CScale – A Programming Model for Scalable and Reliable Distributed Applications

Jose Faleiro, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani

Microsoft Research India
{t-josfal,sriram,krajan,grama,kapilv}@microsoft.com

Abstract. Today's connected world demands applications that are responsive, always available, and can service a large number of users. However, the task of writing such applications is daunting, even for experienced developers. We propose *CScale*, a programming model that attempts to simplify this task. The objective of *CScale* is to let programmers specify their application's core logic declaratively without explicitly managing distribution. *CScale* applications have simple semantics that simplify reasoning about correctness and enable testing and debugging on the single machine. In turn, the *CScale* runtime manages all aspects of execution of a *CScale* application on large clusters, including deployment, state management (replication and data partitioning) and fault tolerance. *CScale* ensures high availability by using distributed wait-free data structures to manage state. *CScale* does impose some constraints on the kind of operations clients can perform. However, we find that many real-world web applications can be naturally expressed using *CScale*.

1 Introduction

Today's connected world demands applications that are responsive, always available, and can service a large number of users. In the last few years, cloud based platforms such as Azure, EC2 and Google App Engine have democratized the infrastructure needed to host such applications, allowing anyone with internet access to deploy applications on a large cluster of machines. At the same time, these platforms expose programmers to pitfalls of distribution, such as process and network failures, imperfect messaging, and shared mutable distributed state. Writing applications that can effectively utilize these platforms and still meet user expectations is an extremely challenging problem.

In conventional web applications, the task of dealing with pitfalls of distribution is typically delegated to (distributed) databases. Databases allow shared state and integrity constraints between parts of state to be declaratively specified and support consistent access to state via transactions. However, this convenience often comes at a cost. In conventional distributed databases, transactions are built using primitives such as 2-phase commit, which introduce blocking and reduce availability, especially under network failures.

Modern distributed databases such as Dynamo, Cassandra and Azure tables (more popularly dubbed as NoSQL databases) have emerged as viable alternatives to conventional databases. These databases use replication and data partitioning for improved

R. Calinescu and D. Garlan (Eds.): Monterey Workshop 2012, LNCS 7539, pp. 148–156, 2012.

availability and throughput but do not necessarily guarantee consistent access to data. In general, the trade-off between consistency, availability and the ability to deal with network failures often permeates through entire software stack, including business logic and increases programming complexity.

In the past, the problem of increased programming complexity is often addressed using programming models and application platforms. Application platforms incorporate best practices in dealing with generic concerns such as scalability, reliability and security and allow developers to focus on application-specific logic. MapReduce and DryadLINQ are examples of programming models that have significantly simplified the task of developing distributed batch processing applications. There is an urgent need for similar programming models to manage complex consistency, availability and partition-tolerance trade-off in real time web applications.

In this paper, we describe *CScale*, a declarative programming model for building scalable distributed web applications. The objective of *CScale* is to allow programmers to specify their application's core logic declaratively without explicitly managing distribution. As we describe later, *CScale* applications have simple semantics (serializability) that simplify reasoning about correctness and enables testing and debugging on the single machine (using an emulator). In turn, the *CScale* runtime manages all aspects of execution of a *CScale* application on clusters, including deployment, state management (including replication and data partitioning), dealing with node and network failures, message delays etc. *CScale* ensures high availability by using distributed wait-free algorithms to manage state. The use of wait-free algorithms guarantees that a *CScale* application continues to service requests even if parts of the underlying systems fail. *CScale* does impose some constraints on the kind of operations clients can perform. However, as we show, many real-world web applications can be naturally expressed using *CScale*.

2 Programming Model

2.1 Language

The *CScale* programming language is a variant of Datalog (a logic programming language) and closely related to SQL. State in a *CScale* application is modelled as a set of relations. Relations fall into two categories, *base* and *derived* (as shown in Figure 1). Base relations are collection of base tuples and derived relations are functions of one or more base relations defined using relational operators (select, project, join, aggregation), negation and recursion. Consider a simple *CScale* program representing an auction application (Figure 2).

In this application, *Items* and *Bids* are base relations whereas *ValidBids*, *HighestBidAmounts* and *HigestBidders* are derived relations. As defined in line 6, a tuple <*bidId, userId, itemId, bidAmount*> belongs to *ValidBids* if there exists an item with the identifier *itemId* in the Items relation and a bid in the Bids relation such that the bid was placed before the auction end time of the item. In line 7, the relation *HigestBidAmounts* is computed by grouping valid bids by item, then selecting the maximum. Finally in line 8, the relation *HigestBidders* is derived from *HigestBidAmounts* and ValidBids a

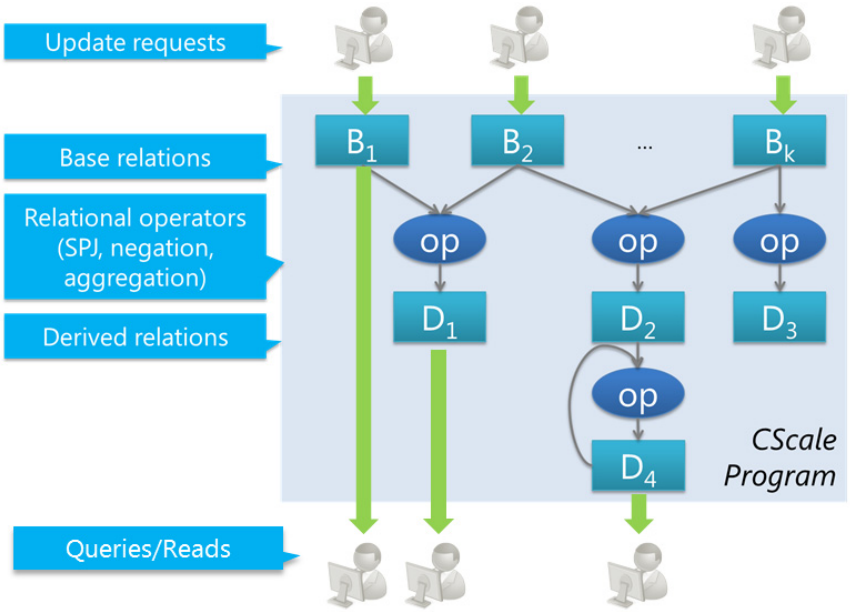

Fig. 1. *CScale* Programming Model

tuple <*bidId, userId, itemId, bidAmount*> exists in *HigestBidders* if the $bidAmount$ is highest among all bids and the bid was placed by a user *userId*.

While we have adopted a language similar to Datalog, it is also possible to express *CScale* applications in other langauges such as LINQ. Unlike batch processing frameworks like MapReduce and DryadLINQ, *CScale* applications are reactive. *CScale* relations are persistent. The lifetime of a *CScale* application extends from the point at which it is deployed on a cluster to the time it is removed. Clients can interact with a *CScale* application either by updating one or more base relations (with some constraints defined later) or by querying one or more relations (base and derived). A *CScale* application performs computation to update derived relations when base relations are modified. In some sense, this computation resembles view maintenance in conventional databases. In the auction application, the relation *Items* is updated when a new item is added or removed and the relation *Bids* is updated when a new bid is placed.

CScale relations are exposed as a RESTful WCF data service. Therefore, clients may interact with a *CScale* application directly via HTTP or using the OData protocol.

2.2 Semantics

CScale applications are designed to receive and process concurrent updates from a large number of clients. Furthermore, *CScale* relations may be partitioned or replicated for throughput and availability, potentially across different geographical locations. In spite of the concurrency and distribution, *CScale* guarantees serializability. In other words,

```
1. decl Items(itemId, itemName, itemDescription, auctionEndDateTime)
2. decl Bids(bidId, userId, itemId, bidAmount, bidDateTime)
3. decl ValidBids(bidId, userId, itemId, bidAmount)
4. decl HighestBidAmounts(itemId, bidAmount)
5. decl HighestBidders(bidId, userId, itemId, bidAmount)

6. ValidBids(bidId, userId, itemId, bidAmount) :-
      Items(itemId, itemName, ItemDescription, auctionEndDateTime),
      Bids(bidId, userId, itemId, bidAmount, bidDateTime),
      bidDateTime < auctionEndDate

7. HighestBidAmounts(itemId, highestBidAmount) :-
      GroupBy(ValidBids(bidId, userId, itemId, bidAmount),
        [itemId], highestBidAmount = max(bidAmount))

8. HighestBidders(bidId, userId, itemId, bidAmount) :-
      HigestBidAmounts(itemId, bidAmount),
      ValidBids(bidId, userId, itemId, bidAmount)
```

Fig. 2. Online auctions application expressed in *CScale*

an update to a base relation (and the computation triggered to re-compute derived relations) appears to occur atomically and in isolation from other reads and updates. In the auction application, serializability ensures that clients do not observe state where a bid appears in the Bids relation but does not reflect in the highest bids (if indeed it was the highest valid bid). Constraints As described before, *CScale* relations may be replicated and/or partitioned for availability and throughput. Achieving serializability and availability in the presence of network and node failures is a hard problem. However, *CScale* is able to meet these requirements by restricting the class of applications that can be expressed in the model. Specifically, *CScale* restricts the kinds of requests clients may issue. Clients may add or remove tuples from base relations. A set of add/remove requests to base relations may be submitted as a batch. All requests in a batch appear to occur together (though not necessarily in isolation). Clients may also specify temporal dependencies between add/remove requests and reads. However, *CScale* does not support arbitrary transactions composed of reads and writes to one or more relations. For example, a transaction that performs a read-modify-write on a base tuple is not supported. Note: In SQL terms, a *CScale* program defines views over base relations. Views have transactional semantics. However, arbitrary transactions on base relations are not permitted.

2.3 Target Applications

The *CScale* programming model is naturally suited for a rich class of web applications. These include ad serving, real time search, financial data processing, real time web analytics (vulnerability and fraud detection), blogs and social networking applications, auctions and online shopping, relaxed variants of resource allocation applications such as banking and ticketing etc.

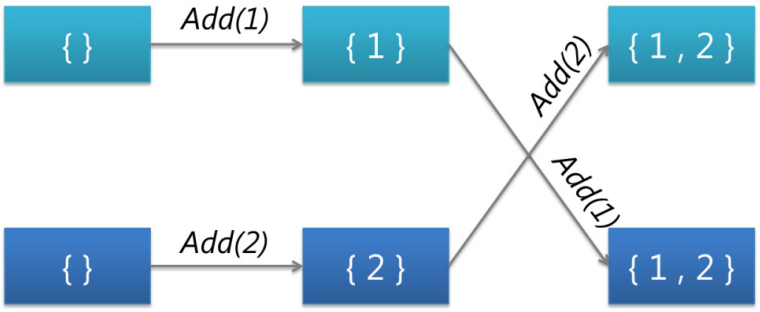

Fig. 3. A replicated set that only supports add operations

3 Implementation

An important feature of applications expressed within the *CScale* model is that they can be implemented with strong consistency and availability guarantees, even in the presence of network and process failures. *CScale* achieves this using novel distributed wait-free algorithms.

3.1 Wait-Free Data Types

CScale uses a set of distributed wait-free data types as fundamental building blocks of *CScale* applications - *CScale* relations are implemented using these data types. A key feature of these data types is that they are carefully hand-crafted to permit asynchronous, wait-free replication and still guarantee serializability. As an example, let us consider how the set data type can be implemented in this fashion. First, consider a set that only supports Add operations. Since all add operations commute, it is easy to see that if such a set were to be asynchronously replicated (i.e. operations performed on one replica are broadcast to all other replicas asynchronously), all replicas will eventually reach the same state.

Figure 3 illustrates this design. Let us assume that each replica starts with an empty state. As long as operations performed on one replica are broadcast to and applied at other replicas (in arbitrary order), all replicas are guaranteed to reach the same state (at quiescence). Hence, this set is eventually consistent. Furthermore, it can be shown that the result of all membership tests and state at quiescence can be obtained by some sequential ordering of all the add operations. Let us now try and extend this set with Remove operations. Unfortunately, Add and Remove operations on the same element do not commute. Hence, a nave implementation of the set does not satisfy the properties mentioned above because the state of the set depends on the order in which the Add and Remove operations are applied. However, in this case, these operations can be carefully designed to ensure that any two concurrent Add and Remove operations on the same element appear to occur in the same order on all replicas, irrespective of the order in which are actually applied. Figure 4 describes one such algorithm (known as the Observed Remove (OR) set) [5].

```
void Add(e) {
  S = S ∪ {e, g}
  Broadcast Add(e, g) to other replicas
}

void Remove(e) {
  G = { g | (e, g) ∈ S }
  S = S - {(e, g) | g ∈ G}
  Broadcast Remove(e, G) to other replicas
}
```

Fig. 4. Add and remove operations on a single replica in the Observed Removed set

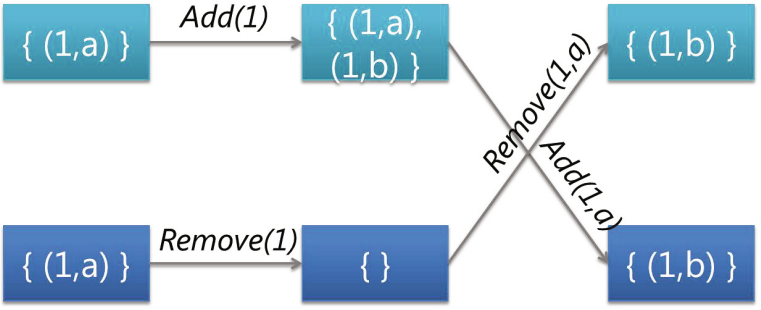

Fig. 5. An example illustrating replication in OR set

The Add operation of the OR set associates a globally unique identifier with every operation and broadcasts this identifier along with the elements. Replicas which receive the Add operation update their copy of the set in similar fashion. The Remove operation identifies the set of globally unique identifiers associated with the element (also known as the observed set) at the source replica (where it is first received from a client). It broadcasts the observed set along with the element. Replicas which receive this operation only remove tuples corresponding to the identifiers in the observed set.

Figure 5 illustrates how the OR set operates. The Remove operation on the second replica observes the identifier *a* associated with element *1* but does not observe the concurrent Add (associated with identifier *b*). The modified remove has the same effect (i.e. appears to occur before any concurrent adds) on all replicas, irrespective of the order in which it is applied. Note: Several other data types such as key-value tables and sequences can be designed to guarantee eventually consistency along the same lines as the OR set. Refer to [5] for more details.

3.2 Consistent Queries via Lattice Agreement

The set implementation described above is eventually consistent. However, intermediate queries against the set are not serializable. The example in Figure 6 shows a scenario where queries may return inconsistent values. Starting with an empty set, if queries are

serviced by multiple replicas as shown, the queries may return a sequence of values $\{\}, \{1\}, \{2\}, \{1, 2\}$, which cannot be obtained by any interleaving queries with any sequential ordering of the two Add operations.

Conventional replicated databases solve this problem by requiring replicas to agree on the order in which (non-commuting) operations are processed. This is achieved using protocols such as 2PC or Paxos [4], or even pessimistic locking. However, agreement in an asynchronous distributed system in the presence of failures has been shown to be impossible [3]. This reflects in choices made in these protocols. The Paxos protocol [4], for instance, preserves safety in the presence of failures but does not guarantee progress. Furthermore, most replication protocols essentially reduce to a form of primary master replication where all operations are first sent to a special replica known as the primary, which is responsible for ensuring that all replicas process non-commuting operations in the same order. This has significant performance implications.

In *CScale* our goal is to support asynchronous, multi-master replication. This form of replication permits each replica to receive and process operations independently, which can result in better scalability and performance. Our implementation is based on the observation that the problem of ensuring consistent reads under conditions where update operations commute can be reduced to generalized lattice agreement. In this problem, each process proposes a sequence of values from an infinite lattice, and the goal is to learn a sequence of values that form a chain. Since all update operations (adds and removes) commute, they form a lattice with set inclusion as the ordering. We can achieve consistent reads by requiring that replicas propose the set of operations they receive, and service reads from set of operations learnt via generalized lattice agreement. Since the sets of values learnt form a chain, serializability follows.

In the running example, the operations Add(1) and Add(2) form the following lattice. The use of generalized lattice agreement ensures that the values replicas learn from a chain in this lattice. Thus any sequence of reads will observe only serializable values.

Recently, we have proposed a wait-free algorithm for solving generalized lattice agreement [2]. The algorithm can tolerate process failures as long as a majority of the processes survive at any point in time. The algorithm has a complexity of O(N) message delays, where N is the number of replicas. In the absence of failures, the algorithm ensures that new operations can be learnt in 2 message delays.

3.3 Incremental Evaluation and View Maintenance

In *CScale* updates to base relations trigger computation to update derived relations. Since changes to base relations at any given moment in time are likely to be small, it is possible to re-compute derived relations more efficiently by incrementally propagating changes. Incremental evaluation is challenging for several reasons. First, operators like negation and recursion require complex algorithms for detecting changes incrementally. Furthermore, if relations are partitioned and distributed, incremental evaluation must be performed without the use of primitives such as distributed transactions [6]. Many existing systems decide to offer weak consistency for derived relations [1] for better performance, which significantly complicates reasoning.

The *CScale* runtime guarantees strong consistency of derived relations using a novel incremental evaluation protocol based on lattice agreement. In this protocol, base relations participate in an agreement protocol to order all operations on base relations. Derived relations then update state according to this order. The details of this protocol are beyond the scope of this paper.

The *CScale* runtime also supports a query optimizer that analyzes a *CScale* application and generates an optimized plan for incremental evaluation. Some of the supported optimizations include early aggregation and index maintenance.

4 Current Status and Experience

CScale proposes a fundamentally different way of building distributed applications. Applications written in *CScale* are declarative and hide low level implementation details such as messaging, failures etc. At the same time, *CScale* imposes some restrictions on how clients can interact with applications and does not (yet) provide primitives to escape the model. Therefore, our first step was to evaluate if the *CScale* model is expressive enough for real world distributed applications and if the model improves programmer productivity. We built a prototype implementation of the *CScale* system (without a distributed backend) and then re-wrote a few applications (including applications from the Windows Azure patterns and practices team such as an online survey application and an online auctions application) in *CScale*.

Our initial experiences with *CScale* are very encouraging. We were able to replace the existing storage layer and most of the application logic in these applications with a simple *CScale* program significantly smaller in the number of lines of code (from a few thousand to few 10s of lines of code). Furthermore, the application expressed in *CScale* was much more understandable and maintainable as compared to the original implementation, where the application logic was spread across multiple projects (and Azure roles) and aspects of distribution such as messaging, fault tolerance etc. was handled explicitly in code. However, we recognize that much work needs to be done to fully ascertain the shortcomings and benefits of this programming model.

A distributed implementation of *CScale* consists of many pieces a programming environment (including testing and debugging support), a compiler, a runtime system (that implements replication and partitioning on a cluster and performs query evaluation), a scalable storage layer, a platform layer that supports efficient communication and provides primitives such as network and node failure detection, a query optimizer and client side interfaces amongst others. We are in the process of building these components. We would also like to ensure that *CScale* applications can be easily deployed on cloud platforms such as Azure. Finally, we are in the process of building a suite of sample applications that illustrate the best patterns and practices for building scalable and reliable *CScale* applications.

References

1. Agrawal, P., Silberstein, A., Cooper, B.F., Srivastava, U., Ramakrishnan, R.: Asynchronous view maintenance for vlsd databases. In: SIGMOD 2009, Stanford InfoLab (June 2009)
2. Faleiro, J., Rajamani, S., Rajan, K., Ramalingam, G., Vaswani, K.: Generalized lattice agreement. In: Principles of Distributed Computing (PODC) (July 2012)

3. Fischer, M.J., Lynch, N., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
4. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16, 133–169 (1998)
5. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Convergent and commutative replicated data types. Bulletin of the European Association for Theoretical Computer Science (EATCS) (104), 67–88 (2011)
6. Zhuge, Y., Garcia-molina, H., Wiener, J.L.: The strobe algorithms for multi-source warehouse consistency. In: International Conference on Parallel and Distributed Information Systems, pp. 146–157 (1996)