids* and update the corresponding *items* as follows: increment the number of bids and update the maximum bid if the new bid is the current maximum. This requires joining table *items* with *bids*.
- **View User Items** (T2): Retrieve all items offered by a user given her name. This requires joining table *users* with *items*.
- **View User Bids** (T3): Retrieve all the active bids (including items information) done by the user given a user id. This requires joining table *users* with *bids*, and *items* with *bids*.

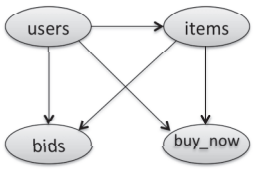

Figure 2: Rubis Benchmark Schema Graph

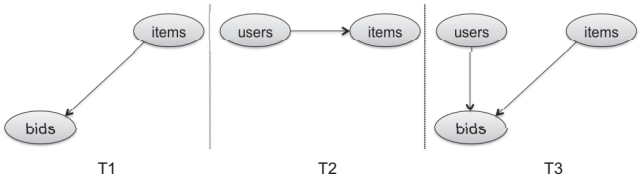

Figure 3: Sample Transaction Graphs

We can model the schema as a directed *schema graph*, where each node represents a relation, and an edge from node *A* to node *B* means that *B* has a foreign-key from *A*. The schema graph for Rubis benchmark is shown in Figure 2. If we consider only equi-joins based on foreign-key references (which are the vast majority of joins in OLTP workloads), each query (template) can also be modeled as a *query graph*, which is a sub-graph of the schema graph: the relations accessed by the query are nodes, and the joins performed by the query are edges. Since a transaction (template) consists of a set of queries that relate with each other, we can model the set of data touched by a transaction similarly as for a query. Thus we can model a transaction as a graph, which we call a *transaction graph*. The transaction graphs for the three sample transactions are shown in Figure 3. The workload is then modeled as a set of transaction graphs.

2.2 Transaction Class Design

Transaction Class (*TC*) was introduced in Microsharding [9] as a way to *declaratively* specify entity groups. We re-introduce it here due to its immediate relevance to the work that follows. TC defines a logical scope of the data where serializability must be maintained. For example, transaction T1 (bidding) updates relation *bids* and *items*, and we can create a transaction class as follows:

```
CREATE TRANSACTION CLASS TC1 AS items BY (id),
  bids BY (item_id);
```

By creating transaction class TC1, we classify the records of *items* and *bids* together into non-overlapping groups by the value of *items* id (primary key) and *bids* item_id (which is a foreign key from items). Consistency is fully guaranteed for transactions inside each group. We say transaction T1 is executed *under* TC1 when we mean that T1 has the ACID guarantees provided by TC1.

A transaction class design is given as a set of TCs. Our goal is to provide a principled support for the developer to choose a set of TCs that gives desirable trade-off between consistency and scalability.

We now introduce four properties of a transaction class and a transaction class design. First we discuss properties that are required for a valid design in order to generate mutually disjoint entity groups.

Valid class: For a TC to be *valid*, its definition must not contain two foreign-key references that end at the same relation. For example, for transaction T3, we cannot define a TC with relations *items*, *users*, and *bids*. This is because, in the presence of one-to-many relationship from *users* to *bids* and many-to-one relationship from *bids* to *items*, there is no valid way to partition relation *bids* in a non-overlapping fashion together with the other two relations. In graph terms, we cannot have a TC graph containing two join edges with the same destination. To avoid invalid TC specification, the syntax of a TC statement allows only one key for each relation.

Valid design: A valid transaction class design is a set of valid TCs with *exclusivity*, which means that a relation can be included in only one TC at the same time. Otherwise, entity groups will not be mutually disjoint. For example, both T1 and T2 access relation *items*. The best possible TC for T2 would be:

```
CREATE TRANSACTION CLASS TC2 AS users BY (id),
  items BY (seller_id);
```

TC1 and TC2 cannot be in the same time design due to overlap on *items*, and hence we must decide whether TC1 or TC2 gets *items*. Suppose TC1 stays intact, TC2 has to be modified to include only relation *users*, and transaction T2 will have no coverage over join (*users, items*).

The definition of TC can be modeled as a sub-graph of the schema graph. The graph for TC1 contains node *bids* and *items*, and there is an edge from *items* to *bids*, denoting the foreign-key based join. A valid TC is a tree where the direction of an edge from a parent to a child is consistent with the foreign-key relationship. A valid transaction class design can be modeled as a set of trees that is a subgraph of the schema graph. We call this graph a *design graph*.

Now we discuss properties that are desirable for a transaction class design.

Coverage: The above discussion illustrates that there may not be a valid design that can guarantee the ACID properties for all the transactions in a workload. We often need to compromise consistency guarantee by excluding relations that appear in a transaction out of the coverage of a TC. From the viewpoint of the ACID guarantees, we want to achieve a maximal *coverage* of transactions by a design. The degree of coverage is formally defined as the coverage for a transaction graph by a design graph. If we assign a weight to each join in a query, we can quantify the coverage of a design as the weighted sum of common edges of the design graph and transaction graph. If query has a join not covered by the TC, application developer has to write extra code to incorporate possible inconsistency.

In addition, we would like the design to have good system performance (based on profiling with test workloads). Based on user needs, our goal of workload design is to automatically derive transaction class designs that provide maximal coverage (and thus consistency) or performance. While the optimization goals are satisfied, the end result consists of: i) a design graph, ii) a potentially modified set of transactions based on the input workload, iii) the assignment of transactions to transaction classes, and iv) a set of compromises (non-serializable transactions) that must be taken care by the developer. It is worthwhile to reiterate that all this needs to be accomplished in a user-friendly fashion.

3. CHALLENGES AND SOLUTION FRAMEWORK

We now briefly introduce the challenges and solution approaches.

Challenges. First, for an application developer who is familiar with relational databases and SQL, it is challenging to design a workload that fits the entity group model. Megastore provides a procedural way for doing so, which is much less intuitive and user-friendly than the declarative manner database developers are accustomed to. We need to make the system user-friendly and not have a steep learning curve. Secondly, while guaranteeing ACID properties in entity groups, we need to optimize the workload to achieve better performance. Sometimes these two goals conflict with each other. Thirdly, when compromises need to be made, users should be informed of consequences intuitively. This requires analysis of consistency violations and performance profiling.

Solution Architecture. Figure 4 depicts the workflow of our proposed solution. Input schema and workload are translated into graph representations, and are then fed to the automatic design suggestion engine, where much of the analysis and design is performed. Depending on user's choice, we could optimize based on coverage or performance. We output different designs in the form of graphs. The design can be evaluated by profiling performance of transaction execution and consistency violation analysis, based on which the user can provide feedback to the design engine in order to guide the design refinement.

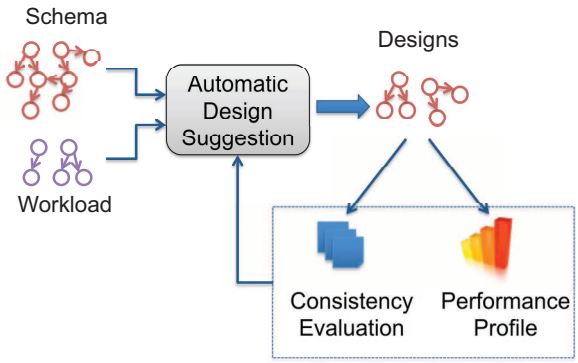

Figure 4: Solution Work Flow

4. RELATED WORK

Ever since the popularization of key-value stores by BigTable [6] and Dynamo [7], supporting transactions on them has attracted much attention. Megastore [5] is one of the earliest effort. Both Megastore and Microsharding [9] provide full ACID inside entity-groups. While Megastore requires application developers to specify entity groups in a procedural manner, Microsharding provides a declarative language similar to SQL. PIQL [3] provides a response-time guarantee for a subset of the workload (when performance can be bounded), and provides a command to traverse a subset of the result when queries are unbounded in performance. The consistency model for PIQL is eventual consistency, a design choice made partly due to the requirement of *scale independence* [4]. Recently, Amazon released DynamoDB [1] to provide higher level consistency (in addition to features such as provisioned throughput and connectivity to the MapReduce/Hadoop ecosystem). Higher level consistency access (such as consistent read) can cost double compared to eventually consistent read (because eventually consistent read can have twice the throughput) [1]. Our work can be useful for workload planning on DynamoDB to optimize for cost or response time.

5. CONCLUSIONS

In this paper, we outline the on-going work towards assisted design of scalable OLTP workload on entity group based system. While systems such as DynamoDB exist, different cost of resource and performance for different level of consistency still makes entity-group based consistency attractive, especially for resource-scarce applications that prefer quick response time rather than full consistency. Our work is the first effort for seamless migration of applications from relational system to entity-group based systems, and can also be used for workload planning for systems with full ACID guarantees. Our tools strive to minimize user's effort and learning curve by automatically deriving the best design based on the workload and optimization metrics.

6. REFERENCES

[1] Amazon DynamoDB. http://aws.amazon.com/dynamodb/.

[2] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *IEEE International Workshop on Workload Characterization*, pages 3–13, 2002.

[3] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.

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

[5] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[8] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[9] J. Tatemura and H. Hacigumus. A declarative approach to support elastic OLTP workloads. In *LADIS, The 5th Workshop on LargeScale Distributed Systems and Middleware*, 2011.