Trinity Graph Engine and its Applications

Bin Shao, Yatao Li, Haixun Wang*, Huanhuan Xia
Micorosoft Research Asia, *Facebook
{binshao, yatli, lexi}@microsoft.com, *haixun@gmail.com

Abstract

Big data become increasingly connected along with the rapid growth in data volume. Connected data are naturally represented as graphs and they play an indispensable role in a wide range of application domains. Graph processing at scale, however, is facing challenges at all levels, ranging from system architectures to programming models. Trinity Graph Engine is an open-source distributed in-memory data processing engine, underpinned by a strongly-typed in-memory key-value store and a general distributed computation engine. Trinity is designed as a general-purpose graph processing engine with a special focus on real-time large-scale graph query processing. Trinity excels at handling a massive number of in-memory objects and complex data with large and complex schemas. We use Trinity to serve real-time queries for many real-life big graphs such as Microsoft Knowledge Graph and Microsoft Academic Graph. In this paper, we present the system design of Trinity Graph Engine and its real-life applications.

1 Introduction

In this big data era, data become increasingly connected along with the rapid growth in data volume. The increasingly linked big data underpins artificial intelligence, which is expanding its application territory at an unprecedented rate. Linked data are naturally represented and stored as graphs. As a result graph data have now become ubiquitous thanks to web graphs, social networks, and various knowledge graphs, to name but a few.

Graph processing at scale, however, is facing challenges at all levels, ranging from system architectures to programming models. On the one hand, graph data are not special and can be processed by many data management or processing systems such as relational databases [1] and MapReduce systems [2]. On the other hand, large graph processing has some unique characteristics [3], which make the systems that do not respect them in their design suffer from the "curse of connectedness" when processing big graphs. In this paper, we discuss the challenges faced by real-time parallel large graph processing and how to rise to them in the system design.

The complex nature of graph. Graph data is inherently complex. The contemporary computer architectures are good at processing linear and simple hierarchical data structures, such as *Lists*, *Stacks*, or *Trees*. Even when the data scale becomes large and is partitioned over many distributed machines, the *divide and conquer*

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was done in Microsoft Research Asia.

computation paradigm still works well for these data structures. However, when we are handling graphs, especially big graphs, the situation is changed. Big graphs are difficult to process largely because they have a large number of interconnected relations encoded. The implication is twofold: 1) From the perspective of data access, the adjacent nodes of a graph node cannot be accessed without "jumping" in the data store no matter how we represent a graph. In other words, a massive amount of random data access is required during graph processing. Many modern program optimizations rely on data reuse. Unfortunately, the random data access nature of graph processing breaks this premise. Without a careful system design, this usually leads to poor performance since the CPU cache is not in effect for most of the time. 2) From the perspective of programming, parallelism is difficult to extract because of the unstructured nature of graphs. As widely acknowledged [3], a lot of graph problems are inherently irregular and hard to partition; this makes it hard to obtain efficient *divide and conquer* solutions for many large graph processing tasks.

Due to the random data access challenge, general-purpose graph computations usually do not have efficient, disk-based solutions. But under certain constraints, graph problems sometimes can have efficient disk-based solutions. A good example is GraphChi [4]. GraphChi can perform efficient disk-based graph computations under the assumption that the computations have asynchronous vertex-centric [5] solutions. An asynchronous solution is one where a vertex can perform its computation based only on the partially updated information from its incoming graph edges. This assumption eliminates the requirement of global synchronization, making performing computations block by block possible. On the other hand, it inherently cannot support traversal-based graph computations and synchronous graph computations because a graph node cannot efficiently access the graph nodes pointed by its outgoing edges.

The diversity of graph data and graph computations. There are many kinds of graphs. Graph algorithms' performance may vary a lot on different types of graphs. On the other hand, there are a large variety of graph computations such as path finding, subgraph matching, community detection, and graph partitioning. Each graph computation itself even deserves dedicated research; it is nearly impossible to design a system that can support all kinds of graph computations. Moreover, graphs with billions of nodes are common now, for example, the Facebook social network has more than 2 billion monthly active users[1]. The scale of the data size makes graph processing prohibitive for many graph computation tasks if we directly apply the classic graph algorithms from textbooks.

In this paper, we present Trinity Graph Engine – a system designed to meet the above challenges. Instead of being optimized for certain types of graph computations on certain types of graphs, Trinity tries to directly address the grand random data access challenge at the infrastructure level. Trinity implements a globally addressable distributed RAM store and provides a random access abstraction for a variety of graph computations. Trinity itself is not a system that comes with comprehensive built-in graph computation modules. However, with its flexible data and computation modeling capability, Trinity can easily morph into a customized graph processing system that is optimized for processing a certain type of graphs.

Many applications utilize large RAM to offer better performance. Large web applications, such as Facebook, Twitter, Youtube, and Wikipedia, heavily use memcached [6] to cache large volumes of long-lived small objects. As the middle tier between data storage and application, caching systems offload the server side work by taking over some data serving tasks. However, the cache systems cannot perform in-place computations to further reduce computation latencies by fully utilizing the in-memory data.

The design of Trinity is based on the belief that, as high-speed network access becomes more available and DRAM prices trend downward, all-in-memory solutions provide the lowest total cost of ownership for a large range of applications [7]. For instance, RAMCloud [8] envisioned that advances in hardware and operating system technology will eventually enable all-in-memory applications, and low latency can be achieved by deploying faster network interface controllers (NICs) and network switches and by tuning the operating systems, the NICs, and the communication protocols (e.g., network stack bypassing). Trinity realizes this vision

[1] `http://newsroom.fb.com/company-info/`.

for large graph applications, and Trinity does not rely on hardware/platform upgrades and/or special operating system tuning, although Trinity can leverage these techniques to achieve even better performance.

The rest of the paper is organized as follows. Section 2 outlines the design of the Trinity system. Section 3 introduces Trinity's distributed storage infrastructure – Memory Cloud. Section 4 introduces Trinity Specification Language. Section 5 discusses fault tolerance issues. Section 6 introduces Trinity applications. Section 7 concludes.

2 An Overview of Trinity

Trinity is a data processing engine on distributed in-memory infrastructure called Trinity Memory Cloud. Trinity organizes the main memory of multiple machines into a globally addressable memory address space. Through the memory cloud, Trinity enables fast random data access over a large distributed data set. At the same time, Trinity is a versatile computation engine powered by declarative message passing.

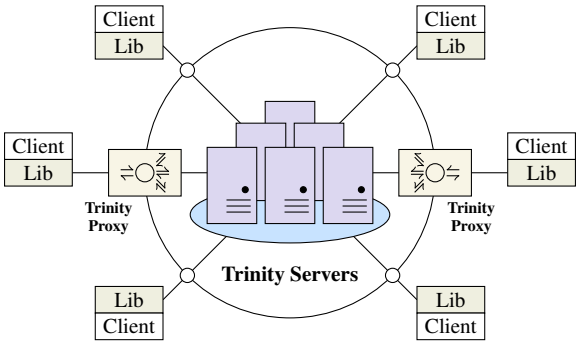

Figure 1: Trinity Cluster Structure

Fig. 1 shows the architecture of Trinity. A Trinity system consists of multiple components that communicate through a network. According to the roles they play, we classify them into three types: servers, proxies, and clients. A Trinity server plays two roles: storing data and performing computations on the data. Computations usually involve sending messages to and receiving messages from other Trinity components. Specifically, each server stores a portion of the data and processes messages received from other servers, proxies, or clients. A Trinity proxy only handles messages but does not own a data partition. It usually serves as a middle tier between servers and clients. For example, a proxy may serve as an information aggregator: it dispatches the requests coming from clients to servers and sends the results back to the clients after aggregating the partial results received from servers. Proxies are optional, that is, a Trinity system does not always need a proxy. A Trinity client is responsible for interacting with the Trinity cluster. Trinity clients are applications that are linked to the Trinity library. They communicate with Trinity servers and proxies through APIs provided by Trinity.

Fig. 2 shows the stack of Trinity system modules. The memory cloud is essentially a distributed key-value store underpinned by a strongly-typed RAM store and a general distributed computation engine. The RAM store manages memory and provides mechanisms for concurrency control. The computation engine provides an efficient, one-sided, machine-to-machine message passing infrastructure.

Due to the diversity of graphs and the diversity of graph applications, it is hard, if not entirely impossible, to support all kinds of graph computations using a fixed graph schema. Instead of using a fixed graph schema and fixed computation paradigms, Trinity allows users to define their own graph schemas, communication protocols through Trinity specification language (TSL) and realize their own computation paradigms. TSL bridges the needs of a specific graph application with the common storage and computation infrastructure of Trinity.

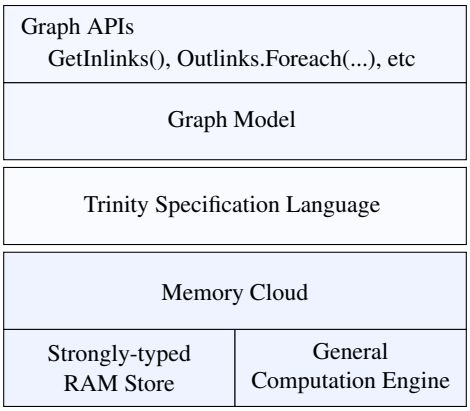

Figure 2: System Layers

3 Trinity Memory Cloud

We build a distributed RAM store – Trinity Memory Cloud – as Trinity's storage and computation infrastructure. The memory cloud consists of 2^p memory trunks, each of which is stored on one machine. Usually, we have $2^p > m$, where m is the number of machines. In other words, each machine hosts multiple memory trunks. We partition a machine's local memory space into multiple memory trunks so that trunk level parallelism can be achieved without any locking overhead. To support fault-tolerant data persistence, these memory trunks are backed up in a shared distributed file system called TFS (Trinity File System) [9], whose design is similar to that of HDFS [10].

We create a key-value store in the memory cloud. A key-value pair forms the most basic data structure of any system built on top of the memory cloud. Here, keys are 64-bit globally unique integer identifiers; values are blobs of arbitrary length. Because the memory cloud is distributed across multiple machines, we cannot address a key-value pair using its physical memory address. To address a key-value pair, Trinity uses a hashing mechanism. In order to locate the value of a given key, we first 1) identify the machine that stores the key-value pair, then 2) locate the key-value pair in one of the memory trunks on that machine. Through this hashing mechanism as illustrated by Figure 3, we provide a globally addressable memory space.

Specifically, given a 64-bit key, to locate its corresponding value in the memory cloud, we hash the key to a p-bit value i ($i \in [0, 2^p - 1]$), indicating that the key-value pair is stored in memory trunk i within the memory cloud. Trinity assigns a unique machine identifier *mid* to each machine in the Trinity cluster. To find out which machine contains memory trunk i, we maintain an "addressing table" with 2^p slots, where each slot stores a *mid* with which we can reach the corresponding Trinity server. Furthermore, in order for the global addressing to work, each machine keeps a replica of the addressing table. We will describe how we ensure the consistency of these addressing tables in Section 5.

We then locate the key-value pair in the memory trunk i. Each memory trunk is associated with a latch-free hash table on the machine whose *mid* is in the slot i of the addressing table. We hash the 64-bit key again to find the offset and size of the stored blob (the value part of the key-value pair) in the hash table. Given the memory offset and the size, we now can retrieve the key-value pair from the memory trunk.

The addressing table provides a mechanism that allows machines to dynamically join and leave the memory cloud. When a machine fails, we reload the memory trunks it owns from the TFS to other alive machines. All we need to do is to update the addressing table so that the corresponding slots point to the machines that host the data now. Similarly, when new machines join the memory cloud, we relocate some memory trunks to those new machines and update the addressing table accordingly.

Each key-value pair in the memory cloud may attach some metadata for a variety of purposes. Most notably, we associate each key-value pair with a spin lock. Spin locks are used for concurrency control and physical

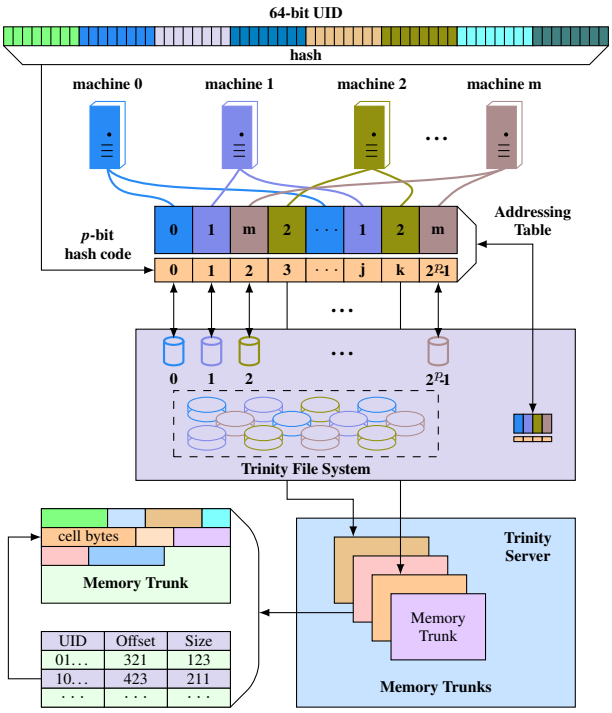

Figure 3: Data Partitioning and Addressing

memory pinning. Multiple threads may try to access the same key-value pair concurrently; we must ensure a key-value pair is locked and pinned to a fixed memory position before allowing any thread to manipulate it. In Trinity, all threads need to acquire the corresponding spin lock before it can access a key-value pair exclusively.

4 Trinity Specification Language

In this section, we introduce Trinity Specification Language (TSL). It is a declarative language designed to specify data schemas and message passing protocols using *cell* and *protocol* constructs. The TSL compiler is essentially a code generator: it generates optimized data access methods and message passing code according to the specificed TSL script.

4.1 Strongly-typed Data Modeling

Trinity supports property graphs[2] on top of its key-value store. "Keys" are globally unique identifiers introduced in Section 3 and their "values" are used for storing application data. The schema of the value part of a key-value pair can be specified by a *cell* structure in a TSL script. A *cell* structure in a TSL script specifies a user-defined data type. Defining a *cell* is pretty much like defining a struct in C/C++ as shown in Figure 4. The *value* part of such a key-value pair stored in Trinity memory cloud is called a *data cell* or simply *cell* when there is no ambiguity. Correspondingly, the *key* of the key-value pair is called its *cell Id*.

The TSL snippet shown in Figure 4 demonstrates how to model a graph node using a *cell* structure. A graph node represented by a *cell* and a *cell* can be referenced by its 64-bit *cell Id*, thus simple graph edges which reference a list of graph nodes can be represented by *List<int64>*. The data schema of graph edges that have associated data can be specified using a TSL *struct*. In the example shown in Figure 4, the schema of *MyEdges* is specified by the *MyEdge* struct.

[2]The nodes and edges of a property graph can have rich information associated.

```
struct MyEdge
{
    int64 Link;
    float Weight;
}
[GraphNode]
cell MyGraphNode
{
    string Name;
    [GraphEdge: Inlinks]
    List<int64> SimpleEdges;
    [GraphEdge: Outlinks]
    List<MyEdge> MyEdges;
}
```

Figure 4: Modeling a Graph Node

To distinguish the cell fields that specify graph edges from those that do not, we can annotate a cell and its data fields using TSL *attributes*. An attribute is a tag associated with a construct in TSL. Attributes provide the metadata about the construct and can be accessed during run time. An attribute can be a string or a pair of strings. An attribute is always regarded as a key-value pair. A single-string attribute is regarded as a key-value pair with an empty value. In the example shown in Figure 4, we use attribute *GraphNode* to indicate the cell *MyGraphNode* is a graph node and use attribute *GraphEdge* to indidate *SimpleEdges* and *MyEdges* are graph edges.

4.2 Modeling Computation Protocols

Trinity realizes a communication architecture called active messages [11] to support fine-grained one-sided communication. This communication architecture is desirable for data-driven computations and especially suitable for online graph query processing, which is sensitive to network latencies. TSL provides an intuitive way of writing such message passing programs.

```
struct MyMessage
{
    string Text;
}
protocol Echo
{
    Type: Syn;
    Request: MyMessage;
    Response: MyMessage;
}
```

Figure 5: Modeling Message Passing

Fig. 5 shows an example. It specifies a simple "Echo" protocol: A client sends a message to a server, and the server simply sends the message back. The "Echo" protocol specifies its communication type is synchronous message passing, and the type of the messages to be sent and received is *MyMessage*. For this TSL script, the TSL compiler will generate an empty message handler *EchoHandler* and the user can implement the message handling logic for the handler. Calling a protocol defined in the TSL is like calling an ordinary local method. Trinity takes care of message dispatching, packing, etc., for the user.

4.3 Zero-copy Cell Manipulation

Trinity memory cloud provides a key-value pair store, where the values are binary blobs whose data schemas are specified via TSL. Alternatively, we can store graph nodes and edges as the runtime objects of an object-oriented programming language. Unfortunately, this is not a feasible approach for the following three reasons. First, we cannot reference these runtime objects across machine boundaries. Second, runtime objects incur significant storage overhead. For example, an empty runtime object (one that does not contain any data at all) in .Net Framwork requires 24 bytes of memory on a 64-bit system and 12 bytes of memory on a 32-bit system. For a billion-node graph, this is a big overhead. Third, although Trinity is an in-memory system, we do need to store memory trunks on the disk or over a network for persistence. For runtime objects, we need serialization and deserialization operations, which are costly.

Storing objects as blobs of bytes seems to be desirable since they are compact and economical with zero serialization and deserialization overhead. We can also make the objects globally addressable by giving them unique identifiers and using hash functions to map the objects to memory in a machine. However, blobs are not user-friendly. We no longer have object-oriented data manipulation interfaces; we need to know the exact memory layout before we can manipulate the data stored in the blob (using pointers, address offsets, and casting to access data elements in the blob). This makes programming difficult and error-prone[3].

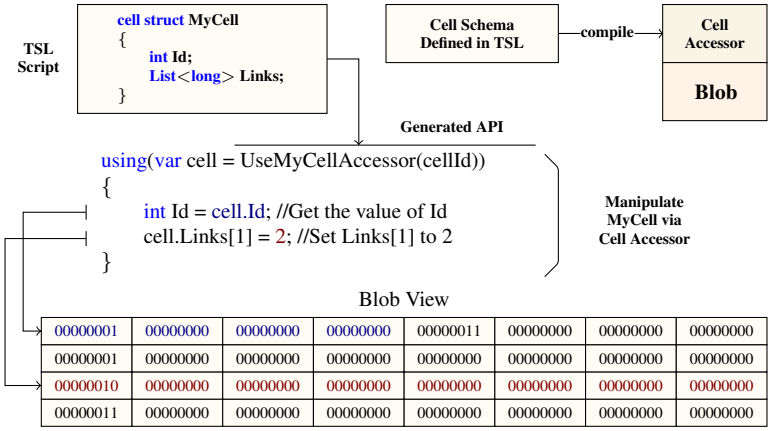

Figure 6: Cell Accessor

To address this problem, Trinity introduces a mechanism called *cell accessor* to support object-oriented data manipulation on blob data. Users first declare the schema of a cell in TSL, then the TSL compiler automatically generates data access methods for manipulating cells stored as blobs in the memory cloud. One of the generated function is *UseMyCellAccessor*. Given a cellId, it returns an object of type *MyCellAccessor*. With the generated *MyCellAccessor*, users can manipulate its underlying blob data in an object-oriented manner as shown in Figure 6.

As a matter of fact, a cell accessor is not a data container, but a data mapper. It maps the data fields declared in TSL to the correct memory locations in the blob. Data access operations to a data field will be correctly mapped to the correct memory locations with zero memory copy overhead. In addition, using the spin lock associated with each key-value pair, Trinity guarantees the atomicity of the operations on a single data cell when the cell is manipulated via its *cell accessor*. However, Trinity does not provide built-in ACID transaction support. This means Trinity cannot guarantee serializability for concurrent threads. For applications that need transaction support, users can implement light-weight atomic operation primitives that span multiple cells, such as MultiOp primitives [12] or Mini-transaction primitives [13] on top of the atomic cell operations provided by

[3]Note that we cannot naively cast a blob to a structure defined in programming languages such as C or C++ because the data elements of a struct are not always flatly laid out in the memory. We cannot cast a flat memory region to a structured data pointer.

cell accessor.

5 Fault Tolerance

As a distributed in-memory system, Trinity needs to deal with subtle fault-tolerance issues. The fault-tolerance requirements are highly application dependent, we discuss the general fault tolerance approaches in this section and application developers should choose proper fault tolerance approaches according to their application needs.

5.1 Shared Addressing Table Maintenance

Trinity uses a shared addressing table to locate key-value pairs, as elaborated in Section 3. The addressing table is a shared data structure. A centralized implementation is unfeasible because of the performance bottleneck and the risk of single points of failure. A straightforward approach to these issues is to duplicate this table on each server. However, this leads to the potential problem of data inconsistency.

Trinity maintains a primary replica of the shared addressing table on a leader machine and uses the fault-tolerant Trinity File System to keep a persistent copy of the primary addressing table. An update to the primary table must be applied to the persistent replica before being committed.

Both ordinary messages and heartbeat messages are used to detect machine failures. For example, if machine A attempts to access a data item on machine B that is down, machine A can detect the failure of machine B. In this case, machine A informs the leader machine of the failure of machine B. Afterwards, machine A waits for the addressing table to be updated and attempts to access the item again once the addressing table is updated. In addition, machine-to-machine heartbeat messages are sent periodically to detect network or machine failures.

Upon confirmation of a machine failure, the leader machine starts a recovery process. During recovery, the leader reloads the data owned by the failed machine to other alive machines, updates the primary addressing table, and broadcasts it. Even if some machines cannot receive the broadcast message due to a temporary network problem, the protocol still works since a machine always syncs up with the primary addressing table replica when it fails to load a data item. If the leader machine fails, a new round of leader election will be triggered. The new leader sets a flag on the shared distributed file system to avoid multiple leaders in the case that the cluster machines are partitioned into disjointed sets due to network failures.

5.2 Fault Recovery

We provide different fault recovery mechanisms for different computation paradigms. Checkpoint and "periodical interruption" techniques are adopted for offline computations, while an asynchronous buffered logging mechanism is designed for online query processing.

5.2.1 Offline Computations

For BSP-based synchronous computations, we can make checkpoints every few super-steps [5]. These checkpoints are written to the Trinity File System for future recovery.

Because checkpoints cannot be easily created when the system is running in the asynchronous mode, the fault recovery is subtler than that of its synchronous counterpart. Instead of adopting a complex checkpoint technique such as the one described in [14], we use a simple "periodical interruption" mechanism to create snapshots. Specifically, we issue interruption signals periodically. Upon receiving an interruption signal, all servers pause after finishing their current job. After issuing the interruption signal, Safra's termination detection algorithm [15] is employed to check whether the system has ceased. A snapshot is then written to the Trinity File System once the system ceases.

5.2.2 Online Query Processing

We now discuss how to guarantee that any user query that has been successfully submitted to the system will be eventually processed, despite machine failures.

The fault recovery of online queries consists of three steps:

1. Log submitted queries.
2. Log all generated cell operations during the query processing,
3. Redo queries by replaying logged cell operations.

Query Logging We must log a user query for future recovery after it is submitted to the system. To reduce the logging latency, a buffered logging mechanism [16] is adopted. Specifically, when a user query is issued, the Trinity client submits it to at least two servers/proxies. The client is acknowledged only after the query replicas are logged in the remote memory buffers. Meanwhile, we use a background thread to asynchronously flush the in-memory logs to the Trinity File System. Once a query is submitted, we choose one of the Trinity servers/proxies that have logged the query to be the *agent* of this query. The query *agent* is responsible for issuing the query to the Trinity cluster on behalf of the client. The agent assigns a globally unique *query id* to each query once the query is successfully submitted.

For the applications that only have read-only queries, we just need to reload the data from the Trinity File System and redo the failed queries when a machine fails. For the applications that support online updates, a carefully designed logging mechanism is needed.

Operation Logging A query during its execution generates a number of messages that will be passed in the cluster. An update query transforms a set of data cells from their current states to a set of new states through the pre-defined message handlers. All message handlers are required to be deterministic[4] so that we can recover the system state from failures by replaying the logged operations. The resulting states of the data cells are determined by the initial cell states and the generated messages.

We designed an asynchronous buffered logging mechanism for handling queries with update operations. We keep track of all generated cell operations and asynchronously log operations to remote memory buffers. Each log entry is represented by a tuple $< qid, cid, m, sn >$, which consists of a query id qid , a cell id cid, a message m, and a sequential number sn. The query id, cell id, and the message that has trigged the current cell operation uniquely specifies a cell operation. The messages are logged to distinguish the cell operations generated by the same query. The sequential number indicates how many cell operations have been applied to the current cell. It represents the cell state in which the cell operation was generated.

we enforce a sequential execution order for each cell using the spin lock associated with the cell. We can safely determine a cell's current sequential number when it holds the spin lock. We send a log entry to at least two servers/proxies once it acquires the lock. We call asynchronous logging *weak logging* and synchronous logging *strong logging*. When a cell operation is acknowledged by all remote loggers, its state becomes *strongly logged*. We allow *weak logging* if all the cell operations on the current cell with sequential numbers less than $sn - \alpha$ ($\alpha \geq 1$) have been strongly logged, where *sn* is the sequential number of the current cell and α is a configurable parameter. Otherwise, we have to wait until the operations with sequential numbers less than $sn - \alpha$ have been strongly logged. There are two implications: 1) The strongly logged operations are guaranteed to be consecutive. This guarantees all strongly logged operations can be replayed in the future recovery. 2) Introducing a *weak logging* window of size α reduces the chance of blocking. Ideally, if the network round-trip latency is less than the execution time of α cell operations, then all cell operations can be executed without waiting for logging acknowledgements.

[4]Given a cell state, the resulting cell state of executing a message handler is deterministic.

System Recovery During system recovery, all servers enter a "frozen" state, in which message processing is suspended and all received messages are queued. In the "frozen" state, we reload the data from the Trinity File System and start to redo the logged queries. In this process, we can process these queries concurrently and replay the cell operations of different cells in parallel.

For any cell, the logged cell operations must be replayed in an order determined by their sequential numbers. For example, consider the following log snippet:

$$\cdots < q_1, c_1, m_{10}, 13 >, \cdots, < q_1, c_1, m_{11}, 15 > \cdots$$

For query q_1, after the log entry $< q_1, c_1, m_{10}, 13 >$ is replayed, the entry $< q_1, c_1, m_{11}, 15 >$ will be blocked until the sequential number c_1 is increased to 14 by another query.

Let us examine why we can restore each cell to the state before failures occur. Because all message handlers are deterministic, for a query, its output and the resulting cell states solely depend on its execution context, i.e., the cell states in which all its cell operations are executed. Thus, we can recover the system by replaying the logged cell operations on each cell in their execution order.

Since the system recovery needs to redo all the logged queries, we must keep the log size small to make the recovery process fast. We achieve this by incrementally applying the logged queries to the persistent data snapshot on the Trinity File System in the background.

6 Real-life Applications

Trinity is a Microsoft open source project on GitHub[5]. It has been used in many real-life applications, including knowledge bases [17], knowledge graphs [18], academic graphs[6], social networks [19], distributed subgraph matching [20], calculating shortest paths [21], and partitioning billion-node graphs [22]. More technical details and experimental evaluation results can be found in [23], [20], [18], [21], [22], [24], and [25].

In this section, we use a representative real-life application of Trinity as a case to study how to serve real-time queries for big graphs with complex schemas. The graph used in this case study is Microsoft Knowledge Graph[7] (MKG). MKG is a massive entity network; it consists of 2.4+ billion entities, 8.0+ billion entity properties, and 17.4+ billion relations between entities. Inside Microsoft, we have a cluster of Trinity servers serving graph queries such as path finding and subgraph matching in real time.

Designing a system to serve MKG faces a new challenge of large complex data schemas besides the general challenges of parallel large graph processing discussed in Section 1. Compared to typical social networks that tend to have a small number of entity types such as *person* and *post*, the real-world knowledge graph MKG has 1610 entity types and 5987 types of relationships between entities[8]. Figure 7 shows a small portion (about 1/120) of the MKG schema graph.

The large complex schemas of MKG makes it a challenging task to efficiently model and serve the knowledge graph. Thanks to the flexible design of Trinity Specification Language, Trinity has met the challenge in an 'unusual' but very effective way. With the powerful code generation capability of the TSL compiler, we can *beat the big schema with big code*! For MKG, the TSL compiler generated about 18.7 million lines of code for modeling the knowledge graph entities in an extremely fine-grained manner. With the generated fine-grained strongly-typed data access methods, Trinity provides very efficient random data access support for the graph query processor.

[5]https://github.com/Microsoft/GraphEngine

[6]https://azure.microsoft.com/en-us/services/cognitive-services/academic-knowledge/

[7]The knowledge graph is also known as Satori knowledge graph.

[8]The size of MKG is ever growing; this number is only for an MKG snapshot.

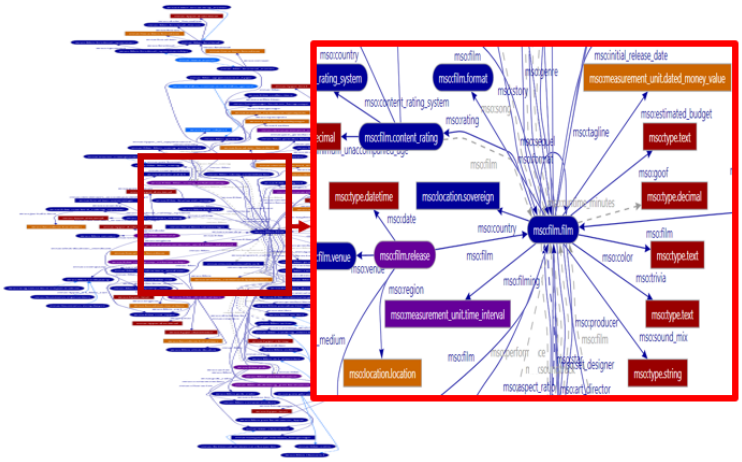

Figure 7: A small portion of the MKG schema graph

7 Conclusion

In this paper, we presented Trinity – a graph engine on a distributed in-memory infrastructure called Trinity Memory Cloud. Instead of being optimized for certain types of graph computations on certain types of graphs, Trinity is designed as a versatile "graph engine" to support a large variety of graph applications. Trinity now is a Microsoft open source project on GitHub and we have been using Trinity to support all kinds of graph applications including knowledge graphs, academic graphs, and social networks.

References

[1] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient rdf store over a relational database," in *SIGMOD*, 2013.

[2] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," ser. ICDM '09. IEEE Computer Society, 2009, pp. 229–238.

[3] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[4] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *OSDI*, 2012, pp. 31–46.

[5] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," ser. SIGMOD '10. ACM.

[6] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, August 2004.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," ser. SOSP '09. ACM, pp. 1–14.

[8] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 92–105, 2010.

[9] J. Zhang and B. Shao, "Trinity file system specification," Microsoft Research, Tech. Rep., 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/?id=201523

[10] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, 2007.

[11] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th annual international symposium on Computer architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 256–266.

[12] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," ser. PODC '07, 2007, pp. 398–407.

[13] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," ser. SOSP '07, 2007, pp. 159–174.

[14] H. Higaki, K. Shima, T. Tachikawa, and M. Takizawa, "Checkpoint and rollback in asynchronous distributed systems," ser. INFOCOM '97. IEEE Computer Society, 1997.

[15] E. W. Dijkstra, "Shmuel Safra's version of termination detection," Jan. 1987. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF

[16] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," ser. SOSP '11. ACM, 2011, pp. 29–41.

[17] W. Wu, H. Li, H. Wang, and K. Q. Zhu, "Probase: a probabilistic taxonomy for text understanding," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 481–492.

[18] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," in *Proceedings of the 39th international conference on Very Large Data Bases*, ser. PVLDB'13. VLDB Endowment, 2013, pp. 265–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=2488329.2488333

[19] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 277–288.

[20] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, May 2012. [Online]. Available: http://dx.doi.org/10.14778/2311906.2311907

[21] Z. Qi, Y. Xiao, B. Shao, and H. Wang, "Toward a distance oracle for billion-node graphs," *Proc. VLDB Endow.*, vol. 7, no. 1, pp. 61–72, Sep. 2013.

[22] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, Eds. IEEE, 2014, pp. 568–579.

[23] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516.

[24] L. He, B. Shao, Y. Li, and E. Chen, "Distributed real-time knowledge graph serving," in *2015 International Conference on Big Data and Smart Computing, BIGCOMP 2015, Jeju, South Korea, February 9-11, 2015*. IEEE, 2015, pp. 262–265.

[25] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang, "G-sql: Fast query processing via graph exploration," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 900–911, Aug. 2016.