Trinity: A Distributed Graph Engine on a Memory Cloud

Bin Shao
Microsoft Research Asia
Beijing, China
binshao@microsoft.com

Haixun Wang
Microsoft Research Asia
Beijing, China
haixunw@microsoft.com

Yatao Li*
HKUST
Kowloon, Hong Kong
ylibg@ust.hk

ABSTRACT

Computations performed by graph algorithms are data driven, and require a high degree of random data access. Despite the great progresses made in disk technology, it still cannot provide the level of efficient random access required by graph computation. On the other hand, memory-based approaches usually do not scale due to the capacity limit of single machines. In this paper, we introduce Trinity, a general purpose graph engine over a distributed memory cloud. Through optimized memory management and network communication, Trinity supports fast graph exploration as well as efficient parallel computing. In particular, Trinity leverages graph access patterns in both online and offline computation to optimize memory and communication for best performance. These enable Trinity to support efficient online query processing and offline analytics on large graphs with just a few commodity machines. Furthermore, Trinity provides a high level specification language called TSL for users to declare data schema and communication protocols, which brings great ease-of-use for general purpose graph management and computing. Our experiments show Trinity's performance in both low latency graph queries as well as high throughput graph analytics on web-scale, billion-node graphs.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Databases*; D.4.2 [**Operating Systems**]: Storage Management—*Distributed memories*

Keywords

Distributed System; Memory Cloud; Graph Database

1. INTRODUCTION

Large graphs appear in a wide range of computational domains, and we are facing challenges at all levels ranging from the infrastructure to the programming model for managing and processing large graphs. Graph applications have varied needs. We may roughly classify such needs into two categories: online query processing, which requires low latency, and offline graph analytics, which requires high throughput. As an example, deciding instantly whether there is a path between two given persons in a social network belongs to the first category while computing PageRank for the WWW belongs to the second. Still, many sophisticated applications have both needs: Given two nodes in a graph, the "distance oracle" algorithm that estimates the shortest distance between the two nodes is an online algorithm. However, to estimate the distances, the algorithm relies on "landmark" nodes in the graph, and the optimal set of landmark nodes are discovered using offline analytics.

Despite the diversity of graph applications, graph computation has some unique characteristics [26]: It usually has a high data-access-to-computation ratio, in other words, it is I/O intensive. Furthermore, graph computation usually requires a high degree of random data access. This is particularly true for online queries as they usually require certain degree of graph exploration (e.g., BFS, sub-graph matching, etc.). Offline graph computations are usually performed in an iterative, batch manner. For iterative computations, keeping data in main memory can improve performance by an order of magnitude due to the reuse of intermediate results as illustrated by Spark [34]. Moreover, the scale of data makes distributed parallel computation the most promising solution for large graph processing. As we shall see, keeping the graph, at least the topology, in distributed memory not only improves the performance, but also enables a new set of graph computation paradigms.

In this paper, we introduce Trinity, a distributed graph engine on a memory cloud. Trinity supports both online graph query processing and offline graph analytics, and it has been used for real life applications, including knowledgebases [33], knowledge graphs [36], and social networks. Trinity is able to scale-out, which means it can host arbitrarily large graphs in the memory of a cluster of commodity machines[1]. Instead of optimizing for certain types of graph computation (e.g., BSP), Trinity directly addresses the random data access problem in large graph computation. Trinity implements a globally addressable distributed memory storage, and provides a random access abstraction for large

*The work was done at Microsoft Research Asia.

[1]Trinity usually makes the graph topology and frequently used information of the graph memory-resident. Trinity provides transparent access to other information associated with the graph in DBMSs.

graph computation. The design of Trinity is based on the belief that, as high-speed network becomes more available and DRAM prices trends downward in the long run, all-in-memory solutions provide the lowest total cost of ownership for a large range of applications [8]. For instance, RAMCloud [30] envisioned that advances in hardware and OS technology will eventually enable all-in-memory applications, and low latency can be achieved by deploying faster NICs and network switches and by tuning the OS, the NIC, and the communication protocol. Trinity realizes this vision for large graph applications, and Trinity does not rely on hardware/platform upgrades and/or special OS tuning, although Trinity can leverage these techniques to achieve even better performance.

	Graph Database	Query Processing	Graph Analytics	Scale-out System
Neo4j [4]	Yes	Yes	Yes	No
HyperGraphDB [22]	Yes	Yes	No	No
GraphChi [25]	No	No	Yes	No
PEGASUS [23]	No	No	Yes	Yes
MapReduce [15]	No	No	Yes	Yes
Pregel [28]	No	No	Yes	Yes
GraphLab [1]	No	No	Yes	Yes

Table 1: Some Representative Graph Systems

Before we discuss the details of Trinity, we survey a few representative graph systems that have been proposed in the last few years. Table 1 summarizes our survey results. Among the existing graph systems, Neo4j [4] and HyperGraphDB [22] focus on supporting online transaction processing (OLTP) on graph data. However, they are not distributed: They do not handle graphs that are partitioned over multiple machines. This limits the size of the graphs they can *efficiently* handle. Furthermore, a single machine also does not have enough computation power compared with a distributed, parallel system. Thus, it is difficult for such systems to handle web-scale graphs.

On the other end of the spectrum are MapReduce [15], Pregel [28], GraphLab [1]. These are high latency, high throughput offline platforms. Unlike Neo4j and HyperGraphDB, they do not support online query processing, instead, they are optimized for analytics on large data partitioned over hundreds of machines. MapReduce computations on graphs depend heavily on interprocessor bandwidth, as graph structures are sent over the network iteration after iteration. Pregel and GraphLab mitigate this problem by passing computation results instead of graph structures between processors. In Pregel, GraphLab, and GraphChi, analytics on the graphs are expressed using a vertex centric computation paradigm. Although some well known graph algorithms, including PageRank, shortest path discovery, can be implemented through vertex centric computing with ease, there are a large range of sophisticated graph computations, for example, multi-level graph partitioning, that cannot be expressed in a succinct and elegant way.

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 enables the development of such modules and hence empowers a large variety of graph applications. In other words, it can easily morph into systems to support any specific graph applications.

The rest of the paper is organized as follows. In Section 2, we outline the design of the Trinity system. In Section 3, we introduce the Trinity's memory cloud. Section 4 describes the Trinity data model. Section 5 analyzes the computation paradigms of typical graph applications. Section 6 discusses some technical details in system implementation. Section 7 presents experimental results on Trinity. We discuss related work in Section 8 and conclude in Section 9.

2. AN OVERVIEW OF TRINITY

We show the architecture of Trinity in Figure 1. Trinity is a storage infrastructure and computation framework built on top of a cluster of well-connected machines. As a storage infrastructure, Trinity organizes the memory of multiple machines into a globally addressable, distributed memory address space (a memory cloud) to support large graphs. Trinity is designed for online query processing applications as well as offline analytics applications, and it supports user-friendly graph modeling, object-oriented data manipulation.

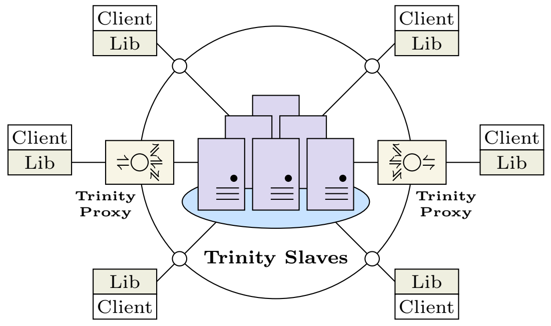


Figure 1: Trinity Cluster Structure

A Trinity system consists of multiple components that communicate through a network. According to the roles they play, we classify them into three types: slaves, proxies, and clients. A Trinity slave plays two roles: storing graph data and performing computation on the data. Computation usually involves sending messages to and receiving messages from other Trinity components. Specifically, each slave stores a portion of the data and processes messages received from other slaves, proxies, or clients. A Trinity proxy only handles messages but does not own any data. It usually serves as a middle tier between slaves and clients. For example, a proxy may serve as an information aggregator: It dispatches requests from clients to slaves and sends results back to the clients after aggregating partial results received from slaves. Proxies are optional, that is, a Trinity system does not always need a proxy. A Trinity client is responsible for enabling users to interacting with the Trinity cluster. It is a user interface tier between the Trinity system and end-users. Trinity clients are applications that are linked to Trinity library. They communicate with Trinity slaves and Trinity proxies through the APIs provided by the Trinity library.

Figure 2 shows the stack of Trinity system modules. The memory cloud is essentially a distributed key-value store, and it is supported by a memory storage module and a message passing framework. The memory storage module manages memory and provides mechanisms for concurrency control. The network communication module provides an efficient, one-sided, machine-to-machine message passing infrastructure.

Trinity Provides a specification language called TSL (Trinity specification language) that bridges the graph model and

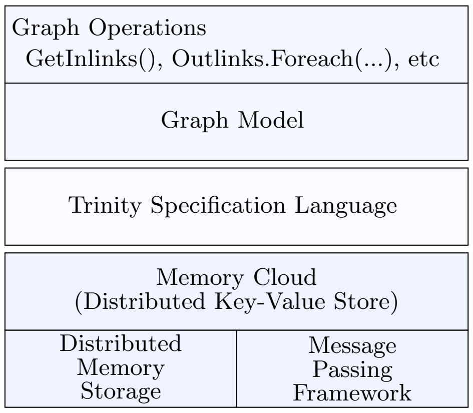

Graph Operations
GetInlinks(), Outlinks.Foreach(...), etc
Graph Model
Trinity Specification Language
Memory Cloud
(Distributed Key-Value Store)
Distributed Memory Storage
Message Passing Framework

Figure 2: System Layers

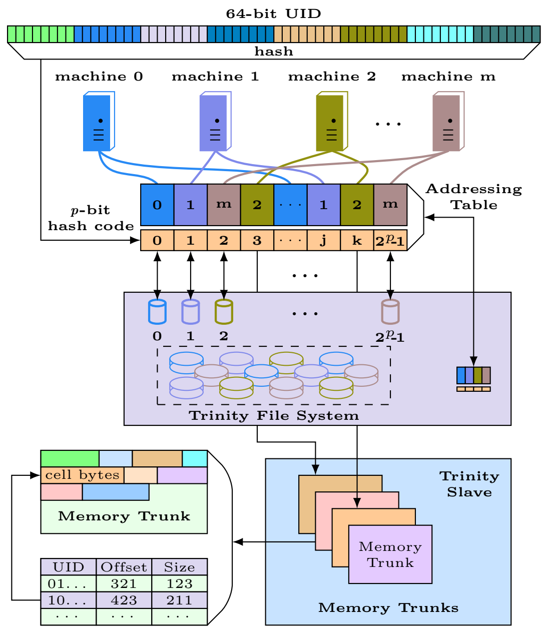


Figure 3: Data Partitioning and Addressing

the data storage. Due to the diversity of graphs and the diversity of graph applications, it is hard, if not entirely impossible, to support efficient general purpose graph computation using fixed graph schema. Instead of using fixed graph schema and fixed computation models, Trinity let users define graph schema, communication protocols, and computation paradigms through TSL.

3. THE MEMORY CLOUD

We create a distributed memory cloud as Trinity's storage infrastructure. The memory cloud consists of 2^p memory trunks, each of which is stored on a machine. Usually, we have $2^p > m$, where m is the number of machines. In other words, each machine hosts multiple memory trunks. The reason we partition a machine's local memory space into multiple memory trunks is twofold: 1) Trunk level parallelism can be achieved without any overhead of locking; 2) The performance of a single huge hash table is suboptimal due to a higher probability of hashing conflicts. Essentially, the entire memory cloud is partitioned into 2^p memory trunks. To support fault-tolerant data persistence, these memory trunks are also backed up in a shared distributed file system called TFS (Trinity File System), which is similar to HDFS [10].

On top of the memory cloud, we create a key-value store. A key-value pair forms the most basic data structure of the graph system or any system built on top of the memory cloud. Here, keys are 64-bit globally unique identifiers, and values are blobs of arbitrary length. As 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, and then 2) locate the key-value pair in one of the memory trunks on that machine. Through this hashing mechanism (shown in 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 $i \in [0, 2^p - 1]$. This means the key-value pair is stored in memory trunk i within the memory cloud. To find out which machine memory trunk i is in, we maintain an "addressing table" that contains 2^p slots, where each slot stores a machine ID. Essentially, we implement a consistent hashing mechanism that allows machines to join and leave the memory cloud (described later). Furthermore, in order for global addressing to work, each machine keeps a replica of the addressing table, and we will describe how we ensure the consistency of the addressing tables in Section 6.2.

We then locate the key-value pair in memory trunk i, which is stored on the machine whose ID is in slot i of the addressing table. Each memory trunk is associated with a hash table. We hash the 64-bit key again to find the offset and size of the key-value pair in the hash table. Given the memory offset and the size, we 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 contain some meta data for various purposes. Most notably, we may associate each key-value pair with a spin lock. Spin locks are used for concurrency control and physical memory pinning. Multiple threads may try to access the same key-value pair concurrently. A physical key-value pair may also be moved by the memory defragmentation thread as elaborated in Section 6.1. Therefore, we must ensure a key-value pair is locked and pinned to a fixed memory position before allowing any thread to manipulate it. For applications that are not read-only, the spin lock mechanism allows a thread to access a pinned physical key-value pair exclusively by requiring all threads to acquire the lock before accessing or moving a cell.

4. DATA MODEL

Trinity is designed to handle graph data of diverse characteristics. In this section, we describe data modeling issues and the Trinity Specification Language that is designed to facilitate the modeling.

4.1 Modeling Graph

A graph consists of nodes and edges. But a graph is more than its topology: In real-world applications, nodes and edges in a graph are often associated with rich information. We may use relational databases, XML, or even plain text files to store graphs. But they do not support efficient access of graph data. Take relational databases as an example. We may use one table to store the nodes, and another table to store the edges. If edges represent more than one relationship, we may need multiple tables to store edges of difficult types. This seems to be simple and intuitive. However, graph operations usually involve graph traversal, which incurs costly, multi-way joins of relational tables. Thus, relational databases are not for processing graph data.

Trinity supports graphs on top of an in-memory key-value store. Here, the "key" is a system wide identifier, and the "value" is used for modeling application data. When the "value" is associated with a schema (defined in TSL, which is described in Section 4.2), we also call it a *cell*, and the (key, value) pair becomes a (cellId, cell) pair. To model graphs on top of a key-value store, we use a cell to implement a node in a graph. A cell may contain a lot of information. For undirected graphs, a cell (a graph node) contains a set of cellIds that represent its neighboring nodes. For directed graphs, a cell (a graph node) contains two set of cellIds, one for incoming links, and the other for outgoing links. A cell may contain additional data associated with a node, such as the name of the node, its description, etc.

Usually, there is no need to represent an edge as a cell. As mentioned above, we may represent a node's outgoing edges by the cellIds of the nodes they connect to. Additional data associated with an edge (e.g., its name, type, weight, etc.) can simply stay with the cellId as (cellId, associatedData) pairs. However, when edges are associated with rich information, we may represent edges using cells, and store the rich information associated with the edges in the edge cells. Correspondingly, a node will store a set of edge cellIds. We can also model hypergraphs in this way, as we can easily store a set of node cellIds in an edge cell.

4.2 Trinity Specification Language

We designed a high level language called TSL (Trinity Specification Language) for data and network communication modeling in Trinity. As we know, graphs have very diverse characteristics, and distributed algorithms on graphs have very diverse communication patterns. In face of these diversity, TSL brings great ease-of-use for general purpose graph management and computing. Its goals and benefits include the following:

- TSL provides object-oriented data manipulation for the underlying blob data in the memory cloud. This will be elaborated in Section 4.3.

- TSL facilitates data integration. It defines an interface between graphs and external data (e.g., data in an RDBMS). Through TSL, we can specify how nodes in a graph are associated with records in a relational table. This enables us to store graph topology and some critical data in Trinity's memory cloud, while leaving other rich information (such as images) on disk. This further enables transparent query processing over memory cloud and RDBMSs (but with dramatically improved performance as join relationships are materialized in Trinity), and automatic data conversion between memory cloud and external data sources.

- TSL facilitates system extension. With data schema and communication protocols defined in TSL, the TSL compiler generates highly efficient and powerful source code for data manipulation and communication, which greatly facilitates the development of advanced system modules in Trinity. For example, we implemented a sophisticated graph query language (TQL) within this framework.

Figure 4 gives an example of using TSL to model the data of a toy graph consisting of movies and actors.

```
[CellType: NodeCell]
cell struct Movie
{
    string Name;
    [EdgeType: SimpleEdge, ReferencedCell: Actor]
    List<long> Actors;
}
[CellType: NodeCell]
cell struct Actor
{
    string Name;
    [EdgeType: SimpleEdge, ReferencedCell: Movie]
    List<long> Movies;
}
```

Figure 4: Modeling a Movie and Actor Graph

The script above defines two types of graph nodes, namely Movie and Actor, using two `Cell structs`. A `Cell struct` is a basic element for modeling graph. It is a data container which may contain an arbitrary number of: 1) primitive data types, such as byte, int, and double; 2) data container types, such as Array, List, and BitArray; and 3) other user-defined structs. In our case, the Movie and Actor cells contain data elements `List<long> Actors` and `List<long> Movies`, which are outgoing edges from the cells. The "[...]" constructs (following C# convention) in the script describes the constructs that follow them. For example, it indicates Actors are `SimpleEdge` from Movie cells to Actor cells. Besides `SimpleEdge` (which is represented by a cellId), Trinity also supports `StructEdge` (which is an independent cell) and `HyperEdge` (hyperedges).

Besides modeling data, TSL also models network communication. This is important for the following reasons. First, graph algorithms have very diverse network communication patterns because they are data driven, and the data is distributed. It is extremely tedious for users to implement all kinds of message passing protocols (e.g., synchronous, asynchronous, etc.) Second, in vertex based computing and other algorithms, a large number of nodes send and receive messages simultaneously. The total number of messages in the system is huge although each message may be small. This incurs a huge cost if the system does not automatically pack small messages between two machines into a single transfer. Third, graph algorithms require a flexible message passing mechanism. The well known message passing framework MPI has drawbacks for distributed graph applications: It is optimized for two-sided bulk synchronous communication.

A lot of tuning is needed to write efficient, asynchronous, fine-grained message passing programs, and writing such code is tedious.

TSL provides an intuitive way of writing efficient message passing programs for graph computation. It provides one-sided communication based on the request-response communication paradigm, and it supports bulk synchronous message passing and transparent message packing for asynchronous messages to increase the network throughput.

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

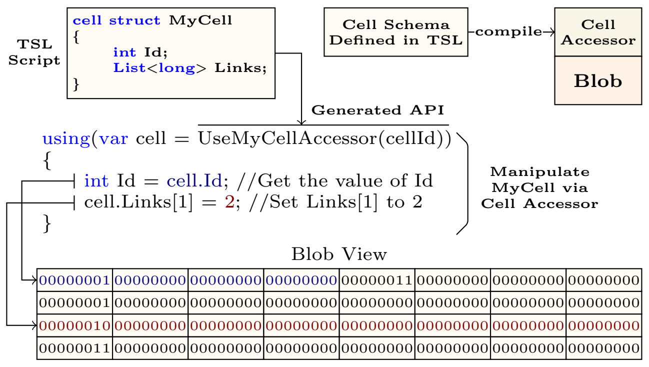
Figure 5: Modeling Message Passing

Figure 5 shows an example. We implement a simple "Echo" protocol: A client sends a message to a server, and the server sends a message back. It is stated that "Echo" uses synchronous message passing, and the type of messages being sent and received is *MyMessage*. TSL compiles the script to generate an empty message handler *EchoHandler*, and the user only needs to implement the algorithm logic for the handler as if implementing a local method. Calling a protocol defined in the TSL is also like calling a local method. Trinity takes care of message dispatching, packing, etc., for the user.

4.3 Object-Oriented Cell Manipulation

The memory cloud provides a key-value pair store, where values are binary blobs. The TSL script in Figure 4 informs the Trinity system the schema of the data, so that Trinity knows how to manipulate the data, including, for example, integrating the data with data from external sources. Alternatively, we can implement graph nodes and edges as runtime objects. Unfortunately, we cannot reference objects across machine boundaries. Second, runtime objects incur significant storage overhead. For C# on the .Net framework, an empty runtime object (one that does not contain any data element) requires 24 bytes of memory on a 64-bit system and 12 bytes of memory on a 32-bit system. For billion-node graphs, this is a tremendous overhead. Third, although Trinity is an in-memory system, we do need to store memory trunks on disk or network for persistence. For runtime objects, we need serialization and deserialization operations, which is costly.

On the other hand, storing objects as blobs of bytes is compact, 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 host machine as we have described. However, blobs are not user-friendly. We no long have an object-oriented interface, and we need to know the exact memory layout before we can manipulate the data in the blob (using pointers, address offsets, and casting to access data elements in the blob). This makes programming difficult and error-prone[2].



```
cell struct MyCell
{
    int Id;
    List<long> Links;
}
```

```
using(var cell = UseMyCellAccessor(cellId))
{
    int Id = cell.Id; //Get the value of Id
    cell.Links[1] = 2; //Set Links[1] to 2
}
```

00000001	00000000	00000000	00000000	00000011	00000000	00000000	00000000
00000001	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000010	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000011	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 6: Cell Accessor

To address this problem, Trinity introduces a mechanism (the *cell accessor* mechanism) to support object-oriented data manipulation on blob data. Users first declare the schema of a cell in TSL, then Trinity automatically generates key-value store interfaces for manipulating cells stored as blob strings in the memory cloud. Specifically, Trinity compiles the TSL script to create a set of APIs for accessing, loading, and saving the data. One of the generated API is `UseMyCellAccessor`. Given a cellId, it returns an object of type `MyCellAccessor`, and users can manipulate its underlying blob data as a runtime object in an object-oriented manner. This is shown in Figure 6. As a matter of fact, a cell accessor is not a data container, but a data mapper. It maps the fields declared in the data structure to the correct memory locations in the blob. Any data accessing operation to a data field will be correctly mapped to the correct memory location with zero memory copy overhead.

4.4 Consistency

Using the spin lock associated with each key-value pair, Trinity guarantees the atomicity of the operation on a single cell (key-value pair). However, Trinity does not provide ACID transaction support. This means Trinity cannot guarantee serializability for concurrent threads. For applications that need transaction support, we can implement light-weight atomic operation primitives that span multiple cells, such as MultiOp primitives [13] and Mini-transaction primitives [7], on top of the atomic cell operation primitives.

5. GRAPH COMPUTATION PARADIGMS

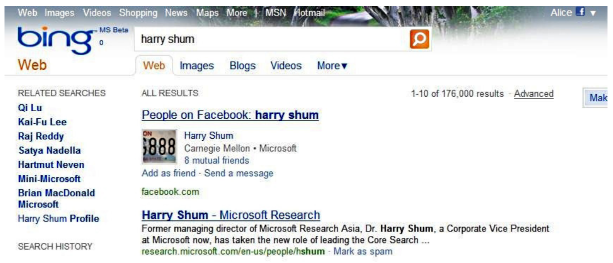
Different graph computations have different data access and communication patterns. In this section, we discuss the graph computation paradigms supported by Trinity.

5.1 Traversal Based Online Queries

A lot of applications require graph exploration, the breadth-first search and the depth-first search being the most typical. Here, we use "people search" on a social network as an example to demonstrate the importance of efficient graph exploration to online query processing. The problem is the following: On a social network, for a given user, find anyone whose first name is "David" among his/her friends, his/her friends' friends, and his/her friends' friends' friends. This is

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

a practical problem: While logged in on Facebook, a user performs a search in Bing. Bing explores the user's Facebook network to see if there is anything relevant. In the case as shown in Figure 7, it finds someone who is the user's friends' friend.


Web Images Videos Shopping News Maps More | MSN Hotmail
Alice
bing MS Beta
Web
Web Images Blogs Videos More▾
RELATED SEARCHES
ALL RESULTS
Mak
SEARCH HISTORY

harry shum

1-10 of 176,000 results · Advanced

Qi Lu
Kai-Fu Lee
Raj Reddy
Satya Nadella
Hartmut Neven
Mini-Microsoft
Brian MacDonald Microsoft
Harry Shum Profile

People on Facebook: **harry shum**
Harry Shum
Carnegie Mellon • Microsoft
8 mutual friends
Add as friend · Send a message
facebook.com

Harry Shum - Microsoft Research
Former managing director of Microsoft Research Asia, Dr. **Harry Shum**, a Corporate Vice President at Microsoft now, has taken the new role of leading the Core Search ...
research.microsoft.com/en-us/people/h**shum** · Mark as spam

Figure 7: Facebook search in Bing

It is unlikely we can index the social network to solve the "David" problem. One option is to index the neighborhood for each user, so that given any user, we can use the index to check if there is any "David" within 3 hops of his/her neighborhood. However, the size and the update cost of such an index are prohibitive for a web-scale graph. The second option is to create an index to answer 3-hop reachability queries for any two nodes. This is infeasible because "David" is a popular first name, and we cannot check every David in the social network to see if he is within 3 hops to the current user.

Trinity solves the "David" problem by leveraging its very efficient memory-based graph exploration capabilities. We deployed a synthetic, power-law graph in an eight-machine cluster managed by Trinity. The graph has Facebook-like size and distribution (800 million nodes, 104 billion edges, with each node having on average 130 edges). We found that exploring the entire 3-hop neighborhood of any node in the graph takes less than 100 milliseconds on average. In other words, Trinity is able to explore $130 + 130^2 + 130^3 \approx 2.2$ million nodes distributed over eight machines in one tenth of a second. The algorithm simply sends asynchronous requests recursively to remote machines, and the performance is achieved by efficient memory access and optimization of network communication.

5.2 A New Paradigm for Online Queries

Trinity introduces a new paradigm for online graph processing by storing web-scale graphs in the memory of a distributed system, and relying on fast random access and parallel computing for query processing, as demonstrated by the above example.

In contrast, instead of storing graph in its native form, many existing graph systems store graph data in relational tables, or matrices, and use *join operations* to simulate graph exploration. The approach does not scale. In order to support more sophisticated online queries such as subgraph matching, existing systems rely on index. The reality is, none of the existing systems and methods support efficient subgraph matching on web-scale graphs. To understand the challenge, consider various kinds of indices developed to support query processing on graphs. Most of them require super-linear space and/or super-linear construction time. For example, the R-Join approach [14] for subgraph matching is based on the 2-hop index [11]. The complexity to build such an index is $O(n^4)$, where n is the number of vertices. It is obvious that in large graphs where the value of n is on the scale of 1 billion (10^9), any super-linear approach will become unrealistic, let alone an algorithm of complexity $O(n^4)$.

In Trinity, the combination of fast random access and parallel computing, offers a new paradigm which enables us to rethink efficient query processing on web-scale graphs. Figure 8(a) shows the performance of subgraph matching on web scale graphs. Here, the size of the graph ranges from 1 million to 128 million nodes, average node degree is 16, average query size is 10 (nodes), and queries are generated using two random methods, DFS and RANDOM [32]. It shows that without any index of graph structure, average query time is 1 second using just 8 machines for parallel query processing, and there is still a lot of room for improvement.

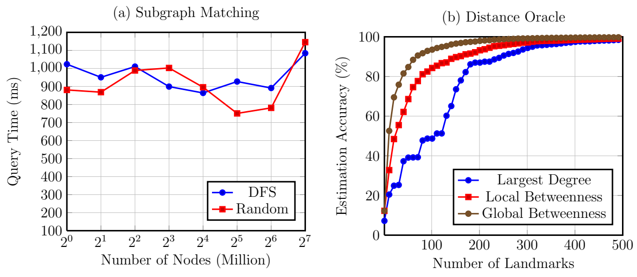


Figure 8: New paradigms of graph computation

5.3 Vertex Centric Offline Analytics

Trinity, as well as Pregel [28] and GraphChi [25], provide a vertex centric computation model for offline graph analytics. A computation task is expressed in multiple iterative super-steps and each vertex acts as an independent agent. During each super-step, each agent performs some computation and communication, independent of each other.

In the following, we first compare Pregel's vertex centric model with Trinity's vertex centric model. We will discuss GraphChi's computation model later at the end of this section.

- *A general model (Pregel).* In each super-step, a vertex may receive messages sent to it by *any* vertex in the previous super-step, send messages to *any* vertex, and modify its vertex values.
- *A restrictive model (Trinity).* In each super-step, a vertex may receive messages sent to it by *a fixed set* of vertices (usually its neighbors), send messages to *another fixed set* of vertices, and modify its vertex values.

As far as the above two computation models are concerned, Trinity is more restrictive than Pregel in the sense that in Trinity's computation model, a vertex only communicates with a subset of vertices (usually its neighbors). The reason Trinity focuses on the restrictive model is twofold. First, in a lot of well-known vertex centric computation, e.g. PageRank and shortest path, vertices only talk with their neighbors. In other words, the restrictive model can already support a large number of graph algorithms on its own. Second, the restrictive model introduces a lot of optimization opportunities. Since the communication is with a fixed set of vertices, the communication pattern is predictable iteration after iteration. This enables us to optimize messages delivery to maximize the performance. With the optimization techniques, Trinity achieves much better performance. With just 8 machines, one BSP iteration on a synthetic, power-law graph of 1 billion nodes and 13 billion edges takes less than 60 seconds. This enables Trinity to support algorithms such as PageRank and shortest distances very efficiently using vertex centric computation.

Before we describe the detail of how Trinity optimizes message passing for supporting the restrictive computation model in Section 5.4, we describe some related work. Recently, GraphChi [25] introduces a computation model which is even more restrictive. It supports asynchronous computation only, which means it cannot implement traversal based graph computation and synchronous graph computation efficiently. The benefit of doing so is that for analytics that can be implemented in an asynchronous manner, GraphChi can accomplish them using sequential disk access. It is clear that there is a compromise between the expressive power of a computation model and its efficiency.

However, unlike Pregel and GraphChi that are specialized systems focusing on one computation model, Trinity is not constrained by any computation model. For instance, Trinity can partition billion-node graphs within a few hours using a multi-level partitioning algorithm [6]. The quality of the partitioning is comparable to that of the best partitioning algorithm (e.g., METIS [24]). To the best of our knowledge, billion-node graph partitioning is an unsolved problem on general-purpose graph platforms. In summary, Trinity can adopt any computation model (BSP as in Pregel, restrictive vertex centric computation, or asynchronous model as in GraphChi), yet because Trinity deploys graphs on a memory cloud, it can express any algorithm using graph exploration, and is not constrained by any computation model.

5.4 Message Passing Optimization

In this section, we describe how Trinity optimizes message passing to support its vertex based computation model. Although a graph is distributed on multiple machines, from the point view of a local machine, vertices of the graph are in two categories: vertices on the local machine, and vertices on any of the remote machines. Figure 9(a) shows a local machine's bipartite view of the entire graph.

In vertex based computation, each vertex is scheduled to run a job. Before the job on a vertex can start running, the vertex must receive all the messages it needs. One naive approach is to wait until all messages to arrive before we start running the job on any of the local vertices. This means the local machine needs to buffer all the messages from remote vertices, and perform random access to retrieve the messages when running algorithms on the local vertices. Since the total amount of messages is too big to be memory resident, we need to buffer the messages on disk, and perform random accesses on the disk. This incurs significant cost.

Another naive approach is to run jobs on local vertices without preparing any messages in advance. When a local vertex is scheduled to run the job, we obtain remote messages for the vertex, and run the job immediately after they arrive. Since the system does not have space to hold all messages, we discard messages after they are used. For example, in Figure 9(a), in order to run the job on vertex x, we need messages from vertices u, v, and others. Later on, when y is scheduled to run, we need messages from u and v again. This means a single message needed to be delivered multiple times, which is unacceptable in an environment where network capacity is an extremely valuable resource.

In Trinity, we assume each vertex mainly communicates with a fixed set of vertices such as its neighbors (the restrictive vertex computation model). This provides opportunity of optimization, because at least we can prepare most of the messages a vertex needs before it is scheduled to run. To do this, we create a bipartite partition of the local graph as shown in Figure 9(b). In the ideal case, local vertices in a partition only needs messages from remote vertices in the same partition. Thus, as long as the local machine can buffer the messages required for each partition in memory, we can run the job on local vertices in the partition without waiting, and we also ensure that each message is delivered just once.

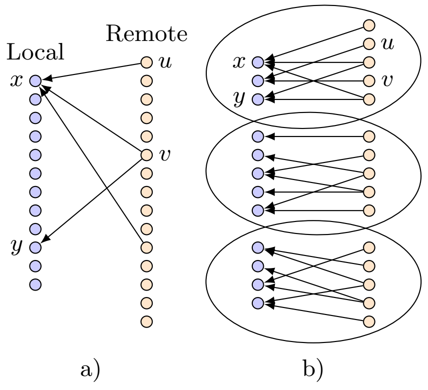


Figure 9: Bipartite View on a Local Machine

However, graph partitioning itself is a very costly task, and it is very difficult to create partitions of even size while minimizing the number of edge cuts. In Trinity, on each local machine, we differentiate remote vertices into 2 categories. The first category contains hub vertices, that is, vertices having a large degree and connecting to a great percentage of local vertices. The second category contains all of the remaining vertices. We buffer messages from vertices in the first category for the entire duration of one iteration. In other words, they will not participate in the partitioning process shown in Figure 9(b). For a scale-free graph, e.g., one generated by degree distribution $P(k) \sim ck^{-\gamma}$ with $c = 1.16$ and $\gamma = 2.16$, 20% hub vertices are sending messages to 80% of vertices. Even if we buffer messages from just 10% hub vertices, we have addressed 72.8% of message needs. Thus, after exclusing these frequently used messages, the partitioning becomes much easier.

In reality, vertices may need messages from vertices in other partitions, or from vertices other than a predefined fixed set of vertices such as their neighbors (the general vertex computaton model), these messages are obtained on demand. Let K_i be the remote vertices that do not belong to partition i but whose messages are needed by local vertices in partition i. We obtain messages from K_i while we are running algorithms on local vertices in partition $i - 1$. Usually, the size of K_i is small, and we can obtain messages from K_i before running the algorithms on local vertices in partition i.

The scheme described above requires remote messages to arrive in certain order (that is, partition by partition, including messages on cut edges K_i). To achieve this, after a local machine performs the partitioning, it sends out to each remote machine an "action script", which specifies the order of the messages it needs. Each machine merges the action scripts it receives from other machines. After the execution starts, each machine sends messages based on the merged script, iteration after iteration.

One observation we can make from Figure 9 is that unlike in online query processing, where it is impossible to predict which part of the graph the next query is going to access, the data access pattern in offline analytics jobs can be predicted. For example, for vertex centric computation, the execution is partition by partition. On each machine, the same sequence of execution will repeat over and over again.

Since we can predict the data access pattern, we no longer have to keep the entire graph memory resident. At any moment, we consider two types of vertices: A) vertices in a partition currently scheduled to run on a certain machine; and B) all of the other vertices in the graph. For a Type A vertex, we keep its cell structure in memory, the computation may need to access its neighbors, attributes, and local variables. For a Type B vertex, we only keep its message box in memory, since Type A vertices may need it. This is illustrated in Figure 10.

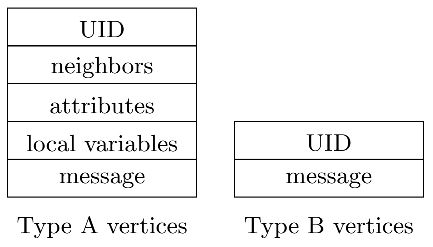


Figure 10: Memory Resident Cell Structures

This arrangement dramatically reduces the memory requirement. Originally, to keep the graph topology memory resident (also include some runtime information), the total memory we need is:

$$S = |V| \cdot (16 + k + l + m) + 8|E|$$

where 16 (bytes) is the space needed to store and access the UID, k, l, and m are the average sizes of attributes, local variables, and message, respectively. In the offline mode, the memory we need is

$$S' = p \cdot S + (1 - p) \cdot |V| \cdot (16 + m)$$

where p is the percentage of Type A vertices. Thus, the saved memory is:

$$S - S' = (1 - p)(k + l) \cdot |V| + (1 - p) \cdot 8|E|$$

Suppose $k = l = m = 8$ and $p = 0.1$, for the Facebook social graph, 708 GB memory space can be saved. This means the number of required machines can be reduced significantly without affecting the performance much.

5.5 A New Paradigm for Offline Analytics

Besides the capabilities mentioned above, Trinity also introduces a new paradigm for offline graph analytics. In Trinity, a web-scale graph is partitioned and stored on a number of machines. This leads to the following question: Can we perform graph computation locally on *each machine* and then aggregate their answers to derive the answer for the entire graph, or can we use probabilistic inference to derive the answer for the entire graph from the answer on *a single machine*? This paradigm has the potential to overcome the network communication bottleneck, as it minimizes or even abolishes network communication. The answer to the above question is positive. If a graph is partitioned over 10 machines, each machine has full information about 10% of the vertices and 10% of the edges. Besides, the edges link to a large amount of the remaining 90% of vertices. Thus, the sample actually contains a lot of information about the entire graph. Furthermore, when a graph is randomly partitioned, each machine holds a random sample of the graph, which enables us to perform probabilistic inference.

We experimented with this new paradigm in Trinity. In one example, we find landmark vertices, and use them to estimate shortest distances between any two nodes in a large graph [37]. In Figure 8(b), we show the effectiveness of using 3 methods to pick the landmark vertices. Here, the X axis shows the number of landmark vertices we use, and the Y axis shows estimation accuracy. The best approach is to use vertices that have the highest global betweenness, and the worst approach is to simply use vertices that have the largest degree. Our approach, which uses vertices that have the highest betweenness computed locally, actually has very close accuracy to the best approach. However, finding landmarks that have the highest global betweenness is significantly more costly than our approach.

6. IMPLEMENTATION DETAILS

We implemented Trinity on Microsoft .Net framework. In this section, we discuss some technical details in memory management and fault tolerance.

6.1 Circular Memory Management

Trinity memory cloud is a key-value store, and the value in a key-value pair is a blob of arbitrary size. Some (graph) applications need to manage billions of key-value pairs. Among these key-value pairs, some are very large (imagine a very popular celebrity on Facebook with a million fans), but most of them are small. It is a challenge to manage billions of key-value pairs of varied sizes in a very compact manner. Traditional memory management mechanisms that use memory buckets are extremely wasteful because they leave memory gaps between billions of key-value pairs. Furthermore, when we update a key-value pair, the size of the blob may increase or decrease, creating further challenges for memory management. For Trinity, the goal is to support fast memory allocation, efficient memory reallocation, and maintains a high memory utilization ratio.

Trinity implements a circular memory management mechanism, which tries to avoid memory gaps between large number of key-value pairs. When we initialize a memory trunk, we reserve a 2GB virtual memory address space from the operating system. As shown in Figure 11 (a), new key-value pairs are sequentially appended to the *committed memory* region. The pointer *append head* points to the next free memory byte. The memory space occupied by existing key-value pairs resides in the region between *committed tail* and *append head*.

When we expand a key-value pair or create a new key-value pair, we always allocate a new memory region from *append head* if there is enough *committed memory*. If no space is left, we allocate one or more new memory pages and advance *committed head*. Hence, memory allocation is efficient, because in most cases, we just advance the *append head*. Shrinking or expanding the size of a key-value pair may leave memory gaps in the *committed memory* region. Trinity activates a defragmentation daemon which scans the *committed memory* region, and moves key-value pairs toward *append head*. After a pass, the freed memory pages at the tail of *committed memory* is released and the *committed tail* is moved forward. In this process, the *committed head*, *append head*, and *committed tail* may move to the start of the memory trunk, performing an endless circular movement until all available memory is occupied.

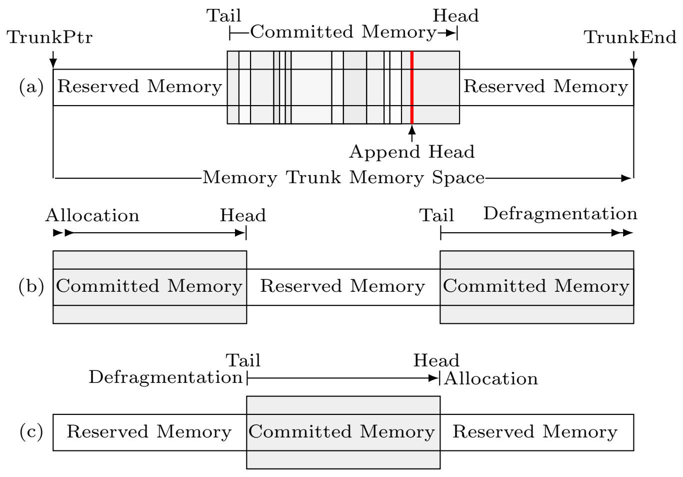


Figure 11: Circular Memory Management

For certain applications (e.g., graph generation, graph streams, etc.), the size of key-value pairs keep increasing (as new edges are added to the node cells). This creates tremendous overhead in committing and decommitting memory pages, and in moving key-value pairs around for defragmentation. To reduce the overhead, we devised a short-lived memory reservation mechanism to support frequent key-value pair reallocation. A naive idea is to reserve a few more bytes during a key-value pair expansion for future expansion. For example, if the current key-value pair needs to expand by 16 bytes, we allocate 32 bytes instead. The problem for this naive solution is that the memory utilization ratio will be lowered since a lot of reserved memory is actually not used. To address this issue, we make all memory reservation short-lived. One reservation is only valid in the interval between two defragmentation passes. The unused reserved memory will be released in next defragmentation. On the one hand, by memory reservation, we greatly decrease the chance of memory committing/decommitting and memory movement so that we can greatly improve the key-value pair expansion performance. On the other hand, through memory defragmentation, unused reserved memory is collected timely so that we can minimize the runtime memory overhead.

6.2 Fault Tolerance

As a distributed system, Trinity needs to deal with various fault-tolerance issues.

Shared Addressing Table Maintenance

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

Trinity maintains a primary replica of the shared addressing table on a leader machine, and uses the fault-tolerant distributed file system TFS to keep a persistent copy of the primary addressing table. An update to the primary table must be applied to the persistent replica before committing.

Trinity uses heartbeat messages to proactively detect machine failures. Besides this, a machine A that attempts to access a data item on machine B which is down can detect the failure of machine B. In this case, machine A will inform the leader machine of the failure of machine B. After that, machine A will wait for the addressing table to be updated, and attempt to access the item again once the addressing table is updated.

On the confirmation of a machine failure, the leader machine will start the recovery process. During recovery, the leader reloads data owned by the failed machine to other alive machines, updates the primary addressing table and broadcasts it. Even if some slave machines cannot receive the broadcast message due to temporary network interruption, the protocol still works since a machine will always sync 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 marks a flag on the shared distributed fault-tolerant file system to avoid multiple leaders in the case that the cluster machines are partitioned into disjointed sets due to network failure.

Fault Recovery

For different computation models, we use different fault recovery mechanisms. For BSP based synchronous computation, we make check points every a few supersteps. These check points are written to the persistent file system for future failure recovery. For asynchronous computation, the fault recovery issue is subtler than that of its synchronous counterpart, as check points cannot be easily created when the computation is running in the asynchronous mode. Instead of adopting a complex checkpoint techniques, e.g. [21], we use a simple "periodical interruption" mechanisms to create snapshots. Specifically, Trinity issues an interruption signal periodically. On receiving this signal, all vertices will pause after finishing the job in hand. After issuing the interruption signal, Trinity calls Safra's termination detection algorithm [16] to check whether the system ceases. A snapshot is written to the persistent disk storage once the system ceases. For read-only queries, we just restart the failed node and reload the data from the persistent disk storage. For online update queries, we use the buffered logging mechanism proposed in RAMCloud to do failure recovery. The key idea is to log operations to remote memory buffers before committing them to the local memory. Detailed discussion about buffered logging can be found in [30, 29].

7. EXPERIMENTAL EVALUATION

We conduct a number of experiments to measure the performance of Trinity for online queries and offline analytics. All the following experiments are performed in a 16-machine cluster. Each machine has 96 GB DDR3 RAM and two 2.67 GHz Intel(R) Xeon(R) X5650 CPUs, each processor has 12 threads. The operating system is 64-bit Windows Server

2008 R2 Enterprise. There are dual network adapters on each machine, one is 40 Gbps Mellanox IPoIB Adapter and the other is 1 Gbps HP NC382i DP Multifunction Gigabit Server Adapter. Trinity is implemented using C# and complied with target platform x64. The Trinity runtime is .NET Framework 4.

People Search Query On Social Graph. We perform people search queries in a social network to measure the performance of data-intensive, traversal-based online graph queries. This experiment measures query response time of searching friends by name within 2 and 3 hops on synthetic social graphs. The out-degree of each node varies from 10 to 200. Eight machines are used in this experiment, the performance curves are shown in Figure 12(a). The response times of 2-hop queries are always under 10 ms. The response time of 3-hop search on the graph with 130 node degree is 96.2 ms in this experiment. Currently the average degree of Facebook is 130. This indicates people search queries on Facebook like social graphs can be answered within 100 ms.

Page Rank Calculation On Web Graph. Calculation of page rank is one of the commonest graph analytics tasks on web graphs. Here we use page rank calculation to measure the offline processing performance of Trinity. In this experiment, the page rank is calculated using synchronous vertex centric computation model on R-MAT graphs [12]. The number of vertices varies from 64 million to 1024 million. The average degree is 13. The computation time for one iteration (a super-step in BSP model) on 8, 10, 12 and 14 machines is shown in Figure 12(b). We can see that one page rank iteration on a graph with 1 billion node can be completed in merely one minute on 8 machines.

Breadth-first Search. Breadth-first search (BFS) is a fundamental graph computation operation. Many graph algorithms are built on BFS. Graph 500 Benchmark [5] adopts BFS as one of its two computation kernels. Figure 12(c) shows the performance curves of performing BFS on 8, 10, 12 and 14 machines. This experiment uses the same data as that in the page rank calculation experiment. For the 1 billion node graph, it takes 1028 seconds on 8 machines, while 644 seconds on 14 machines.

Trinity vs. PBGL. PBGL [19] is a generic C++ library for high-performance parallel and distributed graph Computation. We run BFS on RMAT graphs in a 16-machine cluster using PBGL and Trinity to compare their execution time and memory usage. The node count varies from 1 million to 256 million with average degree 4, 8, 16 and 32. The results in Figure 13 clearly show that Trinity runs 10x faster with 10x less memory footprint. When average degree is 32, PBGL runs out of memory on the 256 million graph. PBGL uses the "ghost cells" mechanism for message passing. This mechanism incurs great memory overhead, especially on not-well-partitioned graphs. It takes near 600 GB memory for the 256 million node graph when average degree is 16. In contrast, Trinity takes less than 65 GB memory for the same graph.

Trinity vs. Giraph. Giraph [3] is a publicly available pregel implementation. We run page rank on Giraph to compare its performance with Trinity. We deployed the latest Giraph on the 16-machine cluster. The required Hadoop version is 0.20.2-RC1. The JRE version is jre-6u31-windows-x64, and the maximum memory heap size is set to 81 GB. The experimental results are shown in Figure 12 (d). Each page rank iteration takes 2455 seconds on the graph with 256 million nodes and 2 billion edges. With half number of machines, each page rank iteration of Trinity on a graph with 1 billion nodes and 13 billion edges only takes 51 seconds. Trinity runs faster by two orders of magnitude. Note: when average degree is 16, Giraph run out of memory on the 256 million node graph. This indicates that the runtime memory footprint of Giraph is much larger Trinity.

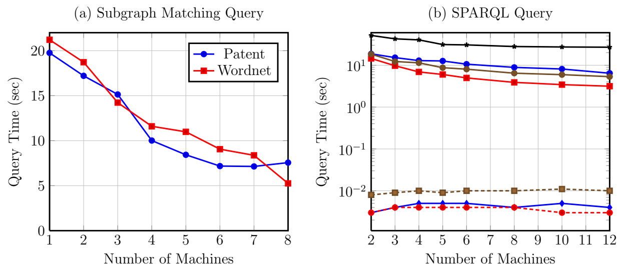


Figure 14: Parallel Speedup for Online Queries

Parallel Speedup. We have performed experiments to evaluate the distributed parallel speedup for both graph analytics jobs and online graph queries. The experimental results of page rank and BFS are shown in Figure 12 (b) and (c). The parallel speedup for online queries is shown in Figure 14. Figure 14 (a) shows the response time of subgraph match queries on two real-life graphs (Wordnet and US patent network), while (b) shows the response time of four SPARQL queries on a LUBM RDF data set [20] with 1,367,122,031 triples [36]. As the number of machines increases, the computation time is dramatically reduced for both offline graph analytics and online query processing. With more machines, the data partition on each machine is smaller, but the network message number will increase. Trinity is designed to scale out, with more machines, the performance would keep going up until the network communication limit is reached.

More Discussions on Scalability. An interesting observation is that Trinity has the least memory footprint when performing jobs on the same graph, compared with PBGBL and Giraph. PBGL and Giraph take more memory than trinity, while run much slower than Trinity. The reasons are twofold: i) Due to the random data access pattern, the graph data must be cached in memory during computation. However, in PBGL and Giraph, graph nodes exist as runtime objects in memory. They take much more memory than Trinity's plain blobs. 2) Trinity is an all-in-memory system, the graph data is loaded in memory before doing computation. So there is no cache-missing penalty. This mechanism is much more efficient than memory caching mechanism even a lot of memory is allocated as cache. Cache-misses hurt performance greatly even if there are only a few occurrences.

8. RELATED WORK

Many applications utilize large RAM storage to offer better performance. Large web applications, such as Facebook, Twitter, Youtube, and Wikipedia, heavily use memcached

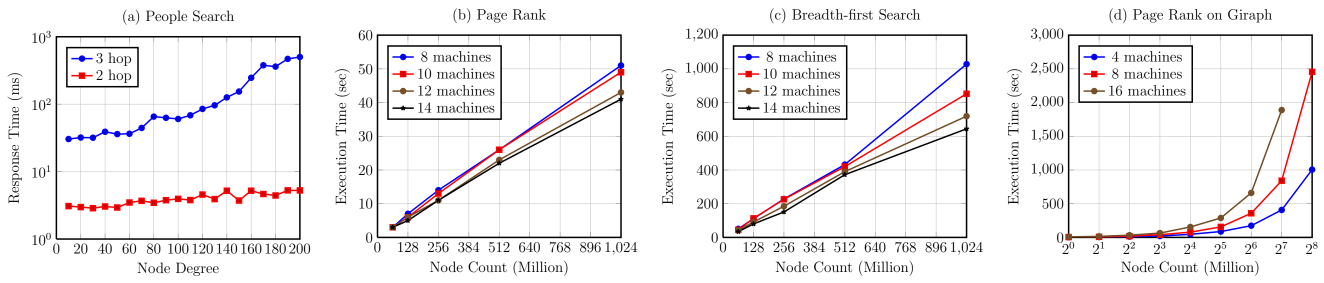


Figure 12: Trinity Performance Experiments

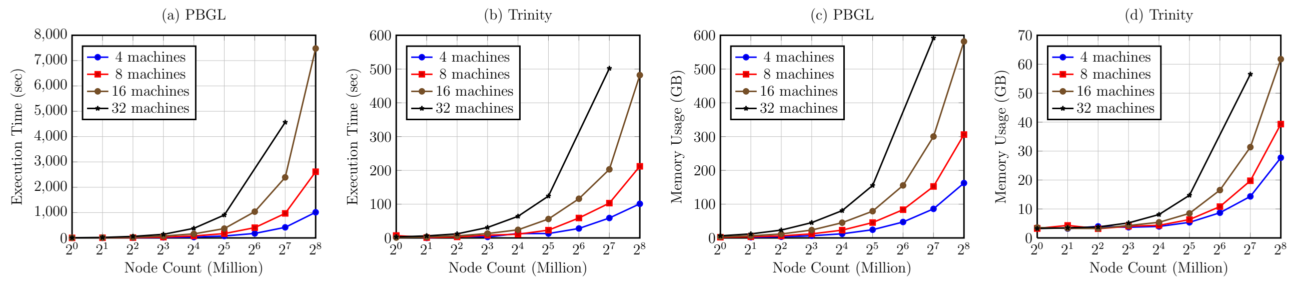


Figure 13: BFS in PBGL and Trinity

[17] to cache large volume 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. Machines in a caching system are independent with each other. They do not communicate with each other to perform computation, hence they do not solve the computation latency problem.

Besides memcached, key-value stores such as Scalaris [31] and Redis [35] also use RAM as their main storage. Recently, RAMCloud [30] pushes the idea of RAM based applications to a new level. RAMCloud is a data storage system that hosts all its data permanently in memory. The memory cloud storage of Trinity resembles the concept proposed in RAMCloud. Although both RAMCloud and Trinity have a large distributed RAM storage, their design goals are quite different. RAMCloud tries to leverage the low latency and high throughput data access power of RAM to support large scale applications, especially large web applications. While Trinity focuses on utilizing the random data access trait of distributed RAM storage to support large graph processing applications, especially those need fast graph exploration. To the best of our knowledge, Trinity is the first mature system that turns the concept of ram cloud into a large graph processing platform.

Many systems have been devised to process large graphs. PEGASUS [23] is a graph mining system built on Hadoop [2], which is an open source implementation of the MapReduce framework [15]. Due to the challenges posed by parallel graph processing [26], MapReduce based approaches are often "ill-suited" for large graph processing as discussed in [28]. To overcome the challenges, Pregel [28], GraphLab [1], PowerGraph [18], and GraphChi [25] adopt vertex centric graph computation models. The idea of vertex centric computation can be traced back to the well studied synchronous and asynchronous network model [27]. The processors in a processor network becomes graph vertices, and communication channels becomes graph edges. To a large extent, these metaphors in the context of graph computation are the key innovations of vertex centric graph computation.

Due to the random data access problem, general purpose graph computations usually do not have efficient disk-based solutions. But under certain constraints, offline graph problems that can be solved by "divide-and-conquer" strategy sometimes have efficient disk based solutions. A typical example is GraphChi [25]. GraphChi can perform efficient disk based graph computation under an assumption that current computation has an asynchronous vertex centric solution. Asynchronous solution means a vertex can perform computation just based on partially updated information from its incoming links. This assumption, on the one hand, frees the need of passing messages of current vertex to all its outgoing links so that it can perform the graph computation block by block. On the other hand, it inherently cannot support traversal based graph computation and synchronous graph computation efficiently because it cannot access the neighborhood of a vertex efficiently. Disk based graph computation solutions are essentially cache mechanisms. If a problem can be well partitioned, then the sub-problems can be loaded in memory and efficiently handled one by one. However, as widely acknowledged [26], most graph problems are usually inherently irregular and hard to partition especially for online queries.

Besides various graph processing systems, a number of parallel graph processing libraries are also available for implementing parallel graph algorithms on large graph data, such as Parallel Boost Graph Library (PBGL) [19] and Multi-Threaded Graph Library (MTGL) [9]. PBGL provides graph data structures and message passing mechanisms to make graph algorithms run in parallel. It uses *ghost cells* (local replicas of remote cells) for message passing. As discussed in [26], the *ghost cell* mechanism only works well for well-partitioned graphs. Great memory overhead would be incurred for not-well-partitioned large graphs. This fact is clearly validated by our experimental results shown in Section 7. A big difference between PBGL and Trinity is their communication infrastructure. PBGL's network communication relies on MPI, which heavily utilizes two-sided communication paradigm. Trinity uses the one-sided communication paradigm, a vertex can send message to other vertices without any prior appointment. This communication paradigm makes fine-grained parallelism possible on large graph data [26]. MTGL is designed for massively multi-threaded architectures. However MTGL only works on massively multi-threaded machines. Thus, the dependence on architecture makes it hard to be adopted widely.

9. CONCLUSION

We introduce a graph engine called Trinity for building online query processing applications as well as offline graph analytics applications. The graph engine is built on top of a distributed memory storage infrastructure called memory cloud. Memory cloud is designed to address the random data access challenge of large graph processing. With this storage infrastructure, Trinity enables a large variety of efficient graph computing paradigms for web scale graphs.

10. REFERENCES

[1] `http://graphlab.org/`.

[2] `http://hadoop.apache.org/`.

[3] `http://incubator.apache.org/giraph/`.

[4] `http://neo4j.org/`.

[5] `http://www.graph500.org/`.

[6] How to partition a billion-scale graph. Technical report, Microsoft Research, 2012.

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

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

[9] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IPDPS 2007*, pages 1–14, 2007.

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

[11] R. Bramandia, B. Choi, and W. K. Ng. Incremental maintenance of 2-hop labeling of large graphs. *TKDE*, 22(5):682–698, 2010.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. SDM '04, 2004.

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

[14] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI '04, pages 137–150.

[16] E. W. Dijkstra. Shmuel Safra's version of termination detection. Jan. 1987.

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

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[19] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. POOSC '05.

[20] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[21] H. Higaki, K. Shima, T. Tachikawa, and M. Takizawa. Checkpoint and rollback in asynchronous distributed systems. INFOCOM '97, pages 998–, 1997.

[22] B. Iordanov. Hypergraphdb: a generalized graph database. WAIM '10, pages 25–36, 2010.

[23] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. ICDM '09, pages 229–238, 2009.

[24] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Supercomputing '96.

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

[26] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[27] N. A. Lynch. *Distributed Algorithms*. 1996.

[28] 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. SIGMOD '10.

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

[30] 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.*, 43:92–105, 2010.

[31] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. ERLANG '08, pages 41–48, 2008.

[32] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

[33] W. Wu, H. Li, H. Wang, and K. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*, 2012.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. HotCloud'10, pages 10–10, 2010.

[35] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. Linux Magazine, 2009.

[36] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. In *VLDB 2013*.

[37] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao. Orion: shortest path estimation for large social graphs. WOSN'10, pages 9–9, 2010.