Industrial and Applications Paper

Strudel: Framework for Transaction Performance Analyses on SQL/NoSQL Systems

Junichi Tatemura
NEC Labs America
tatemura@nec-labs.com

Oliver Po
NEC Labs America
oliver@nec-labs.com

Zheng Li
Univ. of Massachusetts Lowell
zli@cs.uml.edu

Hakan Hacıgümüş
NEC Labs America
hakan@nec-labs.com

ABSTRACT

The paper introduces Strudel, a development and execution framework for transactional workloads both on SQL and NoSQL systems. Whereas a rich set of benchmarks and performance analysis platforms have been developed for SQL-based systems (RDBMSs), it is challenging for application developers to evaluate both SQL and NoSQL systems for their specific needs. The Strudel framework, which we have released as open-source software, helps such developers (as well as providers of NoSQL stores) to build, customize, and share benchmarks that can run on various SQL/NoSQL systems. We describe Strudel's architecture and APIs, its components for supporting various NoSQL stores (e.g., HBase, MongoDB), example benchmarks included in the release, and performance experiments to demonstrate usefulness of the framework.

1. INTRODUCTION

As a large number of web applications adopt cloud computing platforms, various types of "NoSQL" systems have emerged and been employed as scalable and elastic data stores. They are expected to serve transactional workloads[1] of an application that interacts with a large and varying number of users on top of commodity server resources (typically in the cloud).

Now that application developers have many choices of NoSQL systems as well as SQL systems (i.e., RDBMSs), they face various questions (which we would call "SQL-or-NoSQL questions"): When should we use a NoSQL store instead of a traditional RDBMS? How can we choose a NoSQL system that suits for our purpose? With a particular NoSQL system, what kind of trade-off do we face between scalability/elasticity gain and the cost of reduced consistency/integrity support? What about other alternatives such as purchasing a parallel RDBMS product or sharding (partitioning) open-source RDBMSs?

Standard benchmarks (such as TPC-C) have been very helpful for evaluating and choosing RDBMS products. On the other hand, the effort on benchmarking for NoSQL does not seem catching up with the evolution of NoSQL systems. Currently, YCSB [16, 12] is the most commonly used benchmark for NoSQL, but it mainly focuses on micro-benchmarking of key-value read/write operations. As NoSQL supports various features such as transaction and more complicated queries, more benchmarks are needed to capture the requirements behind such features.

It is challenging to develop benchmarks, especially to compare SQL-based systems and NoSQL systems together, given many different query APIs. It is also challenging to cover the wide range of the application needs. A transactional data store is just part of a larger application system, and its role and requirements are significantly different among applications.

Thus, we can hardly expect that a limited number of standard benchmarks are enough for transactional workloads over SQL/NoSQL systems. Given the variety of data stores and the variety of application needs, we need a way to efficiently develop a large and evolving suite of benchmarks, from micro-benchmark level to application-level, with minimum engineering effort in terms of (1) developing a new benchmark on existing SQL/NoSQL systems and (2) supporting a new SQL/NoSQL system for existing benchmarks.

In this paper we introduce our development and execution framework, called Strudel, for transactional benchmark workloads with expectation to contribute to the benchmarking effort in the community.

The design philosophy of the Strudel framework is to provide composability and reusability with (1) decomposing benchmark implementations into small components with multiple abstraction layers and (2) employing a configuration description language to combine these components into a specific workload in a reproducible and shareable manner. In order to bridge the gaps among various data stores, the framework provides multiple abstraction layers: most notably Entity DB API and Session Workload framework. A configuration description language is adopted in the framework to enable developers to compose a system and workload with custom properties in XML.

We have used and kept extending the framework for years through our research and product development of elastic relational stores (SQL engines on top of KVS [25]). As it

[1]We focus on user-facing transactional application workloads instead of analytic ones.

has matured as a generic and extensible framework, we recently released it (including benchmarks and SQL/NoSQL supports as described later) as open source software[9] for wider development purposes.

In this paper, we describe the design and architecture of the Strudel framework, its support on various NoSQL systems, benchmark examples we have developed, and performance experiments to demonstrate usefulness of the framework.

2. RELATED WORK

YCSB The Yahoo! Cloud Serving Benchmark (YCSB) [16, 12] is the most commonly used benchmark for NoSQL systems. However, from our objective to conduct performance studies on transactional aspects of SQL and NoSQL systems, the original YCSB has very limited support of transactional workloads. Researchers have extended YCSB to experiment transactions over multiple key-value objects (e.g., [17]). With an appropriate framework, we should be able to share such custom efforts to serve for more general application performance analyses.

As NoSQL systems evolve from a simple Key-Value store, there are increasing needs of performance studies with higher-level (i.e., application-level) workloads, which are especially important to compare SQL systems and NoSQL system from the application developers' viewpoints. YCSB++ [23] extends the original YCSB to evaluate advanced features of NoSQL systems (HBase and Accumulo). The features that are relevant to transactional workload evaluation include (1) pushing filtering to the data store, (2) measurement of data staleness due to weak consistency. One possible idea is to integrate such features with the Strudel framework to evaluate advanced NoSQL features not only for micro-benchmarks but also for application-level benchmarks.

OLTP-Bench OLTP-Bench [18, 6] is an extensible testbed for benchmarking RDBMSs for (primarily) transactional workloads. Our framework and OTLP-Bench are not mutually exclusive but complimentary. We focus on a special class of OLTP problems where developers have "SQL or NoSQL" questions. OLTP-Bench, aiming for more general OLTP use cases, provides a lot of useful features that our framework misses, such as sophisticated (e.g., more realistically skewed) data and workload generation and a rich set of SQL workloads (including traditional ones that we do not focus on). A skilled developer may reuse these features combined with the Strudel framework.

Performance studies There have been various performance studies reported on NoSQL systems, including the the original work of YCSB [16]. For an example of performance studies from the application's viewpoint, Klein et al.[22] report their performance evaluation of NoSQL systems for a healthcare application. Their customer (a healthcare provider) requests them to evaluate NoSQL technologies for a new electronic healthcare record system to replace the current version that uses an RDBMS. They developed evaluation tests by modifying the code of YCSB to fit the application's data model. With an appropriate development framework provided, their development could have been easier. In addition, whereas their study excludes RDBMSs (according to the customer requirements), it would need more engineering effort, in general, to compare NoSQL with RDBMSs.

Floratou et al. [20] report comparative performance studies between SQL and NoSQL systems, including comparison between SQL Server and MongoDB for YCSB workloads. We hope Strudel is useful to extend such studies to cover more NoSQL systems and various benchmarks that capture application-level requirements.

3. ARCHITECTURE

Figure 1 illustrates the layered architecture of the Strudel framework that provides composability and reusability in benchmark application development. The abstraction layers are visualized as orange boxes with underscored italic words (representing the names of APIs). Among them, Java Persistence API (JPA) is a Java standard for object-relational mapping (that converts Java object manipulation to SQL queries). The Strudel framework provides the other APIs.

Entity DB is a simplified data access API that covers transactional data access features that are common in various NoSQL systems as well as relational databases (Section 4). Basic implementation of Entity DB on a NoSQL system would not be very difficult (we also provide another API, Transactional KVS, to make it easier). If a benchmark is implemented on Entity DB API, it can run on various NoSQL systems as well as RDBMSs that support the JPA standard.

The Session Workload is a framework that helps developers to implement benchmark application on different data access APIs (Entity DB, JPA, and native NoSQL APIs) by reusing the code as much as possible (Section 5).

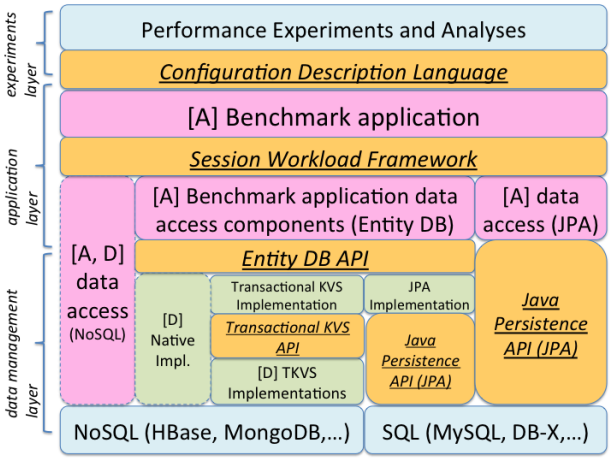

Figure 1: Layered Architecture of Strudel

In Figure 1, red boxes labeled with `[A]` are components a developer needs to implement for each benchmark application, and green boxes with `[D]` are implemented for each NoSQL system. The label `[A,D]` indicates a component to be implemented for each pair of a benchmark and a NoSQL system, and our goal is to minimize such components.

A developer can conduct a specific experiment by combining these components with a particular set of configuration parameters. We employ a configuration description language for such experiments to make experiments reproducible and individual components reusable across different experiments (Section 6).

Strudel also provides workload management and execution engines for experiments in a cluster environment in an

automated manner (Section 7).

4. ENTITY DB API

Entity DB API is one of the abstraction layers that are useful to develop workloads that can run on various data stores. It employs a subset of JPA (Java Persistence API) to fill the gap between SQL and NoSQL systems. JPA is a standard Java API for Object-Relational Mapping, providing a way to map Java objects (entities) and relational tables and a way to access data in a relational database through such Java objects.

JPA provides a basis for us to abstract out the details of underlying data stores so that application developers can focus on data handling in an object-oriented manner. However, to most NoSQL systems, JPA is not applicable directly since the concept of object-relational mapping relies on expressive power and declarativeness of SQL and the relational model.

Thus, we have designed a simplified version of APIs that consists of standard Java annotations (e.g., @Entity, @Id) of JPA and extended annotations as well as simplified data access methods. Whereas this API is not meant for application production use (missing various features required in production)[2], it would support simplified application prototypes to quickly compare data store alternatives before the real application version is developed.

4.1 Entity Group Annotations

One of the key difference of NoSQL systems from the traditional RDBMS is that not all the data items are equal in terms of transactional data access. Distributed transactions are often expensive in a commodity cluster (especially cloud) environment in order to handle a large number of concurrent read/write accesses with high availability under a short response time requirement of interactive applications. Thus, most NoSQL systems provide a way to compromise transactional consistency for scalability to avoid distributed transaction as much as possible. In a typical case, a NoSQL system allows ACID data access only on a data set associated with a single key. HBase [2] (open source implementation of Bigtable [15]) provides an atomic check-and-update operation on a single row of a (big)table. For transactions over multiple rows, we need to use an external transaction manager (e.g., Omid [19]) to implement concurrency control and recovery.

In order to provide a common API to incorporate such transaction support, we adopt the concept of *entity groups*. Helland [21] argued that, in order for an application to be truly scalable, it must forgo expensive distributed transactions; instead, each transaction must operate on a uniquely identifiable collection of data (i.e., *entity groups*) that lives on a single machine. Google Megastore [14] supports entity group as a way to associate multiple entities to a group key and guarantee efficient ACID transactions within a single group.

We introduce a set of annotations to specify entity groups in a similar way to Megastore. In addition to standard annotations (`@Entity`, `@Id`, `@IdClass`), we introduce the following annotations: `@Group`, `@GroupId`, `@GroupIdClass`.

[2]For production use, there is an open source product, for example, DataNucleus that supports common APIs for Java data persistence on some of NoSQL systems: http://www.datanucleus.org/

We illustrate how these annotations are used in an example benchmark (which emulates an auction application) in Figure 2 . The code uses the JPA standards to define Bid Java class as an entity (`@Entity`) with a compound key `(sellerId, itemNo, bidNo)` (as annotated with `@Id`), which is packaged as one object of a class BidId (`@IdClass`).

```
@Group(parent = AuctionItem.class)
@Entity
@Indexes({
  @On(property = "auctionItemId"),
  @On(property = "userId")
})
@GroupIdClass(ItemId.class)
@IdClass(BidId.class)
public class Bid {
    @GroupId @Id private int sellerId;
    @GroupId @Id private int itemNo;
    @Id @GeneratedValue
    private int bidNo;
    private double bidAmount;
    private long bidDate;
    private int userId;
```

Figure 2: Example code with annotations

With extended annotations, a benchmark developer can associate two entity classes together (as a parent-child relationship) in one group by specifying `@Group` annotation at a child class to indicate its parent. In this example, Bid is associated with AuctionItem (so that we can consistently access all the bids on one particular auction item and update the item when the maximum bid price changes). A group id (`@GroupId`) is a member of a compound key (a group key) that specifies a group instance. A set of group ids on an entity class must be a subset of the set of ids (i.e., a (compound) primary key) that are annotated with `@Id`.

4.2 Data Access Operations

CRUD Operations Entity DB API supports basic CRUD (Create-Read-Update-Delete) operations: (1) create (2) get, (3) update, and (4) delete operations (Figure 3). They (roughly) correspond to persist, find, merge, and remove operations of EntityManager, a data access interface of JPA[3].

Secondary Key Access Unlike JPA, the current Entity DB API does not support a SQL-like query language or automatic retrieval of related entities with a join column (i.e., annotations such as `@OneToMany`, `@ManyToOne`). Instead, it provides a way to read multiple instances of the same entity class by specifying one of the entity's property as a *secondary key* (`getEntitiesByIndex` in Figure 3).

Group Transactions The current version of the framework only supports a transaction within an entity group (designing API to indicate a "global" transaction is a plan for a future version). A transaction starts with a given group key and commits after multiple CRUD operations. Figure 4 shows an example of transaction execution. The application code gives an instance of EntityTask interface to the EntityDB API ("`edb.run()`"), then the underlying Entity

[3]A subtle difference from JPA is that there is no concept of attachment/detachment in the current version: an application always needs to use an update operation to apply changes in an entity Java object to the database.

```
<T> T get(Class<T> entityClass, Object key);
void create(Object entity);
void update(Object entity);
void delete(Object entity);
<T> List<T> getEntitiesByIndex(
    Class<T> entityClass, String property,
    Object key);
```

Figure 3: EntityDB Interface (partial)

DB implementation runs the instance of EntityTask by giving an EntityTranscation object (`run(EntityTransaction tx)`). The entity task (i.e., the application code) uses this transaction handler (`tx`) to issue multiple CRUD operations.

The reason behind this rather convoluted API design (instead of providing usual begin/commit operations) is to abstract out transaction retry handling, especially for optimistic concurrency control. A transaction of applications for NoSQL systems is often simple and lightweight. It is easy to retry the entire transaction when a commit request fails under optimistic concurrency control. In order to analyze the impact of transaction conflict in different NoSQL systems, we want experiment different retry policies without changing the application code.

```
BidResult r = edb.run(Bid.class, itemId,
  new EntityTask<BidResult>() {
    public BidResult run(EntityTransaction tx) {
      AuctionItem item =
          tx.get(AuctionItem.class, itemId);
      if (item == null) {
        return BidResult.NONE;
      }
      if (bid.amount() <= item.getMaxBid()) {
        return BidResult.LOST;
      }
      tx.create(bid);
      item.setMaxBid(bid.amount());
      tx.update(item);
      return BidResult.SUCCESS;
    }
});
```

Figure 4: Example of group transaction

4.3 Auxiliary Data Maintenance

Entity DB API supports two features that involve maintenance of auxiliary data in the underlying data store: (1) values for automatic unique key generation and (2) indices.

Auto Key Generation Entity DB API lets the developers specify the standard `@GeneratedValue` annotation to use a generated unique value for an id. Just as the standard JPA, an underlying implementation can choose a way to generate unique values based on the data store's capability.

The major difference from the standard specification is the uniqueness requirement: In JPA, `@GeneratedValue` is used to generate a unique value within a table. In Entity DB, it can only be *locally* unique: it is required that the compound group/primary keys that include this id as a member are unique. For example, in Figure 2, the id `bidNo` is automatically generated when a Bid entity instance is created. In fact, `itemNo` is also a generated value specified in the AutionItem class definition. The value of `itemNo` must be unique within a particular user (identified with `sellerId`), and the value of `bidNo` must be unique within a particular auction item.

This local uniqueness requirement gives the underlying data store more freedom to use a scalable and efficient way to generate values.

Indices To use secondary-key data access, the developers need to explicitly specify an index on a property of an entity. Notice that the role of an index at logical design level is different from the case with relational databases where selection of an index is logically transparent from queries. Thus, we introduce a special annotation (`@Indexes`) separated from JPA's index annotation (`@Table.indexes`).

One non-trivial semantics of this index-based entity access is its consistency under a specific entity group design. In terms of transaction isolation, an index entry can be seen another (system-defined) entity, and question is whether it is grouped together with the entities it refers to.

An index can be included (and is included by default) in the group if the compound index key (i.e., a set of ids) includes the compound group key. We call such an index an *in-group index* and otherwise we call it an *out-of-group index*. For example, in Figure 2, there are two indices on AuctionItemId (which is in fact a compound key with sellerId and itemNo) and userId. The former index is in-group (the index key equals to the group key) and the latter index is out-of-group.

If the index is not included in the group, we cannot prevent a *phantom read*: a transaction cannot read the all the bids on the same item (which is done through the index) in an isolated manner (e.g. it cannot be isolated from insertion of a new bid).

In the current Entity DB, we only allow an index on immutable columns: the value is specified only at creation of an entity instance and does not change until the instance is deleted.

4.4 Implementations

4.4.1 Generic Transactional KVS

Whereas Entity DB API is simplified for minimum support for entity data access, it still needs engineering efforts to develop an implementation for a particular NoSQL system. We provide yet another API for transactional key-value data access so that a provider of a NoSQL system can quickly implement this further simplified API instead of directly implementing Entity DB.

In Transactional KVS, a data record is just a pair of byte-array key and value, and records are grouped by a group key (another byte array). Data access is done by a group transaction (started with a group key) and simple put/get operations.

The framework provides an Entity DB implementation for Transactional KVS, which automates (1) mapping from entities to byte array key-value objects, (2) index management, and (3) auto key generation.

Index and Key generation As a baseline implementation of Entity DB API, we implemented an index and a key-generation counter as sets of key-value objects on top of the Transactional KVS data model. For each index key, we create an object with the index key and a value that encodes the pointers to the indexed entities. A counter object is created for each parent key (i.e., the compound primary

key except an ID to be generated) of an entity, and its value is a counter number.

Consider the example in Figure 2 when an application workload creates a new instance of Bid entity. Bid has two indices (`auctionItemId` and `userId`) and a generated key (`bidNo`). There are three auxiliary key-value objects updated when we create a new Bid (which is a new key-value object by itself): (a) an index object of type `auctionItemId` with key `(sellerId, itemNo)`, (b) an index object of type `userId` with key `userId`, and (c) a counter object of type `Bid` with key `(sellerId, itemNo)`. Among them only the object a is included in the group with the Bid entity, and creation of Bid involves a following sequence of three transactions:

1. updates a counter object c and acquires a new value for `bidNo`.
2. updates an index b to insert a key of Bid `(sellerId, itemNo, bidNo)` using the result of transaction 1.
3. updates an index a and creates a new key-value object for a Bid entity.

The order of these transactions is important to keep an index b consistent. Even if another transaction accesses the index b between transaction 2 and 3 (or even if transaction 3 fails), it will just read a dangling pointer in the index to a non-existent entity, which does not cause inconsistency. On the other hand, if these transactions create a Bid entity but fail to update the index b, it results in inconsistency (i.e., the existing entity cannot be accessed by the index). When the entity is deleted, the order of transactions will become opposite. When the entity is updated, we do not update the index (i.e., the current Entity DB assumes keys are immutable).

Notice also that the counter object c could have been updated in the transaction 3 if there were no out-of-group index such as b (that needs a new committed value of `bidNo` from c). In this case, creation of an entity is done by a single transaction and the counter c and index a is implemented with a single key-value object since their have the same key.

We have developed the following implementations of Transactional KVS.

HBase [2] To implement transactions, we employ HBase's check-and-put operation, which is in general called a *compare-and-swap (CAS)* operation and operates value comparison (read) and update (write) in an atomic manner.

Since check-and-put is applicable only to a single row, all the records (entities) that belong to the same group must be packed into one row. To do this, we implement each key-value record as a column name-value pair (HBase/Bigtable's columns are created in an ad-hoc manner).

In addition to these columns, each row has a special column that holds a transaction version. When the Entity DB starts a transaction, it will read the current value of this transaction version on a row that corresponds to an instance of entity group. All the updates are buffered at the client side during the transaction. At a commit request, the Entity DB issues a check-and-put operation that applies the buffered updates to the corresponding row if the current transaction version equals to the one at the beginning of the transaction.

Omid [5] Omid is a transaction server on top of HBase in order to realize ACID transactions over multiple rows [19]. It employs optimistic multi-version concurrency control using multi-versioning of HBase (each row can have multiple versions associated with (logical or physical) timestamps). For each transaction, Omid server issues a new timestamp value, with which a transaction client puts updates to HBase rows. A commit request is sent to the Omid server and ensured after conflict check and writing a log entry for recovery. Although it is a centralized server, the required computation at the transaction server is lightweight and it will not become a bottleneck for scalability easily. For our Entity DB implementation, we also implemented *sharded* Omid servers, by applying hash partitioning over the group key and routing a transaction request to one of multiple Omid servers. In our small-scale experiments up to 10 HBase region servers, however, we did not need more than one Omid server to achieve scalability (Section 8.2).

MongoDB [3] MongoDB's update operation is atomic for a single document and consists of query part and update part (i.e., a more general form of the CAS operation). Similar to the HBase implementation, we pack entities of the same group into one document. In the query part of the update, we include the document id (that corresponds to a group key) and a value of a transaction version (stored as a field in a document).

TokuMX [10] TokuMX is an enhanced version of MongoDB. One of the enhancements is to support a multi-statement transaction over multiple documents whereas the original MongoDB only supports a single statement transaction (i.e., an update operation) over a single document. A client can begin and commit or rollback a transaction. During a transaction, a client can read and update multiple documents. Isolation is achieved by locking documents (i.e., it takes pessimistic concurrency control).

One big limitation in the current TokuMX version is it does not support a multi-statement transaction for sharded document collections (i.e., partitioned data).

Our Entity DB implementation uses a cluster of independent TokuMX servers and partition data based on the group key. It emulates sharded MongoDB with application-level request routing. Since a transaction for one group key is always executable with a single server, we can employ TokuMX's multi-statement transaction.

This implementation has a limitation when it is used in practice: it does not support rebalancing of partitions (or "chunks" in MongoDB's terminology), which is one of the most important feature of NoSQL to provide elasticity.

4.4.2 Java Persistence API

The Strudel framework includes an implementation of Entity DB API using JPA so that a benchmark on Entity DB can run on any RDBMSs as long as it supports JPA. It is straightforward to implement Entity DB API using JPA since most of the features of Entity DB have the direct counterpart in JPA.

The implementation automatically translates a secondary key access to a query in JP QL, JPA's standard query language (which is then translated to SQL of a specific RDBMS).

In order to optimize physical design of the database, the developer can use any other JPA annotations. For example, selection of indices is an independent decision from the secondary key access specification (`@Indexes`) of Entity DB API: the developer specifies indices using the standard JPA (i.e., `indexes` attribute of `@Table` annotation).

4.4.3 Native Implementations

Our framework lets the developer implement a custom way to map entity access to a specific NoSQL system. We expect a future version of Strudel include such custom implementations for popular NoSQL systems.

For example, by mapping parent-child relationship to a specific data model supported by a NoSQL system, we can eliminate some of the indices specified in `@Indexes` as follows:

Nested data structure Various NoSQL systems, such as HBase and MongoDB, support a nested data structure: HBase's column family can be used to represent a set of child records (e.g., a set of bid records on a particular auction item). MongoDB's data model is a document, allowing to group entities in a flexible manner. If the secondary key to access is the parent key (e.g., `auctionItemId` index in Figure 2), we can retrieve these nested entities in one operation.

Range key access. HBase employs range partition to distribute a table and supports a range query on the row ID. By encoding parent-child relationship as a prefix of a row ID, we can efficiently implement a secondary key access (if the secondary key to use is a parent key).

5. SESSION WORKLOAD FRAMEWORK

Although Entity DB provides a common API which is reasonably implementable for many NoSQL systems, it is often too restrictive for a specific NoSQL system or an RDBMS, which have more advanced features that can contribute to higher application workload performance.

We provide another abstraction layer, *Session Workload*, at an application level for session-oriented workloads so that developers can create benchmarks that can run various data access APIs besides Entity DB API.

The Session Workload framework enables developers to build workloads that emulate interactive applications in a similar way to TPC-W [11] (emulating e-commerce) and RUBiS [8] (emulating auction) benchmarks.

Emulated user interaction for each user is called a *session*, which consists of a sequence of actions (called *interactions*). An interaction is a unit of the application's work, which is a predefined data accessing procedure without user intervention (one interaction may execute multiple transactions to perform a unit of work). A user issues a request for an interaction one by one (with optional intervals called "think time"). A user behavior is modeled as a state transition and the next interaction request is chosen based on the predefined probability and the results of the previous interactions.

The Session Workload framework makes the benchmark code reusable and customizable through the following features: (1) Interaction interface that separates data access logic and other part of benchmark code, and (2) highly configurable parameters including state transition definitions.

5.1 Interaction Interface

Figure 5 shows the interface for interactions. An interaction must implement three parts: `prepare`, `execute`, and `complete`. When an execution engine runs one interaction, it calls these three methods in this order.

The `prepare` operation is to generate a parameter that indicates a specific action that the interaction will take in the next `execute` operation. Typically, this operation emulates a thinking process of a human for this interaction (i.e. not the application side procedure). For example, an auction benchmark emulates how a bid price is decided given the current session state (e.g., information on the auction item retrieved in the past interactions).

The `execute` operation implements the actual action that accesses the data. Given the parameter (`param`) generated by `prepare` and the data access API (`db`), the method performs transactions with the data store.

The `complete` operation defines how the session state is modified based on the result of `execute` operation. For example, to emulate a human's browsing activities on a web application, the result of a browsing interaction includes a list of retrieved items. The `complete` operation may choose one of such items as "current item of interest" (i.e. part of the *state*). The modified state is used in the following interaction, which may take an action on the chosen item (e.g., placing a bid).

```
public interface Interaction<T> {
  void prepare(ParamBuilder paramBuilder);
  Result execute(Param param, T db,
                 ResultBuilder res);
  void complete(StateModifier modifier);
}
```

Figure 5: Interface for Interaction

We designed to split these methods so that we can implement a benchmark in a reusable manner for multiple data access APIs, as we describe in the following.

Generic Interaction Interface The Interaction interface employs Java's Generics to parameterize a data access API and reuse the benchmark code as much as possible. In Figure 5, the type variable `T` corresponds to a class of data access API (e.g., EntityDB and JPA's EntityManager). The application code can be written agnostic to a specific data access API as long as it does not need to know what `T` actually is. For example, the prepare method does not have to know if an interaction is used with EntityDB or any other API.

Abstract Interaction Classes To make a benchmark reusable for many data access methods, a developer is encouraged to create an *abstract interaction class* for each interaction in the benchmark. An abstract interaction class implements two methods of the interface, `prepare` and `complete`, and lets its sub-class implement the remaining `execute` method.

In the benchmarks we have developed, we implement both EntityDB and EntityManager (JPA) versions of interactions. These two implementations share majority of the benchmark code (e.g., entity definitions, data generation, workload parameter generation, state transition) (see Section 8.6 for details).

5.2 Session State Transition

A benchmark workload based on the Session Workload framework can be easily customized for a specific experiment. A state transition model that emulates a user behavior is given at run-time as an XML data. Figure 6 shows an example of an XML element (`session`) that contains state transitions (`transitions`). The `session` element typically contains various other parameters that take part of the session state in order to customize behavior of the interactions.

```
<session>
  <packageName ...>
  <Transitions>
    <transition name="START">
      <next name="HOME"/>
    </transition>
    <transition name="HOME">
      <next name="SELL_AUCTION_ITEM" prob="0.2"/>
      <next name="SELL_SALE_ITEM" prob="0.1"/>
      <next name="VIEW_AUCTION_ITEMS_BY_SELLER" prob="0.1"/>
      <next name="VIEW_SALE_ITEMS_BY_SELLER" prob="0.1" />
      <next name="VIEW_AUCTION_ITEMS_BY_BUYER" prob="0.1"/>
      <next name="VIEW_SALE_ITEMS_BY_BUYER" prob="0.1"/>
      <next name="VIEW_BIDS_BY_BIDDER" prob="0.1"/>
      <next name="VIEW_WINNING_BIDS_BY_BIDDER" prob="0.1"/>
      <next name="END" prob="0.1"/>
    </transition>
    <transition name="SELL_AUCTION_ITEM">
      <next name="HOME"/>
    </transition>
```

Figure 6: State transition in XML

5.3 Benchmarks

The current Strudel also includes example implementations of benchmarks for micro-level and application-level experiments on top of the Session Workload framework.

Micro Benchmark The Micro benchmark emulates a simplified user content management application in order to serve as a microbenchmark. The data and workload scale in terms of user IDs. To represent different patterns of user data access, the user content consists of the following four types of entities:

- *personal items* represent content privately owned by individual users. Each user has a set of items as one entity group (i.e., the number of groups scales as the number of users). An item is only read and written by its owner.
- *shared items* represent shared content written and read by the users. Items are grouped into multiple entity groups associated with set IDs (which give another scaling factor besides the user IDs).
- *public items* represent individual users' content that are open to the public for reading. An item is only written by its owner but can be read by other users.
- *message items* represent content exchanged from one user to another, having a sender ID and a receiver ID.

The benchmark defines various read-write and read-only interactions for each type of entities. A developer can compose a workload by creating a state transition that includes any subset of these interactions. By mixing interactions on these four types of entities, a developer can emulate the need of a specific application to some degree without coding a new benchmark.

In Section 8, we use *personal* items and *shared* items to demonstrate various scenarios of transaction performance analyses.

Auction Benchmark For application-level benchmarks, we have implemented an auction benchmark, which is similar to AuctionMark in OLTP-Bench [6] and RUBiS benchmark [8] but customized to use entity groups. The Bid entity in Figure 2 is part of this benchmark (shown after omitting some detailed code).

6. CONFIGURATION DESCRIPTION LANGUAGE

The Strudel framework provides abstraction layers to separate a benchmark application into various customizable pieces from a data access API implementation of a specific data store to a parameter generation of a benchmark workload. In order to combine these pieces together as one specific benchmark experiment, we employ a configuration description language that is similar to ones used for system component deployment in Grid and cloud infrastructures [13, 24, 7]. We have separately released this language as open source software called Congenio [1].

Our XML-based language supports the following features: (1) inheritance (`@extends` attribute), (2) document unfolding (`foreach` element), (3) reference resolution (`@ref` attribute), and (4) value expression (See the web site [1] for more details of the language).

With `@extends`, an experiment document can refer to existing templates (that define various components such as benchmarks and data stores) and customize the default values of these templates. With `foreach` elements, an experiment document can generate a set of documents, each of which corresponds to one workload execution with a specific set of parameters. Figure 7 illustrates such an experiment document to run an auction benchmark on HBase with different number of data servers (5, 10) and different workload scales (i.e. the number of users and worker servers).

```
<jobSuite>
  <foreach name="scale">
    <s><w>1</w><u>10000</u></s>
    <s><w>2</w><u>20000</u></s>
    <s><w>4</w><u>40000</u></s>
  </foreach>
  <foreach name="server" sep="␣">5 10</foreach>
  <job extends="auction-hbase">
    <workerNum ref="scale/w"/>
    <serverNum ref="server"/>
    <userNum ref="scale/u"/>
  </job>
</jobSuite>
```

Figure 7: Job definition in XML

The definition of an experiment in Figure 7 refers to a specific job template as illustrated in Figure 8. A job template combines various components including a workload (benchmark), database (access to data stores), and cluster (worker servers that run workloads).

```
<job>
  <workerNum>1</workerNum>
  <serverNum>1</serverNum>
  <userNum>10000</userNum>
  <threadsPerWorker>200</threadsPerWorker>
  <cluster extends="cluster">...</cluster>
  <database extends="tkvs-hbase">
     <name>auction</name>
     ...
  </database>
  <workload>
     <session extends="session-auction">
        <numOfThreads ref="threadsPerWorker"/>
        ...
     </session>
     <measure>...</measure>
   </workload>
   <report>...</report>
</job>
```

Figure 8: Example of job composition

When the execution platform (Section 7) runs an experiment with a given job definition, it records a document after inheritance resolution along with other information

(measured results, etc.). After inheritance resolution, the document includes all the information imported from other documents (referred to by `@extends`). This is very useful to reproduce the same experiments. In our lab, we use a version control system (git) to commit this document and experiment results together in the same version.

7. EXECUTION PLATFORM

Strudel's execution platform consists of the workload manager and a cluster of workers.

The workload manager starts with a given job definition XML file (in the configuration description language) and interacts with worker servers as well as servers of SQL/NoSQL systems. The workload manager has the following features:

- server configuration and start-up (invoking external scripts for NoSQL/SQL systems).
- data generation and population
- workload control and workflow management
- performance monitoring (JMX) and aggregation of the reports from workloads.
- performance reporting as JSON files.

The Strudel framework does not include the individual scripts to configure and start/stop servers since it depends on the infrastructure (e.g., whether the system is deployed on the cloud platform, a Hadoop cluster, or a simple cluster servers mounting a shared file system).

The actual workload is run by a cluster of worker nodes that receive a workload definition from the workload manager. A worker is a workload execution engine that is deployed on a cluster of server machines. The workload manager coordinates a cluster of workers to run a benchmark workload in a scalable manner (a large number of threads) to put enough load on a scalable data store.

The Session Workload is one type of workloads the workers can run. It can run any custom workloads if they implement a workload interface defined in the Strudel framework. For example, it should be easy to develop a special workload that runs the YCSB benchmark.

8. DEMONSTRATION

In this section, we demonstrate some use cases of Strudel to conduct performance experiments. Notice that the objective of the following experiments is not a formal performance study to state any conclusive claims on a particular data store but a demonstration of the features of our framework.

8.1 System Settings

In our experiments, we use the following settings.

HBase [2] We use HBase version 1.1.1 on top of Hadoop version 2.7.1. HBase servers consist of a single master server (which manages the entire system and metadata) and a set of region servers (which manages data partitions (i.e., regions)). In the experiments, we mean the number of region servers by the number of data servers. A region server is collocated with Hadoop HDFS data node (to maximize locality of I/O). The name node of HDFS is located separately in a dedicated server. HBase also requires ZooKeeper processes to achieve coordination across servers. We use 3 ZooKeeper processes collocated with the master server and two of the region servers.

Omid [5] We use version 0.8.0. Omid works with HBase and we use the same setting for HBase as the HBase-only data store. Omid supports multiple ways to persist transaction status for recovery. We use the default of the current version: storing the states on HBase. We use the same HBase cluster with the application (benchmark) workloads. In the experiment, we use only one Omid server and the number of data server refers to the number of region servers as in the case of the HBase-only setting.

MongoDB [3] MongoDB's version is 3.0.5. To employ sharding (horizontal data partitioning), we need to deploy 3 config servers (just like ZooKeeper for HBase) and a set of shard servers (which we refer to by "data servers" in the experiments). We also need "mongos" servers that route applications' requests to appropriate shard servers. We deploy one mongos server for each worker server as it is a common use case to collocate a mongos server with an application server.

TokuMX [10] For TokuMX we use version 2.0.1. As mentioned in Section 4.4, when we use multi-statement transaction with TokuMX, we cannot use sharding (automated partitioning). So we deploy a set of independent single-node TokuMX servers (which we call "data servers"), and let our EntityDB implementation route data access to these servers (emulating the application-level sharding).

MySQL [4] For experimenting JPA-based implementation of benchmarks, we use MySQL (Ver 14.14 Distrib 5.1.73) with default settings. We only use a single server MySQL in this demo.

Server machines. We use cluster machines in our lab with the following features: CentOS 6.6 (Linux 2.6.52), Intel Xeon E5620 2.40 GHz 16 core CPU , 16GB 1333 MHz RAM, Intel Pro2500 SATA SSD 240GB. Some OS parameters (e.g., the maximum number of open files) are set as data store providers recommend.

8.2 Data Store Scalability

First, we demonstrate a simple workload running on various data stores and show how these stores scale with an increasing number of data servers.

Based on the Micro benchmark, we composed a workload executing a single interaction that updates 4 *personal* items in the same group (which is randomly chosen from 1 M groups). We configure the workload so that transactions never conflict with each other: Each execution thread randomly chooses one user ID from an individual pool that is disjoint from the pools of other threads and uses the ID to choose a group (that belongs to this user).

We measure the throughput of the workloads using 1600 session concurrency (16 workers each of which runs 100 threads that keep running the update interaction without think time) for different data stores (except Omid) with changing the number of data servers (3, 5, 10). We made sure that the throughput is saturated (i.e., increasing session concurrency does not increase the throughput).

For Omid, we needed a larger number, 10800 (36 workers and 300 threads per worker), of concurrency to saturate the same number of data stores: Because one interaction takes longer time, a larger concurrency is needed to generate a sufficient number of read/write operations on the data stores. Using the Omid transaction server with HBase adds some

overhead (longer response time for each interaction and a larger number of data servers to achieve a throughput number) but it does not limit scalability at least for 10 region servers.

For MySQL we show the result of the JPA-based implementation (the figure has only one bar for a single server MySQL execution). We also ran the same workload on the Entity DB-based implementation but it did not show any significant performance difference for this simple workload (hence, it is omitted). In fact, these two implementations would generate the same SQL queries. Notice that the throughput of MySQL is good as a single data server: A 3-node NoSQL system does not achieve the same throughput *per data server*. This observation is consistent with [20], which reports the superior efficiency of an RDBMS compared to NoSQL stores. If the application workload can fit with a single data server, an RDBMS might be the most cost effective approach. If elasticity (dynamic re-balancing of partitions) is not required, purchasing a parallel RDBMS product may pay off for its efficiency.

In this demonstration, we did not cover parallel RDBMS products but it would be easy to run the same workload as long as the product is given with JPA support.

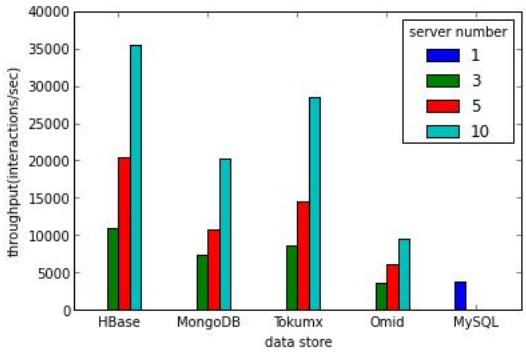

Figure 9: Throughput of item update interactions (4 items per interaction, 1600 concurrent sessions) on different number of data servers.

8.3 Transaction Concurrency

The result of the previous experiment demonstrates efficiency of a lightweight implementation of entity-group transaction with a simple check-and-update (HBase) compared to an approach with an additional transaction server (Omid). One drawback of this approach is that there is no concurrency allowed within a group (i.e., concurrent updates on the same group will fail). In many applications, this may not be a problem: when a group is associated with an individual user, a single user would not issue a large number of concurrent transactions. However it would not be always the case (e.g., auction bidding).

The next experiment we demonstrate is to see the trade-off between HBase (single-row transaction) and Omid (multi-row transaction) in terms of transaction concurrency. We use a workload that updates one *shared* item randomly chosen from a randomly chosen group. We fix the total number of items (80K) and change the size of group (400, 40, 4 items per group). From the viewpoint of the Omid transaction manager, these cases are identical (no difference between intra-group and inter-group). But for HBase, the total number of groups will decide the concurrency limit of the workloads (if two transaction updates different items in the same group, they will conflict with each other and only one can be successful). The result with 3200 concurrent sessions on 5 data servers (region servers) is shown in Figure 10. As the number of groups becomes smaller, HBase's throughput degrades and becomes worse than Omid.

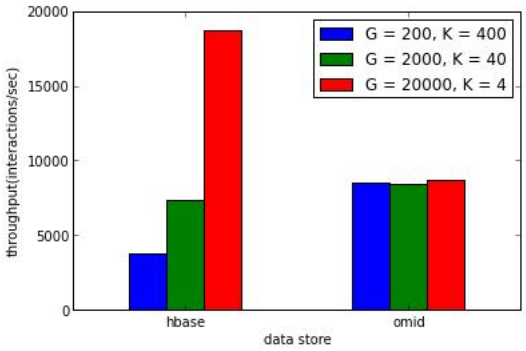

Figure 10: Throughput under different transaction concurrency: 3200 concurrent sessions, 5 data servers, 1 update/interaction, 80K items in G shared groups (K items/group)

It is beneficial to use a transaction server when the concurrency within a group needs to be high, even if it adds significant extra overhead (additional commit processing) compared to the main part of transaction (amount of read-/write),

In a real application development setting, developers will need to manage the trade-off by building workloads to emulate the application's needs and conducting similar experiments.

8.4 Transaction Conflict

Recall that TokuMX is an enhanced version of MongoDB and supports a multi-statement transaction based on locking of documents (data items). An application developer would wonder how and when this feature should be used. One interesting experiments using our framework would be comparison between optimistic concurrency control with MongoDB (single-document transactions) and pessimistic concurrency control with TokuMX (multi-document transactions).

At high-level, we know a rule of thumb, which is to take a pessimistic approach when conflict will likely happen in order to avoid unnecessary re-computation. But it always depends on a specific case.

In this example, a workload consists of a single interaction that updates 4 items in a randomly chosen group. The difference from the experiments in Figure 9 is that there are (varying degrees of) conflicts. We use 3200 concurrent sessions that update items in 3200 groups under the following three conditions: (1) 400 *personal* items per group: each thread will keep updating its own group (i.e., no conflict), (2) 400 *shared* items per group: each thread will randomly choose one of 3200 groups and choose 4 from 400 items (i.e., mild conflict), (3) 40 *shared* items per group: each thread

will randomly choose one of 3200 groups and choose 4 from 40 items (i.e., heavy conflict). The results are shown in Figure 11.

We notice the difference besides the concurrency control in the two versions: they are also different in allowed transaction concurrency (just like HBase and Omid). The MongoDB version of entity group transaction cannot have concurrency within a group. Hence, we do not see the difference between the case 2 and 3 for MongoDB. Their throughput values are almost equal to each other and are much lower than the throughput in the case 1.

On the other hand, the TokuMX version employ a lock for each item (i.e., document) and it looks very effective in the case 2 showing only slight degradation from the case 1.

However, the behavior of the TokuMX version is quite different in the case 3, showing a very low performance. In fact most of the transactions fail due to either deadlock or failure to acquire a lock, and these transaction will keep retrying until they finish successfully.

To compare optimistic and pessimistic concurrency control under heavy conflict, there is a fundamental difference in the cost of retrying transactions. In this particular case of optimistic concurrency control using CAS operations, the conflict relationship among transactions is very simple and there will be no deadlock: i.e., at least one of the conflicting transactions will "win." Although retrying involves inefficiency, there is always progress in the computation. On the other hand, the pessimistic concurrency control may suffer from deadlock, in which case nobody wins. Thus, to ensure progress of the computation, the execution threads need to *back off* and wait longer time before retrying In fact, the above result is after tuning the back-off policy using configuration options provided by the Strudel framework.

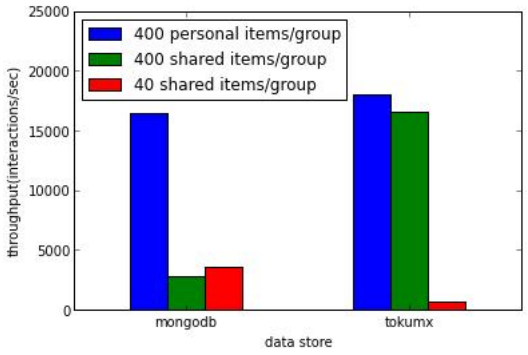

Figure 11: Throughput under different degree of conflict: 3200 concurrent sessions, 5 data servers, 4 updates/interaction over 3200 shared/personal groups

When it is very cheap to retry a transaction, the optimistic concurrency control can be an easier approach. In a practical setting, employing pessimistic concurrency control might be tricky in a cluster environment (especially when the system is built with open-source components and deployed on the cloud platform). Careful performance analysis is necessary to validate if it is really worth employing. The best approach would depend on the requirement of a specific application, and our tool can help the developer to explore various options.

8.5 Application-level Performance

To demonstrate a scenario of an application-level performance analysis, we compare HBase and MySQL using the auction benchmark. For MySQL we use two benchmark implementations based on Entity DB API and JPA, respectively.

The JPA version of auction benchmark uses join queries when they are applicable. For example, in an interaction that shows the information on all the bidding by a particular bidder, the tables of items and bids are joined together. In the auction workload, all the interactions that use join are read-only, and the number of tables joined is always 2.

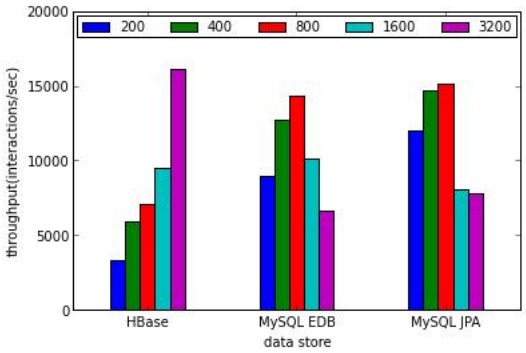

Figure 12: Throughput of auction benchmark with different session concurrency on different data stores

In this experiment, we increase the session concurrency from 200 to 3200 (200 threads per worker server) without think time on the same number of data servers (10 for HBase and 1 for MySQL). The number of users (and the size of the data set) is made proportional to the session concurrency (50 users per thread). The throughput of the workload is visualized in Figure 12.

As expected, HBase is scalable for an increasing number of concurrent user sessions. One observation, however, is that its throughput values are lower than the values of a single MySQL server when the number of concurrent sessions is small. This implies that MySQL's execution of interactions with SQL is more efficient than executing the same interactions with put/get operations of HBase.

Another observation is that the JPA-based version performs better than Entity DB-based version on MySQL when the session concurrency is small, whereas the upper limit of throughput does not seem much different between these two implementations.

To see more detail of the efficiency of interaction execution, Figure 13 visualizes the average response time of individual interactions when the session concurrency is small (200). For the purpose of presentation, we only visualize 5 interactions picked up from 15 interactions used in the workload.

First, we observe the response time of two read-write interactions: sell-auction-item and store-bid. One noticeable point is that the store-bid interaction takes much longer time than the sell-auction-item on HBase (whereas the sell-auction-item performs similarly among three data stores).

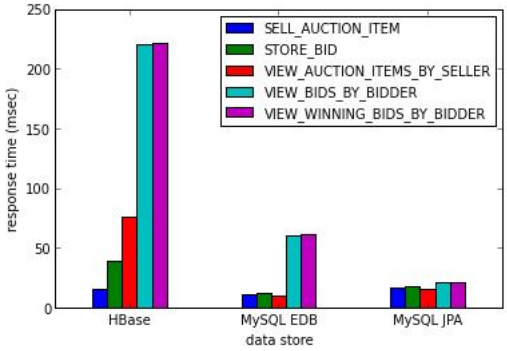

Figure 13: Response time of different interaction types in auction benchmark on different data stores (session concurrency = 200)

The store-bid interaction creates one Bid entity and updates one AuctionItem entity in one transaction (i.e. updating one row). In fact, however, creating one Bid involves two additional row updates for out-of-group auxiliary data items: a key-generation counter and an index on the bidder id. On the other hand, key-generation and index maintenance are internal operations for MySQL, adding only negligible overhead.

We see much larger difference between HBase and MySQL for read-only interactions. We picked up three read-only interactions to represent three types of queries: (1) *view-auction-items-by-seller* gets auction items with a secondary key (the user ID of a seller). It illustrates different use of an index in Entity DB and JPA, (2) *view-bids-by-bidder* gets bids by a particular bidder as well as the corresponding auction items. The JPA-version uses a two-table join query with Bid and AuctionItem, (3) *view-winning-bids-by-bidder* gets bids by a particular bidder that *won* the auction items. The JPA-version uses a two-table join query with additional filtering conditions.

The view-auction-items-by-seller interaction reveals the difference in HBase and MySQL Entity DB: MySQL uses its internal index mechanism to implement Entity DB's secondary key access, which is more efficient than an index object implemented on top of HBase. For this interaction, MySQL uses the same SQL for Entity DB version and JPA version (hence similar performance).

The view-bids-by-bidder interaction takes much longer time in Entity DB versions (MySQL and HBase) compared to the JPA-based implementation that uses a join query. However, the JPA-based implementation did not gain further benefit by adding filtering conditions for the view-winning-bids-by-bidder.

In a real development case, we need to take response time requirements for individual interactions to choose an implementation strategy. For example, 200 milliseconds for the view-bids-by-bidder interaction of HBase in the figure might not be acceptable for an interactive web application. The current implementation of this interaction executes a nested loop of get operations to emulate a join of Bid and AuctionItem. A possible improvement is to issue get operations asynchronously to hide latency of individual get responses.

8.6 Code Reusability

In addition to the above experiment scenarios, we also demonstrate the reusability of the code enabled by the Strudel framework.

Table 1 shows the size of components to implement the Entity DB interface for each NoSQL store. Each cell contains the lines of code and the number of classes (in a parenthesis). In the table, TKVS refers to the code of Transactional KVS (Section 4.4) that is commonly used by every implementation.

Table 1: The size of store components: lines of code (number of classes)

TKVS	HBase	Omid	MongoDB	TokuMX
3130 (36)	796 (6)	454 (4)	680 (4)	507 (4)

Table 2 shows the size of components to implement Auction and Micro benchmarks. The labels entity, param, and base correspond to definition of entity objects, parameters used in session interactions, and abstract interaction classes (Section 5), respectively. The remaining two columns, Entity DB and JPA, are components specific to data access APIs. The table does not include XML files that define session state transitions, which are part of configuration the developer can customize for specific experiments. The session state transition is agnostic to data access APIs.

Table 2: The size of benchmark components: lines of code (number of classes)

	entity	param	base	Entity DB	JPA
Auction	943 (9)	202 (3)	1346 (17)	1090 (18)	1043 (17)
Micro	681 (8)	212 (4)	1004 (19)	931 (19)	985 (19)

Notice that a more important point than the number of lines is *separation of concerns* achieved by the framework. For example, the benchmark components that are specific to data access APIs only need to implement individual data reads and writes that appear in the interactions.

8.7 Other Scenarios

Besides the scenarios the above demonstration covers, we have also used the Strudel framework for our research and development in a more customized manner. We developed custom components for our proprietary systems to run various experiments, including: (1) elasticity analysis to evaluate dynamic server scaling out (using a custom workflow that invokes various scripts to control data migration while a workload is running), (2) evaluation of bulk-loading APIs of NoSQL systems (using a custom workload that is not based on the session workload framework).

Especially, the elasticity analysis is essential to evaluate NoSQL systems. In a future version, we plan to include a generalized version of our custom components in the framework.

9. FUTURE WORK

We consider the following items in the future version of Strudel:

- Extended Entity DB API as a larger subset of JPA to incorporate more powerful query functionality of NoSQL (e.g. MongoDB) such as mapping parent-child entity relationship to a nested document (which enables retrieving parent and children together in one operation).
- Supporting multi-entity-group transactions on Entity DB API in a generic way to cover various solutions of multi-key transactions on NoSQL systems.
- Native EntityDB support of representative NoSQL systems such as HBase and MongoDB (Section 4.4.3).
- Better support of online analyses (e.g., an additional framework for scale-out analysis)
- Various data/workload generation (e.g. integration with such features from YCSB, OLTP-Bench).
- Better and easier integration with underlying infrastructure (e.g., software containers (e.g., Docker), resource managers (Hadoop YARN), and cloud platforms) as well as software configuration and deployment tools (e.g., Puppet [7]).

In actual applications, scalable transaction support is only part of the data management support. There are other data management features that must be considered: (1) entity search, (2) integration with analytic workloads. In either case, the developer has to choose if these functionality should be achieved by the same data store that serves transactions or done by external systems (search engines or analytic stores). Choice of SQL and NoSQL systems must take such features into account, which are beyond the scope of the current framework.

10. CONCLUSION

We introduce Strudel, a development and execution framework for transactional workloads both on SQL and NoSQL systems. Entity DB API provides a way to develop a benchmark using a common access API that is reasonably implementable on various NoSQL systems as well as RDBMS (through JPA). Session Workload framework provides another abstraction layer to decouple logic on data access (with a particular access API) from other logic in the benchmark (such as session state transition and parameter generation). We have implemented Entity DB API for various NoSQL systems by introducing a lower level API for transactional key-value access. A future version of the framework will explore custom EntityDB implementation on individual NoSQL systems to exploit advanced features of these systems (such as a query on a nested data structure).

11. REFERENCES

[1] Congenio: Configuration generation language. github.com/tatemura/congenio.
[2] HBase. www.hbase.apache.org.
[3] MongoDB. www.mongodb.org.
[4] MySQL. www.mysql.com.
[5] Omid. github.com/yahoo/omid.
[6] Otlp-bench. oltpbenchmark.com.
[7] Puppet. puppetlabs.com.
[8] RUBiS: Rice university bidding system. rubis.ow2.org.
[9] Strudel. github.com/tatemura/strudel.
[10] TokuMX. www.percona.com/software/mongo-database/percona-tokumx.
[11] TPC-W. www.tpc.org/tpcw.
[12] Ycsb. github.com/brianfrankcooper/YCSB.
[13] CDDLM configuration description language specification version 1.0. www.ogf.org/documents/GFD.85.pdf, 2006.
[14] J. Baker, C. Bond, J. Corbett, and J. J. Furman et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.
[15] 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.
[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154, 2010.
[17] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, Apr. 2013.
[18] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.
[19] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 676–687, 2014.
[20] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the elephants handle the nosql onslaught? *Proc. VLDB Endow.*, 5(12):1712–1723, Aug. 2012.
[21] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.
[22] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser. Performance evaluation of nosql databases: A case study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, PABS '15, pages 5–10, New York, NY, USA, 2015. ACM.
[23] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM.
[24] R. Sabharwal. Grid infrastructure deployment using smartfrog technology. In *Proceedings of the International Conference on Networking and Services*, ICNS '06, pages 73–, Washington, DC, USA, 2006. IEEE Computer Society.
[25] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacigümüs. Partiqle: an elastic sql engine over key-value stores. In *SIGMOD Conference*, pages 629–632, 2012.