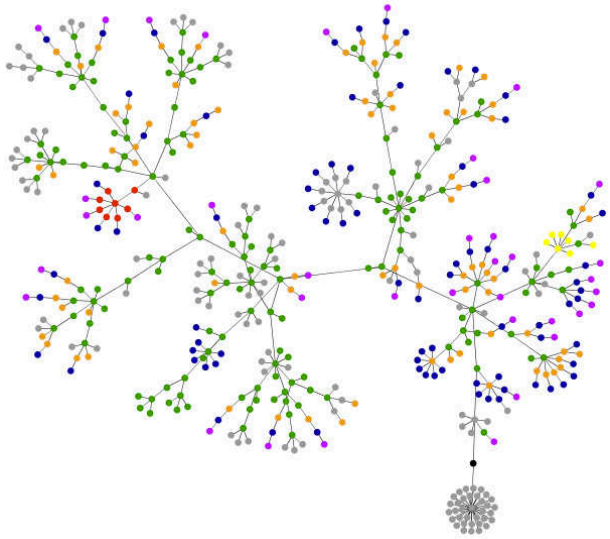
University of Zurich[UZH]

Bachelor Thesis August 16, 2011

Making Signal/Collect Scale



Daniel Strebel
of Zurich ZH, Switzerland

Student-ID: 07-908-072
daniel.strebel2@uzh.ch

Advisor: **Philip Stutz**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to use this chance to thank all the people who helped me realizing this thesis and supported my throughout the whole project. First of all I thank Professor Abraham Bernstein, Ph.D., for giving me the opportunity to develop this thesis at the Distributed Information Systems (DDIS) group and letting me participate in a highly interesting research project.

I am sincerely grateful to my supervisor Philip Stutz, Ph.D. student at the DDIS group and author of the Signal/Collect framework, for introducing me to this exciting topic and involving me in some of the strategic architectural changes of the framework. He supported me with helpful advice when I ran into problems and shared my excitement when they were solved. His dedication helped making the whole project an equally educational and fun experience.

Last but not least, I want to thank my family and friends, especially Jasna, for supporting me during all the ups and downs I had, while coding and writing these pages.

Daniel Strebel
Zurich, Switzerland

Abstract

The size of the indexable web, and other data collections structured as a graph, is growing at an exorbitant pace and undoubtedly exceeds the available memory resources of a single machine. This thesis presents a way that allows the Signal/Collect to process data sets that would not fit into main memory, by storing the elements of the graph on disk. It describes different back end solutions to hold the vertices and describes other measures that have to be taken in order to load large graphs on disk and execute algorithms on them. In the evaluation we show the effect of these optimizations and that on-disk storage allows processing a graph with one million vertices with only 500 MB of RAM. It is also shown that the on-disk version of a SSSP computation is considerably slower than a comparable distributed implementation and why computation times of an on-disk SSSP computation will not scale linearly with the graph size or with the number of worker threads.

Zusammenfassung

Die Grösse des indizeribaren Webs und anderen Datensätzen mit Graph-Struktur wächst mit schwindelerregender Geschwindigkeit und übersteigt zweifelsfrei die verfügbaren Arbeitsspeicher-Ressourcen einer einzelnen Maschine. Diese Arbeit präsentiert Wege, die dem Signal/Collect Framework erlauben Datensätze zu verarbeiten, welche aufgrund ihrer Grösse nicht im Arbeitsspeicher Platz finden würden, indem es die einzelnen Elemente des Graphs auf einer Festplatte ablegt. Es werden verschiedene Speicheroptionen und andere Optimierungen beschrieben, die notwendig sind um grosse Graphen zu laden und Berechnungen drauf durchzuführen. In der Evaluation kann gezeigt werden, dass mithilfe von Disk-basierter Speicherung ein SSSP Algorithmus auf einem Graph mit einer Million Knoten in nur 500 MB Arbeitsspeicher durchgeführt werden kann. Zudem wird gezeigt, dass die Disk-basierte Berechnung eines SSSP ein vielfaches mehr Zeit in Anspruch nimmt als eine vergleichbare Berechnung in einem verteilen System und warum SSSP-Berechnungen mit On-Disk Speicher weder linear mit der Anzahl Knoten im Graph noch mit der Anzahl paralleler Threads skaliert.

Table of Contents

1

Introduction

The amount of data available in the web but also for may other highly active research areas is growing at an exorbitant pace. The Signal/Collect framework [Stutz et al., 2010] enables parallelizing computations on data entities that can be expressed as a set of vertices in a graph and have edges connecting them to control the flow of the sub-steps of an algorithm. The problem currently faced by the framework is that the size of a graph that can be processed is limited by the amount of memory available. The framework stores all vertices and edges in memory of a single machine and can therefore only process graphs whose size fit into the available memory resources. To enable processing graphs of a larger size the framework needs a way to scale beyond the limitation imposed by the constrained resources. To provide the needed scalability two approaches would be possible. One solution to overcome the limitations would be to distribute the computation on many machines, while the other approach would include storing the data needed for computation on a secondary storage device and only retrieve it, if it was needed for computation. This thesis covers the latter concept and lays out an extension of the Signal/Collect to handle on-disk storage, while a distributed version of Signal/Collect is developed in parallel.

The thesis presents ways to handle graphs with a large number of vertices by serializing them and storing them on secondary storage devices. It discusses three different storage implementations and compares them among each other as well as with the standard in-memory approach. Other optimization factors, such as caching, serialization, in-memory compression or changes in the way signals are propagated to the vertices, are also exploited and evaluated. The goal of this analysis is to describe if and where it is a useful solution to extend Signal/Collect with on-disk storage capability and to present suggestions for what cases the different on-disk implementations promise a better performance.

1.1 Structure of the Thesis

The next chapter shows the problem faced by the in-memory implementation of Signal/Collect and explains the need for a storage implementation that consumes less main memory in order to enable storing more vertices in the graph. In addition, the requirements for the on-disk implementations are stated. Chapter 3 begins by describing the default in-memory storage structure and then introduces an interface that allows encapsulating different storage implementations. This is followed by a declaration of the different storage implementations themselves. Further optimizations, such as how to reduce the footprint of the vertices in memory as well as on disk or the measures taken to optimize the load phase of a graph, are also considered in this chapter. Chapter 4 shows how these different implementations perform compared to each other as well as in comparison with the in-memory data structures. The effectiveness of additional measures such as optimized serialization variants are demonstrated as well. In chapter 5 the on-disk approach is compared to the solutions proposed by other graph processing frameworks to overcome the limitations of scarce main memory. The last chapter gives an outlook on how these results can be integrated with Signal/Collect and whether on-disk vertex storage can allow handling very large data sets or could serve as a utility feature that provides a reliable fallback state, which could be necessary in a unreliable - maybe even distributed - environment.

2

Motivation

Calculations on graphs have been performed for a long time and their importance for practical problems has increased along the ongoing development of the World Wide Web and other prominent graph-networks. The fast growing amount of data for example in the World Wide Web, but also in many other areas, such as semantic web engineering or in biotech, require scalable solutions to be able to process their data sets. For performing computations on large real world data sets, a user is currently faced with the decision to either implement a custom processing engine that fits the specific needs of the intended application scenario or use a generic programming model that already implements a lot of the necessary lower-level functionality. As computational models for data that can be represented in a graph-like structure two concepts of computation have evolved. MapReduce [Dean and Ghemawat, 2008] and similar frameworks to distribute computational tasks have been proven to be successful in a number of cases, however mapping graphs, such as the representation of the web, to a key/value model so that they fit in the general programming model of MapReduce is unintuitive and adds unnecessary complexity. As a solution for this, a second category of computational frameworks has evolved that can exploits the relations of entities by performing computations on the data in a graph structure. One implementation of this model is the Signal/Collect framework [Stutz et al., 2010], which is a scalable computing model that can perform graph algorithms in synchronous or asynchronous fashion. Currently the framework scales almost linear to the number of available processor cores, but is limited to processing graphs that fit into the main memory of a single machine. This constraint makes it currently impossible to perform computations on large datasets such as the set of all static indexable webpages. Taken this set to assess the needed scalability, would yield a size of several billions vertices, because the size of this set was estimated to contain

around 11.5 billion pages in 2005 [Gulli and Signorini, 2005] and has grown even larger since then. For pure in-memory storage the framework is currently limited to process artificial web graphs of about two million pages. To overcome this limitation two approaches would be possible. The first way to handle data sets, which do not fit into main memory of a single machine, would be to store the graph, or parts of it, on a persistent storage device that has more and cheaper storage capacity available. Another solution would be, to distribute the computation on several machines and just increase the number of collaborating machines to match the requirements of the calculation to perform. This thesis covers the first approach and serializes the graph structure to disk in order to allow processing larger graphs. This avoids network latency and the costly management of an infrastructure with a number of computers.
The goal of the on-disk implementation is to provide a storage back end that is capable of storing a large number of vertices and allows retrieving them by some identifier. The storage should be optimized for frequent reads and updates and implement a generic storage interface that allows to easily replace different storage implementations. In addition to that serializing the vertices should work without requiring any additional work by the user, with the possibility to provide a more efficient serialization schema if desired. For tuning the performance of the on-disk storage, the access on the vertices needs to be optimized and some additional measures need to be introduced that do not only benefit on-disk storage but also improve the capabilities of the in-memory implementation. As an example, the special implementation of a vertex for a PageRank algorithm that is constructed to facilitate its serialization and deserialization has also a much simpler object structure that basically consists only of basic types. This vertex can also be used for in-memory execution to reduce the memory needed. This illustrates the last but most important requirement of the proposed storage implementations, which is that they have to extend the existing in-memory implementation without compromising its performance, so that in-memory storage is still performant without restrictions.

3

Design and Implementation

This section describes different storage implementations as well as the interface that is used to encapsulate them and hide the specifics of the storage implementations, while making the different steps of the storage process more transparent for the user. It specifies the overall architecture of the vertex storage backend and explains its main components. Five solutions for different storage back ends are proposed to store the vertices and edges of a Signal/Collect graph. The presented solutions contain in-memory, on-disk and cached approaches and are compared against each other in the next chapter. The chapter closes by presenting other changes and optimizations that were required to speed up the computation. Some of these adaptions are designed to improve on-disk storage, while others can be used to improve computation irrespective of the storage backend.

3.1 Interface Design

In order to facilitate different storage implementations that can arbitrarily be replaced a consistent storage interface needs to be introduced. The goal of this interface is to define all methods needed to store the required information for a Signal/Collect computation. The computing model of the framework requires it to provide a way for storing vertices and handling updates on them. In addition to that the storage interface has to declare methods that enable accessing the collections, which hold all the vertices that need to signal or collect at the current point of time. The proposed interface allows the user of the framework to replace the default storage backend with another storage implementation. The optional storage implementation can be one of the storage back ends provided by the framework or write an own implementation. This abstraction allows the user to

choose the storage backend that best fits the needs and resources of a specific application scenario.

3.1.1 Original In-Memory Storage Implementation

In the original implementation of the Signal/Collect framework the elements of a graph are distributed among a set of worker-objects. Each worker stores the vertices that are associated with it. Which worker is responsible for a specific vertex is currently determined as a function of the vertex id, but one could use other heuristics to better balance the load among the workers in the future. Together with each vertex all the edges are stored that have that vertex as their source. The edges are unidirectional connectors of the vertices and therefore they need to contain the id of another vertex to which the source vertex is connected. Along these edges the vertex can later send signals according to Signal/Collect's computational model. This computational model requires that each worker essentially has to maintain three collections of vertices. One collection containing the vertices themselves, with all the state information associated with them, and two utility collections that hold all the vertices that need to signal respectively to collect in a later phase.
Figure 3.1 illustrates the storage environment of at worker in its original design, i.e. when the framework was designed to only handle in-memory vertex storage. Before introducing the interface to encapsulate vertex access, the vertices were only accessed from within the worker class through a standard Java *HashMap*. A vertex stored its state as well as a collection of edges that linked it to other vertices. Optionally other static or dynamic information could be stored if required to perform the algorithm. In addition to that, each vertex referenced a message-bus that belonged to its worker and was used to send messages to other vertices or the coordinator. The edges stored on the vertices had a referenced their source vertex directly to gain access to its properties. This reference allowed the edges to directly access the state of its source in order to calculate the signal that is then propagated to the target vertex. Apart from storing the vertices the worker also held the *toSignal* and *toCollect* sets, which were stored as Java *HashSets*. These sets contained direct references to the vertices that needed to be handled. Since all vertices were simply held in-memory, storing the vertices that needed to signal or to collect by reference was very convenient for the computation process. The set of vertices that have to signal for example was able to invoke the signal function on each vertex directly without first having to request the most recent instance of that vertex from the store. As explained later this does not hold true anymore for on-disk storage and additional adaptions are

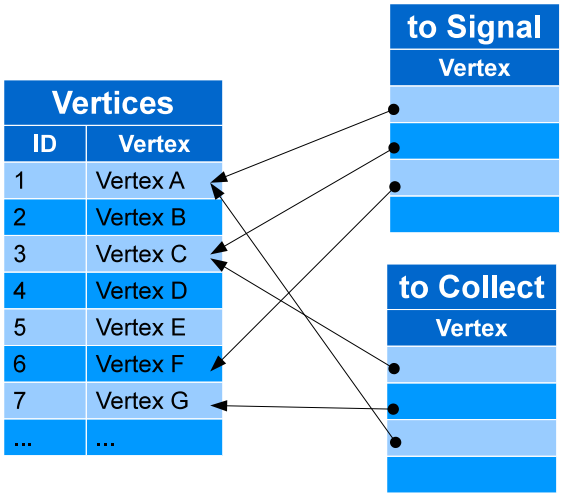


Figure 3.1: Original vertex storage structure

required. Another big advantage of storing all the vertices in a in-memory data structure and accessing them via direct reference, is that changes of the state of a vertex do not need to be retained explicitly. Since the updating function holds a direct reference to the vertex object, the updated state is automatically reflected the next time that vertex is being used, for on-disk storage where the vertex object involved in the computation is basically a copy of the value stored on the disk the changed state has to be explicitly written back to disk to persistently store the updated state.

3.1.2 Generic Vertex Storage Interface Design

To provide access to the vertex store independent of the specific implementation of the storage back end, a new interface had to be specified. That interface needs to take into account all aspects for storing and retrieving vertices from a generic vertex storage. As already mentioned above the worker needs to hold three separate collections to manage vertices. The intention for the new storage interface is to explicitly expose these three collections and the methods available on them. This allows the worker to not only provide a custom implementation for the collection that holds the vertices but optionally also change the implementations of the collections that store which vertices have to signal or to collect. For the collection that holds the actual vertices some new methods had to be added in order to allow on-disk storage. These methods were not needed for the original in-memory vertex storage because the vertices were always directly referenced for computation. One of the methods is the *updateStateOfVertex* method, which causes the

storage of the vertices to serialize the vertex so that its changed state can be persistently written to disk or to a data base. This method needs to be called after every change that is performed on a serialized vertex otherwise this change will not be reflected the next time that vertex is recalled from the storage.
In the collections that hold the information about which vertices need to be processed in the next signal or collect step respectively, the original approach to store references to the vertices in memory is no longer practical. In an on-disk scenario, once the vertex is serialized and written to disk it should be garbage collected to free memory and therefore no more references to that vertex may exist. Later when it is again needed to perform some computational steps, the current state of the vertex should be retrieved from the vertex store. For this reason the new collections for keeping track of which vertices should signal or collect only hold the ids of the vertices that need to be processed. Instead of directly calling the signaling function on the reference the stored id needs to be used to query a vertex from the vertex storage. As shown in listing 3.1 the interface to serve different storage implementations has two different collections for the vertices which need to signal and the ones that have to process a collect step. The main difference is that in the collection for signaling, only the mere ids of the vertices are stored, while the collection for the collect step stores signals that have to be collected. Because the signals have the id of the vertex where they should be delivered included, the collection also knows on which vertices it has to execute the collect function.

Listing 3.1: Storage Interface

```
/**
 * High-level interface to abstract all vertex storage related implementations
 */
abstract class Storage {
  def vertices: VertexStore
  def toSignal: VertexIdSet //collection of all vertices that need to signal
  def toCollect: VertexSignalBuffer // collection of all vertices that need to collect
  def cleanUp
  def serializer: Serializer
}

/**
 * Stores vertices and makes them retrievable through their associated id.
 */
trait VertexStore {
  def get(id: Any): Vertex[_, _]
  def put(vertex: Vertex[_, _]): Boolean
  def remove(id: Any)
  def updateStateOfVertex(vertex: Vertex[_, _])
  def size: Long
  def foreach[U](f: (Vertex[_, _]) => U)
  def cleanUp
}
```

```
/**
 * Allows storing a set of id and iterating through them
 */
trait VertexIdSet {
  def add(vertexId: Any)
  def remove(vertexId: Any)
  def size: Int
  def isEmpty: Boolean
  def foreach[U](f: Any => U, removeAfterProcessing: Boolean)
  def cleanUp
}

/**
 * Allows storing a collection of signals and iterating through them
 */
trait VertexSignalBuffer {
  def addSignal(signal: Signal[_, _, _])
  def addVertex(vertexId: Any)
  def remove(vertexId: Any)
  def size: Int
  def isEmpty: Boolean
  def foreach[U](f: (Any, Iterable[Signal[_, _, _]]) => U, removeAfterProcessing: Boolean, breakConditio
  def cleanUp
}
```

3.2 To Handle collections

As mentioned above, the storage interface exposes three different collections for each worker. One collection holds all the vertices a worker is responsible for and the other two keep track of the vertices that need to be marked for collecting or signaling in a further step.

The collection that keeps track of which vertices have to send a signal to their connected vertices is basically a set of vertex ids with the usual set condition that each id can be only be contained once. In the interface shown in listing 3.1 this collection is therefore named *VertexIdSet* to emphasize this property. The collection supports adding and removing vertices via their ids, which can be thought of as adding the id of a vertex to a symbolic to-do-list. To work off that list a dedicated function exists, that takes a function parameter itself. This function allows applying the provided function to each vertex that is references by an id in the to-do-list. A second parameter is available for the *foreach* function to provide the flexibility of letting the user determine whether the ids in the collection should be removed after iterating over them. With this parameter set to false the framework would support iterating over all the vertex identifiers in the collection, e.g. to print them to a log file, without removing them. When the parameter is set to true it saves the user the burden of removing the ids manually inside the function parameter, if the removing behavior is intended. Because the *toSignal* collections usually do not grow

very large and are updated relatively frequently they are by default kept in an in-memory data structure, such as a Java *HashSet*. However since the collection only holds the vertex ids, storing this collection on disk would also be possible, but obviously results in a significant performance drawback.
The collection that determines the vertices that should collect their received signals is slightly different from the one for signaling. Before the framework supported on-disk storage of vertices, the workers directly forwarded incoming signals to their receivers who then stored them in some collection for processing them later. The old *toCollect* list at the time just stored the references of the vertices and called the collecting function on these vertices. For an on-disk storage of vertices these references have to be replaced by the ids of the vertices as described above. Working the to-do lists would be possible analogous to the way signaling is handled. The collect operation would simply be invoked on all the vertices referenced by the entries in the collection and the retrieved vertices could execute the collect operation on the uncollected signals. The problem with this approach is, that in a scenario where vertices are serialized to disk the previously described process of directly delivering the signals to the vertices is very expensive because the storage back end has to read the vertices from disk just to add one single signal and then write the updated state back to disk again. To reduce the number of these costly reads and updates on the vertices, we decided to buffer the incoming signals for all vertices already at the worker and deliver them all at once when the vertex is executing its collect function. Because this buffer already holds the information about which vertices have signals waiting for them and therefore should be collected, it would have been redundant to also store the ids of all the vertices that need to be collected in an additional collection. In order to avoid this redundancy and to simplify the process of managing the *toCollect* list, we decided to merge the *toCollect* list with the buffer of unprocessed signals. The central component of this new collection is a map that contains an entry for each vertex that has to collect. Each entry holds the id of the vertex as its key and a set of signals that are not yet delivered to this vertex as its value. When working though that collection each vertex referenced by an entry in the map is handed its signals to process the collect function on them. This design also allows requesting a collect operation for a vertex that does not have any signals buffered for it. In the default implementation this would just result in a new entry in the map with an empty list as its value. This way the collect function is still called on the vertex and it lies in the responsibility of the vertex implementation how it should react on a collect operation without new signals. Since the collection will receive new signals from other workers concurrently or while working off its entries, a Java *ConcurrentHashMap* needs to be used here to enable thread safety. Working of the list

is quite similar to the *foreach* function described above with the additional property that a function parameter can be provided that allows to escape the iteration loop through the collection. This could be necessary when a task of higher priority, e.g. checking the inbox, needs to be executed that cannot wait for the collect step to finish. The buffer of the signals is not intended to be stored on disk, because this would ruin the performance gain compared to the original instant delivery of signals in the on-disk case. To prevent the buffer maps to grow too large and consume a significant amount of memory, two different approaches would be feasible. One way to reduce the size of the buffer is to shorten the interval of collecting so that the signals in the buffer are processed faster. This would however only work in the asynchronous execution because in the synchronous case the iteration between signaling and collecting is fixed by definition. For some algorithms, such as a the single source shortest path (SSSP) computation performed in chapter 4, it is also possible to extract the information of the signals in the buffer and only store the relevant data. When a new signal is added to the buffer, a reducing function checks the contained value of that signal and eventually replaces the buffered signal. This optimization is not limited to the SSSP algorithm, but can be performed on any algorithm that allows aggregating the signals.

3.3 Storage Implementations

This section describes three on-disk storage implementations that differ in their storage back end and the way the vertices are stored on disk. For the back ends three different database systems are used. Berkeley DB JE and OrientDB are pure Java database systems that require no additional installation and can be run directly from the included jars. Mongo DB is a open source, document oriented database that allows clustering multiple machines to increase both performance and reliability. Before specifying the on-disk implementations, the in-memory approach is described to serve as a starting point for further reference. The in-memory implementation is also used as the default storage for vertices. Additionally a fifth storage back end is described that stores the vertices in-memory in a serialized form to save memory. With the help of special traits to implement a factory pattern [Gamma et al., 1995], all the components of the default storage implementation can be changed to a different implementation. The default configuration of the storage is shown in listing 3.2.

Listing 3.2: Default configuration of the storage

```
/**
 * Default configuration for storing vertices and the toSignal and toCollect collections
 * Uses in-memory implementations for all collections.
 */
class DefaultStorage extends Storage {

  var vertices = vertexStoreFactory
  protected def vertexStoreFactory: VertexStore = new InMemoryStorage(this)

  var toCollect = vertexSignalFactory //holds all signals that are not collected yet
  protected def vertexSignalFactory: VertexSignalBuffer = new InMemoryVertexSignalBuffer
  var toSignal = vertexSetFactory //holds all vertex ids that need to signal
  protected def vertexSetFactory: VertexIdSet = new InMemoryVertexIdSet(this)

  def serializer: Serializer = DefaultSerializer

  def cleanUp {
    vertexStoreFactory.cleanUp
    toCollect.cleanUp
    toSignal.cleanUp
  }
}
```

3.3.1 In-Memory

The in-memory storage implementation is the current default implementation of the vertex storage. It is mostly consistent with the implementation that existed before the new storage interface was introduced. The core element of this storage is a map data structure that matches vertex ids with their corresponding vertex objects. For performance reasons [1] the *HashMap* from the Java library is here chosen over the corresponding Scala class. Because the vertex id is allowed to be of type *Any*, which is the corresponding class of Java's *Object* class, its hash value is used as key value for the map to provide consistency even if vertices of different types are used. The add and remove methods of the interfaces are directly mapped to the respective functions provided by the underlying map. Since all the vertices returned by the get method are passed by directly referencing the vertex objects in the map, the *updateStateOfVertex* function is not needed here and therefore has no functionality implemented.

3.3.2 Berkeley DB JE

Oracles Berkeley DB Java Edition (JE) [Oracle, 2006] is a pure Java, transactional database that is based on a key/value model and designed for high performance application scenarios. It originated at the University of California Berkeley as part of their open source

[1] http://jameslao.com/2010/10/18/scala-hashmap-performance/

operating system BSD. The software was distributed by Sleepycat Software, which was acquired by Oracle Corporation in 2006. Berkeley DB JE is designed to embed seamlessly into a Java application without requiring any additional installation. Because of its small overhead and the straightforward storage structure it is used for data storages of various sizes, from simple storages in smart phone apps to large scale data processing. For its application in Signal/Collect Berkeley DB JE is very interesting from a usability perspective, because the user does not have to care about starting a database daemon and setting up data folders for it because the database can be directly integrated in the system. Berkeley DB JE is a non-relational database system that provides atomicity, consistency, isolation and durability (ACID) guarantees. In he context of Signal/Collect atomicity assures that a change on a vertex is either fully performed on a vertex or not at all. Consistency requires the database to be in a consistent state before as well as after a storage transaction. This assures that all the stored vertices comply with the properties defined for an algorithm at any point in time. Isolation is especially important to assure that a change on a vertex is not influenced by any other process that is executed concurrently. Durability would be important for providing reliable states in case of a failure. Since error recovery is not yet covered by the Signal/Collect framework, this property is less important right now, but would be a crucial factor when building a fault recovery mode in the future.

Berkeley DB Implementation

Our implementation uses the Berkeley DB JE Direct Persistency Layer (DPL), which allows us to handle the whole storage process at a higher level of abstraction. This avoids the need to store keys and values explicitly as a pair of byte arrays by using the more efficient library functionality. Instead of having two byte arrays, we use a wrapper class to construct an *Entity*-object for each vertex with its id as the entity's primary key. To store the serialized vertex a separate field is added. Our wrapper class is shown in listing 3.3. The enwrapped vertex can then be added to the *EntityStore* object that is part of Berkeley DB's DPL package. The *EntityStore* provides the *PrimaryIndex* that is used to efficiently handle the storage, retrieval and update of the vertices. Each worker has its own *EntityStore* object, while they all share the same environment. The environment configuration also sets the caching factor that determines the fraction of the available memory that can be used by the Berkeley DB to build its own in-memory cache. To speed up the loading phase of the graph, Berkeley DB supports a deferred write mode that allows the database to postpone the writing of the vertices on disk. The deferred writes are held in memory and not persistently written to disk until the store is closed or synced manually.

Listing 3.3: Berkeley DB JE Entity Wrapper

```
@Entity
class Vertex2EntityAdapter(idParam: String, vertexParam: Array[Byte]) {

  @PrimaryKey
  var id: String = idParam
  var vertex = vertexParam

  def this() = this(null, null) // default constructor

}
```

3.3.3 MongoDB

MongoDB[2] is a schema free, document oriented database system. The software is distributed under the GNU Affero General Public License[3] (GNU AGPL v3.0) by the company 10gen[4]. Despite its relatively young age, MongoDB has earned a lot of attention by the open source community and is in production at various places in the industry where scalability is a crucial factor of success. Among the most prominent users of MongoDB are foursquare which uses it for its check-in storage[5] or it is used to enable data aggregation from different relational and non-relational sources of up to several petabytes in CERN's CMS data aggregation system [Kuznetsov, 2010].

Two of the main reasons for the impressive success of MongoDB are its support for replication and its simple but effective sharding functionality. Replication in MongoDB is done by building a replica set consisting of two or more MongoDB instances. Replication is performed asynchronously, where only one instance is allowed to act as the primary member of the replica set which gives it the exclusive right to accept write commands. Depending on the consistency model chosen, reads from other members of the replica set are also allowed. The replication assures that if the primary node of the replica set will stop working, another member of the set will take over its position and act as the new primary member. This provides the redundancy needed especially for distributed scenarios, where the probability that one machine will fail is rather high. Sharding on the other hand provides horizontal scalability for the database. It splits up the data into chunks and distributes them evenly among all available shards and through that increases the capacity to store items and the throughput of the system because it allows for more reads in parallel. Sharding and replication are illustrated in figure 3.2. As it turned out later on MongoDB has performance issues on computers with a non-uniform memory access

[2] http://www.mongodb.org/
[3] http://www.gnu.org/licenses/agpl.html
[4] http://www.10gen.com/
[5] http://www.10gen.com/presentation/misc/foursquare

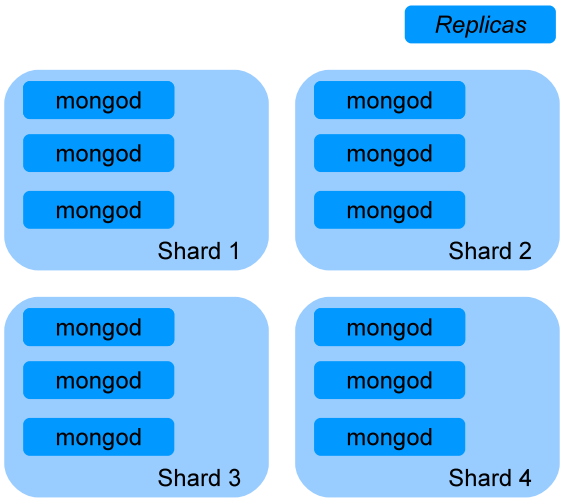


Figure 3.2: MongoDB setup with 4 shards and a replica set with three instances each

(NUMA) design. The effect of this limitation is hard to estimate but the MongoDB community advises to avoid running MongoDB on a NUMA architecture machine because the throughput will be reduced.

MongoDB Implementation

For its use within the Signal/Collect framework casbah[6] which is the official MongoDB driver for Scala is being used. The casbah driver itself uses the MongoDB Java driver as its base to handle the MongoDB wire protocol. In addition to that casbah provides the upper layers of Scala programs with useful features of the functional programming language Scala such as a *foreach* loop that iterates through all stores elements in a MongoDB collection. Casbah would also support conversion from native Scala types such as *List* or *Seq* to their corresponding Java types, which the Java driver for MongoDB then could map to a MongoDB storage entity. Because we wanted to be able to store vertices of arbitrary type and therefore cannot rely on such convenient features to work, we decided to not add the vertices directly to the MongoDB store, but rather use our own serializer to generate a byte array first. For a description of the serializer see section 3.5 below. The generated byte array is then stored in a simple MongoDB document that only consists of two fields. One field stores the id of the vertex and is unique for each vertex in the collection. Because the Signal/Collect framework allows the id to be of any arbitrary type, the string representation of that id is being used to store it. Obviously this requires the

[6] http://api.mongodb.org/scala/casbah/

string representation to be unique as well. The other field in a vertex document holds the serialized representation of a vertex object. Since this is simple byte array it needs no further adaptation and can directly be handled by the casbah adapter. To retrieve a vertex the document containing the id of the requested vertex must be found then the value of the vertex field can be accessed and deserialized to access the vertex-object. For a simple demonstration of the storage and retrieval process see listing 3.4. The update function that is needed to write an updated state of a vertex to disk, works in a similar way by providing a query document and a new document to replace the result of the query in the collection. Since the framework allows for multiple workers which each handle a subset of all the vertices there are also many workers that have to read and write vertices from a MongoDB. For performance reasons we decided to use one single database instance for the whole graph but having a private collection for each worker thread.

Listing 3.4: Example storage and retrieval in MongoDB using casbah

```
// Things needed

val mongoStorage = MongoConnection()(databaseName)(collectionid) // MongoDB collection
val myVertex: Vertex[_, _] = ... // Some vertex
val id = myVertex.id

// Storing a vertex object in the collection

val newDocument = MongoDBObject("id" -> myVertex.id.toString,
        "obj" -> serializer.write(myVertex)) // create a new document
mongoStorage += newDocument     // add the document to the collection

// Retrieve the vertex

mongoStorage.findOne(MongoDBObject("id" -> id.toString)) match {
        case Some(x) =>  {  // read the vertex from the document we have found
                val serialized = x.getAs[Array[Byte]]("obj")
                read(serialized.get).asInstanceOf[Vertex[_, _]]
        }
        case _ => null  // in case no document with that id field exists
}
```

3.3.4 OrientDB

OrientDB[7] is a no-sql database that is optimized for large datasets. It is written in Java and publicly available under the Open Source License Apache 2.0[8]. The maximum number of records in this database is advertised as 9.223×10^{18} when the database is distributed on multiple disks on multiple nodes. OrientDB can run as a singe machine

[7] http://www.orientechnologies.com/

[8] http://www.apache.org/licenses/LICENSE-2.0

server or distributed over multiple instances using a distributed hash table algorithm. Even if OrientDB is part of the so called no-sql databases, it supports SQL as an alternative query language which facilitates the retrieval process. Like MongoDB, it is built as a document-oriented database instead of using a relational structure. This also means that no schemas have to be created in order to store documents. OrientDB uses a special tree algorithm called multi value red black tree (MVRB-Tree), which is essentially a red-black tree with the additional property that each leaf node can contain several values. This adaption to the design of the red black tree allows the MVRB-Tree to use less memory than the standard implementation while allowing for fast retrieving and storing of nodes. Like Berkeley DB JE that was explained above, OrientDB can be run directly from a packaged jar file and needs no additional installation or setup, which would make it convenient to use without having the user to care about which data store is used. Compared to Berkeley and MongoDB, OrientDB is much less known and seldom used in the industry. Because OrientDB is not backed by a large company or community such as the other two mentioned, documentation is also rather rare and except for Java Doc APIs and a promotional webpage almost inexistent. Despite the missing information or references, the published benchmarks on the advertising site promise, that OrientDB could be a serious candidate for the Signal/Collect storage backend.

OrientDB Implementation

In our storage backend based on OrientDB we use a special wrapper class to create a document that contains the vertex in serialized form as well as its id as a separate identifier field to facilitate querying the documents. Listing 3.5 shows this wrapper class. These documents are stored in document databases that are private to the workers. For retrieving the vertices we query the ids of the vertices in a SQL like way. To update the vertices a document that contains the vertex with a given id is retrieved and the field that contains the serialized vertex is replaced by the most recent version. Since the generated id field is not changed, this document replaces the older version when it is reinserted into the database.

Listing 3.5: Vertex wrapper for OrientDB

```
case class OrientWrapper(vertexID: String, var serializedVertex: Array[Byte]) {
  @Id
  var id: String = _ // Internal identifier of the document
  def this() = this(null, null) // default ctor for unmarshalling
}
```

3.3.5 Serialized In-Memory

When comparing the three on-disk storage versions above with the default in-memory implementation two main disadvantages are self-evident. First of all, the objects have to be deserialized every single time a vertex is accessed and since almost accesses also modify the state of the vertex, the changed vertex has to be serialized again to be retained in the store. The second drawback arises from the higher transfer delays that occur when storing data on disk instead of holding it in main memory. Independent of the type of hard drive, the access times are several magnitudes slower than for accessing main memory. To optimize the performance of vertex storage data structures these two bottle necks could be optimized. The specifics of serialization and ways of optimization are covered in section 3.5. However with respect to the overall storage performance, serialization speed is less critical than disk latency, since serialization is only a computational task and does not require disk I/O it scales with the number of available processors. The huge advantage of serialization is that generally a vertex in serialized form uses less memory than the same vertex object in unserialized form. These serialized objects are traditionally stored on a secondary storage device such as a traditional hard drive, which unfortunately comes with the additional cost of higher access times especially when data is distributed among several blocks on a disk. Optimizing block accesses for computations in a graph-like data structure is far from trivial and does not provide a solution to the problem. The rotational latency factor of traditional hard drives could be avoided by using flash based storage devices as shown by Pearce et al [Pearce et al., 2010] but access times are still much slower than access times for main memory because of the limited bandwidth. Because solid state disks are still rather expensive to buy and we wanted the Signal/Collect framework to run on machines with a standard configuration too, we decided to design a additional storage version that handles our problem from a different angle. Rather than trying to optimize disk access patterns and database infrastructures this approach uses in-memory storage capabilities but uses serialization as a method of compression so that a machine can hold more vertices in its main memory than in the default in-memory implementation, while maintaining faster access time and scalability. The resulting implementation uses the same Java **??** as the in-memory implementation described earlier, but the values of that map consist of simple byte arrays rather than the vertices in object representation. For further expanding the maximum amount of vertices that fit into memory the byte array that represents the vertex could also be compressed. Claude and Navarro [Claude and Navarro, 2010] showed that even a compression that makes processing a graph several times slower can be a good choice when it can avoid

the need to store the graph on the much slower disk. The drawback of storing serialized vertices instead of the actual objects is that every time a vertex has to be recalled from the store the byte array has to be deserialized just as in the on-disk case. Compared to the default in-memory case the *updateStateOfVertex* function is now needed because the vertex objects get garbage collected after they are serialized and each request from the store returns a copy of the stored object instead of the reference to it. For vertex sets that are to big to fit into main memory in standard object representation keeping the vertex in a in-memory data structure in a serialized representation can help avoid the performance drawbacks of secondary storage devices. Another advantage of this storage implementation is that it facilitates buffering because vertices on disk and in memory are represented in the same way and can therefore easily be swapped. Serialized in-memory storages could be used where it allows to fit all vertices in memory where this would not have been possible with the standard approach.

3.4 Caching

The on-disk storage implementations described in the previous chapter have two big disadvantages compared to the in-memory solution. First it is indisputable that disk access is several times slower than accessing data that resides in memory. The second disadvantage arises from the fact that all the vertices need to be deserialized and serialized back to the storage every time they are accessed. For many real world algorithms the workload within a graph is not uniformly distributed among the vertices, which means for an execution in Signal/Collect that there are some vertices that have to signal and collect relatively frequently, while others remain idle most of the time. This property could be exploited by keeping these highly active vertices in memory instead of having to serialize them on every access and store them with the overhead of disk latency. Caching functionality is a common feature of an advanced database systems and is also contained in the on-disk storage implementations presented earlier. The drawback of letting the database decide, which elements should be cached and which ones will be needed less frequently and are therefore best stored on disk is, that the vertex needs to be brought already to a serialized form before it is inserted to the database system. However if the vertex will stay in memory the serialization would not have been needed. For this reason, with built in caching, only one of the two overhead factors can be avoided. On the other hand the database caching algorithms allow for more complex caching strategies, possibly also depending on the actual access patterns, which could make up for the

serialization overhead if it leads to a smaller cache miss quota. For algorithms where the active vertices can be determined with relatively straightforward heuristics, using a custom caching strategy would promise better results than the built-in approaches of the databases, because it avoids the need to serialize the entries by holding the vertices in a separate in-memory storage. Finding an appropriate caching strategy for a general-purpose framework like Signal/Collect is a challenging task, because the cache algorithm should be flexible enough to perform efficiently on various algorithms. Apart from performance consideration the cache should also be flexible enough to support different storage backend. To allow the user to cache any storage backend, the provided cache implementations add an additional layer on top of a pluggable generic storage backend. When the cached storage is being filled with vertices, it first fills the in-memory storage of the cache layer and as soon as the caching threshold is reached, it handles evicting parts of the cache according to a predefined strategy. To provide appropriate caching for algorithms with different access patterns, two different cache layers have are provided. The first caching implementation uses a least recently used (LRU) algorithm to evict cached entries. This cache strategy is effective if vertices that were active in the near past are likely to be active in the following step, while inactive vertices will stay inactive with a high probability. However for an algorithm with evenly distributed vertex accesses this cache will not provide any performance gain but add considerable overhead because vertices have to be frequently evicted from the cache and the underlying linked list that holds the access sequence needs to be updated on each access. To avoid these frequent updates, the second cache implementation follows a different strategy for evicting unused vertices from the cache. For this cache each vertex has a cache score that indicates how cache-worthy it is. If the cache is filled up to its limit it iterates through all the cached vertices and evicts the vertices who's score is below the average of all scores. This way insertions to the cache are much faster because the can happen at any free cache slot and no structure needs to be maintained. As an additional benefit the costly evictions from the cache happen less frequently than with the on disk case. For large vertex sets, storing a map of all cache scores would require a lot of memory and therefore this implementation can have a large memory footprint.

3.5 Serialization

Serialization is a fundamental aspect of on-disk storage since each access to a vertex essentially means one deserialization step, i.e. to read the vertex from disk, and one seri-

alization step, i.e. to persistently write it back to the storage. As Opyrchal and Prakash [Opyrchal et al., 1998] and experiments[9] on the web show, the Java default serialization is several factors slower than an optimized serialization process but very simple in usage. The Signal/Collect framework is designed to provide maximal flexibility to allow the user to perform any computation that is mappable into signaling and collecting steps. The computation model works without making any assumptions about the structure of the vertices other than that they must have a unique id and some state, which are each of some arbitrary type. This flexibility is the reason why using a serialization library such as protobuf[10] or kryo[11] was impractical for our case. The user would have had to specify all the types used in a serialization protocol or register them with the serializer. Because of that we decided to use the default Java *ObjectOutputStream* to write our objects. The elegant property of this implementation is, that the user is free to implement the *Externalizable* interface and provide custom serialization and deserialization methods to speed up serialization and reduce the size of the serialized object. An exemplary custom serialization for a vertex that represents a page in a page rank algorithm is displayed in listing 3.6. This serialization has proven to be even faster than prominent the serialization libraries by the experiments mentioned above, because no types need to be checked and serialization can follow a fixed process.

Listing 3.6: Exemplary custom serialization of a vertex

```
class MemoryEfficientPage(var id: Int) extends Vertex[Int, Float] with Externalizable {

  var state = 0.15f
  var lastSignalState: Option[Float] = None
  type UpperSignalTypeBound = Float
  protected var targetIdArray = Array[Int]()
  protected var mostRecentSignalMap: Map[Int, Float] = Map[Int, Float]() // key: signal source id, value


  /* Functions omitted here */

  def this() = this(-1) //default constructor for serialization

  def writeExternal(out: ObjectOutput) {
    out.writeInt(id)
    out.writeFloat(state)
    lastSignalState match {
      case Some(oldState) => out.writeFloat(oldState)
      case None => out.writeFloat(-1) //Safe because a page rank score should not be negative anyway
    }
    // Write links
    out.writeInt(targetIdArray.length)
```

[9] http://code.google.com/p/thrift-protobuf-compare/
[10] http://code.google.com/p/protobuf/
[11] http://code.google.com/p/kryo/

```
    for (i <- 0 until targetIdArray.length) {
      out.writeInt(targetIdArray(i))
    }
    //write most recent signals
    out.writeInt(mostRecentSignalMap.values.size)
    mostRecentSignalMap.foreach(signal => {
      out.writeInt(signal._1)
      out.writeFloat(signal._2)
    })
  }

  def readExternal(in: ObjectInput) {
    id = in.readInt
    state = in.readFloat
    val oldSignal = in.readFloat

    if (oldSignal < 0) {
      lastSignalState = None
    } else {
      lastSignalState = Some(oldSignal)
    }
    // Read links
    val numberOfLinks = in.readInt
    targetIdArray = new Array[Int](numberOfLinks)
    for (i <- 0 until numberOfLinks) {
      targetIdArray(i) = in.readInt
    }
    // Read most recent signals
    mostRecentSignalMap = Map[Int, Float]()
    val numberOfMostRecentSignals = in.readInt
    for (i <- 0 until numberOfMostRecentSignals) {
      mostRecentSignalMap += ((in.readInt, in.readFloat))
    }
  }
}
```

3.6 Other Adaptations

In order to be able to process graphs that are several orders of magnitude larger than the ones that fit in memory of a single machine, some additional changes in the framework had to be made. Some of these changes can be used with arbitrary algorithms, while others are designed to improve the performance of the algorithms we use for the evaluation in chapter 4.

3.6.1 Coordinator Throttling

One major problem we run into when scaling up the number of vertices in a graph is, that the receiver's message inbox size is not considered when sending a message. One place where this is problematic is at the loading phase of the graph. A central coordinator constructs a vertex or an edge and sends the request to add this component encapsulated

as a message to the corresponding worker. After placing the message in the worker's inbox the coordinator already constructs the next message to deliver it to the appropriate worker. The worker thread on the other hand loops through its inbox and processed the requests to add new vertices or edges. In a scenario where the workers store their vertices in-memory, the coordinator and the workers are working at a comparable pace so the inbox size of each worker remains relatively small. This changes when the worker has to do the entire overhead involved with storing vertices on disk. When the worker has to serialize each vertex and store it on disk, processing the inbox takes much more time for the worker than it takes for the coordinator to put new vertices in it. As a result the worker's inbox can grow considerably large and cause the system to run out of memory in the worst case. In a more generic way this is similar to a classical producer-consumer problem but with the additional property that the buffer i.e., the message inbox, is not limited a priori but by the amount of available memory. To avoid this problem we have to slow down vertex creation at the coordinator, when the inbox of a worker is full. Preventing the inbox from growing too big could be achieved in two ways: One approach would require the worker to constantly check its inbox size and compare it to some kind of threshold. If the number of messages in the inbox of that worker was to large, the worker had to send a special message to the coordinator in order to inform it that it should wait for the worker's inbox to shrink. The advantage of this approach is, that the coordinator knows on which workers the load is often too high and could use this information for better balancing the load among all the workers. On the other side checking the size of the inbox is a rather expensive operation because, for the current implementation, it requires iterating through the elements every time the function is called. The other problem of this implementation occurred when the message that indicates that the worker has recovered a reasonable inbox size was lost on its way to the coordinator. The coordinator and the workers ended up in a deadlock situation, because the coordinator was waiting for the workers message that indicates that the it can continue sending new messages, while the recovered worker has emptied its inbox and waits for new messages from the worker. Instead of letting each worker determine on its own when the maximal inbox size is reached, we decided to handle this decision centrally at the coordinator. Each worker periodically sends the number of messages it has sent and the number of messages it has received in a special status message to the coordinator. The coordinator on the other side manages a map that holds the most recent status message from each worker. These status messages allow the coordinator to constantly calculate the global message inbox size, which is equal to the total number of messages sent minus the total number of messages received. Based on this value, the worker then can decide if it can

send another message to a worker or should wait for the inboxes to diminish. Since the worker doesn't have to call the size method on the inbox anymore but rather uses the two message counters that are also used for checking if a worker has complete all its jobs, this is much more efficient and also suited for larger inbox sizes. The drawback of this implementation is that it increases the message flow between the workers and the coordinator even if the inboxes are small and that it only covers global inbox statistics. For a single machine execution this is currently enough information since the available heap space is shared between the workers anyway, but if the Signal/Collect framework should also support load balancing or work distributed over many computers with possibly different memory resources, collecting only global statistics will not provide enough information.

3.6.2 Graph Loading

Apart from throttling the coordinator as described above, the loading process has to be optimized further to handle a large amount of inserted vertices. As opposed to throttling the optimization presented here only applies to our evaluation algorithm and cannot directly be used for arbitrary algorithms. However the general concept can be applied in many other situations as well to reduce loading times of a graph. When loading a graph in memory updates to a vertex that is already loaded are relatively cheap because the objects are accessible directly through a map data structure. For this reason, vertices can be created first and their outgoing edges can be added in a later process. However, once the vertices are stored on disk adding a new edge for a vertex involves seeking the entry in the database, deserializing, adding the new edge, serializing and storing it back on disk. This overhead can be avoided by preassembling the complete vertex including the edges before inserting it in the graph. When our *MemoryEfficientPage* is used this process not only saves the time of serializing and storing the vertices more than once, but also removes the need to generate edge objects. This is possible because this page implementation covers collecting as well as signaling by itself without the need of an additional edge object. Even though for the in-memory case the delay of retrieving a previously added vertex from the store for adding a new edge will not be as high as for the on-disk case, preassembling the vertices is a good way to reduce the number of messages sent to the workers and risking to overflow the inboxes with messages and should therefore be applied when ever possible.

4

Evaluation

This chapter will compare the different storage back ends that were presented in the previous chapter. For comparison we selected the single source shortest path (SSSP) computation and the widely used PageRank [Page et al., 1999] algorithm. The SSSP computation is performed by electing a vertex from the graph as its source and setting the value of its state to be zero. The states of all other vertices in the graph are initialized with a large number that is a lot bigger than the maximal shortest path from the source to any other vertex. The computation starts by letting the designated source signal its state to all the vertices it is connected. For any graph size this means, that in the first round of a synchronous execution mode only one signal operation will be performed. The signal that is sent to a neighboring vertex is computed by adding the weight of the intermediate edge to the state of the source vertex. For simplicity and to conform with other SSSP evaluations in the literature, our edges all have the same weight and therefore we can just add a constant factor to the state of a vertex. The receiving vertex on the other hand, selects the minimal value of all received signals and sets it as its updated state if it is also smaller that the currently stored state. To be able to keep the signal buffers in memory even for larger graphs, the buffer checks the incoming signals if the value contained in the signal is smaller than the one already buffered and keeps only the smallest value in the buffer. The computation is completed when no vertex receives a signal anymore that is smaller than its current state. For non-negative weights of the edges this guarantees that the algorithm will converge. As a result each vertex stores the distance of the shortest path from the source vertex to itself or the initial value if it is not reachable from the source vertex.

The PageRank, as the other algorithm used in this chapter, computes a value of a vertex in a graph as a function of the states of all the vertices that link to it and the number of

its own outgoing links. As opposed to the algorithm in the original Signal/Collect paper [Stutz et al., 2010] the calculation for the signals has changed slightly. As a consequence of merging vertices and edges to one single entity the page is now responsible on its own for calculating and distributing the signals. In the implementation used here the signal is calculated as follows:

$$\text{signal} = \frac{\text{state}}{\text{\# outgoing links}}$$

The semantics of the computed values are not affected by this internal change. In order to be able to evaluate the system on graphs of different size, we decided to use synthetic graphs rather than a fixed size real world dataset. Taking random samples from such a dataset would not guarantee that the samples also reflect the properties observed on the whole graph. To best resemble a scale free real world graph such as the web graph or collaboration networks we use a lognormal distribution of out degrees to construct our synthetic graphs. The probability distribution used is

$$p(d) = \frac{1}{\sqrt{2\pi\sigma^2}d}\, e^{-\frac{(\ln d-\mu)^2}{2\sigma^2}} \tag{4.1}$$

with the parameters μ = 1.3 and σ = 4, which were also used in the evaluation of the Pregel system [Malewicz et al., 2010]. To obtain the out degree of a random vertex we use

$$d = e^{\mu+\sigma R} \tag{4.2}$$

where R is a (pseudo) randomly drawn integer. With a set of vertex using this out degree distribution we build an artificial scale free graph with a mean out degree of 127.1 where the vast majority of all the vertices has a low degree of outgoing edges but a few vertices are highly connected. Clustering properties, which are characteristic for may real world networks including the web graph and social networks, can not be imitated by this synthetic graph generation would have to be considered for deployment in a real world scenario since they affect the flow of computation in almost every case. If not noted differently the evaluations are executed on a computer with two AMD Opteron™6174 processors and 12 cores each, 64GB of RAM and two hard drives that use RAID 0 bit level stripping.

4.1 Serialization

As described earlier, we provide a specialized serialization process for vertices of a PageRank computation. For this algorithm we designed a specialized *Externalizable* version of the standard PageRank vertex called *MemoryEfficientPage*. The *MemoryEfficientPage* without its method declarations was shown in listing 3.6 of the previous chapter. This new page implements the vertex interface directly and can so reduce a lot of the overhead that is needed in the default vertex implementation to provide maximal flexibility. Because the page does not have to hold entire edge objects in a *HashMap* anymore, but uses an array to store just the ids of the pages it links to its size can be considerably reduced. Together with other optimizations on some of the instance variables of the page the serialized size, using generic Java serialization techniques, is 4 times smaller than the one of the default page implementation. When the serialized version is written explicitly by the provided *read-* and *writeExternal* functions of the *Externalizable* interface the size of the serialized vertex can even be reduced by a factor of 12 compared to the default page implementation serialized with the standard Java serializer. Table 4.1 shows the serialized size of these two vertex implementations using different serialization implementations. The upper part of this table shows the sizes of the default page as well as the optimized page implementation. For completeness and to differentiate the effect of the optimized serialization technique from the optimizations in the structure of the vertex, the *MemoryEfficientPage* is also measured with de default serialization procedure i.e., without the methods of the Externalizable interface. On top of the default Java serialization we built another way for serialization, called compressing serializer, which uses the ZLIB compression library to compress the resulting byte array of any serializer to further reduce its size. Since the compression works on byte level and does not need to know about the actual object the byte array represents the compression extension can be used to transform any byte array, including the one resulting from custom serialization on a efficient vertex structure. The lower part of table 4.1 shows, that the effect of compression strongly depends on the structure of the input array. The default page implementation can be reduced to almost half of its original size by applying the compression schema after the vertex has been serialized. On the other hand compression appears to be a bad choice for the *MemoryEfficientPage*. While the page that used the default Java *ObjectWriter* was at least reduced by about 20%, the size of the same vertex even increased after compression when the custom serialization plan was used first. The cost of this size reductions can be seen in table 4.2. For both serializers and both types of vertices tested, the serialization with additional compression took more than three times longer than the simple serial-

ization. The overhead of the additional decompression while deserializing the vertex is lower relative to the standard deserialization time. Considering the additional time that compression consumes, compression is only useful in a setting where compression can be parallelized among several machines and the vertices are in a generic form that benefits from the compression schema. For vertices that are implement the *Externalizable* interface to provide a custom serialization, compression appears to be a poor choice.

Table 4.1: Serialization size comparison of a PageRank vertex

Vertex Type	Serializer	Serialized Size in bytes
Page	Default Serializer	1438
MemoryEfficientPage	Default Serializer	363
MemoryEfficientPage	Custom Serializer	117
Page	Compressing Serializer	780
MemoryEfficientPage	Compressing Serializer	306
MemoryEfficientPage	Compressing Custom Serializer	122

Table 4.2: Serialization speed comparison of a PageRank vertex

Vertex Type	Serializer	Serialization Time in ns	Deserialization Time in ns
Page	Default Serializer	75,635	176,802
MemoryEfficientPage	Default Serializer	57,399	121,488
MemoryEfficientPage	Custom Serializer	47,582	60,959
Page	Compressing Serializer	262,623	212,965
MemoryEfficientPage	Compressing Serializer	233,264	153,257
MemoryEfficientPage	Compressing Custom Serializer	156,057	62,954

4.2 Memory Consumption

The serialization sizes described in the previous section provide important information to estimate the costs of disk I/O and the footprint of the graph on disk. However, to evaluate how many vertices a graph can possibly hold, these measurements are not the only factor to consider. The pure byte arrays of our reduced version of a PageRank page

in serialized form would fit about nine billion times on a singe hard disk with 1TB capacity. Moreover adding more capacity is equally simple and cheap. Another limiting factor that to be considered, even when the possibility of storing vertices on disk exists, is memory consumption. This section shows how the memory consumption at load time for PageRank algorithms. The information about the size of a vertex in memory is an important starting point for selecting the appropriate storage back end for a given problem and environment or to estimate the size of the buffer for an on-disk implementation. The measured vertices are built with the lognormal distribution of out degrees shown in (4.2). For these runs, the collections that hold the information about which vertices have to signal or to collect are left empty because they can be built in the first run of execution. For a PageRank computation these collections are redundant at initialization, since every vertex has to signal and collect in the first round anyway. This saves memory load time at the cost of execution time in the first round. The vertex sizes in table 4.3 are collected while loading graphs of various sizes. Because the memory consumed scaled linearly with the number of vertices in all cases only the relation size per vertex is shown.

Table 4.3: Memory consumption of a PageRank page

Storage back end	Size in KB
InMemory	0.618
Serialized InMemory	0.687
Berkeley DB Cached	0.752
Berkeley DB on disk	0.002

Even though these numbers suggest that it theoretically possible to load over 90 million vertices in 64 GB memory of a single machine, the actual number of vertices that fit into memory is just a fraction of that amount. As described in section 3.6.1 the inboxes require a considerable amount of memory to store the adding requests from the coordinator and the signals during the execution of the algorithm. In a scenario where the vertices already use up a lot of space, the throttling parameters have to be set more restrictive in order to prevent the inboxes to consume too much memory. Also, because the system slowly runs out of memory the garbage collection phases get more frequent and the throughput, i.e. the number of vertices loaded per time units, declines until no additional vertices can be loaded anymore because the workload of the garbage collector is too high. Figure 4.1 shows the throughput as a function of the amount of memory consumed by the vertices in percent of the total available memory. This execution used

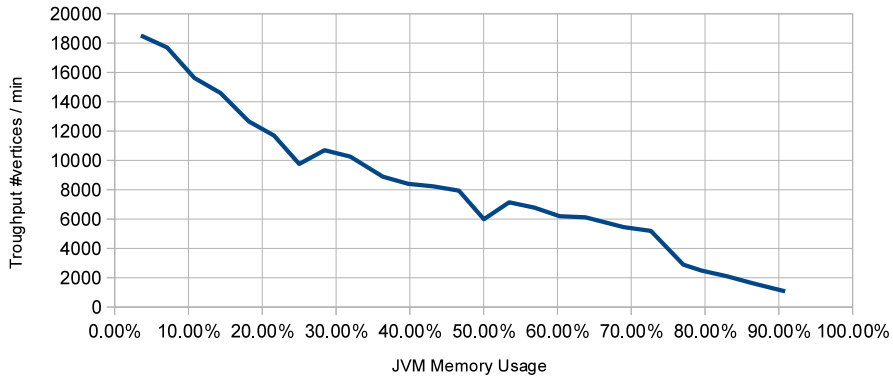


Figure 4.1: Throughput with exhausted memory

the in-memory storage engine, 24 workers and worker throttling to limit the total inbox size of all the workers to 12 million messages. This example shows clearly how the total amount of vertices is limited by the available memory and that the system must leave enough space in the memory for temporary data such as signals or other messages.

To reduce the amount of redundant, temporary data that has to be stored, special reducers were presented. One algorithm whose memory consumption can be decreased by applying such a reducer is the SSSP. Figure 4.2 shows the memory consumption of a synchronous execution of a SSSP computation on a graph with 1 million vertices and about 127 million edges that were stored in a Berkeley DB. The run was executed on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo with 8GB RAM and a 120GB SSD hard drive. The synchronous SSSP execution consisted of five signal- and five collect-steps and consumed 500 MB of RAM in the maximum case. In our simple implementation the signals are propagated through the graph in a wave-like form, where the number of neighboring vertices increases on each signal step. This increase can be observed in the memory footprint during the execution because the size of the signals sent clearly manifests itself in the total memory usage. Therefore each step of the synchronous execution is shown as an actual step in the graph of the overall memory consumption. The zigzag pattern within each step arises from the reducer that continuously discards incoming signals for a location if the included distance is higher or equal to the one already buffered. Without the reducing extension the buffered signals would have easily outgrown the available amount of memory.

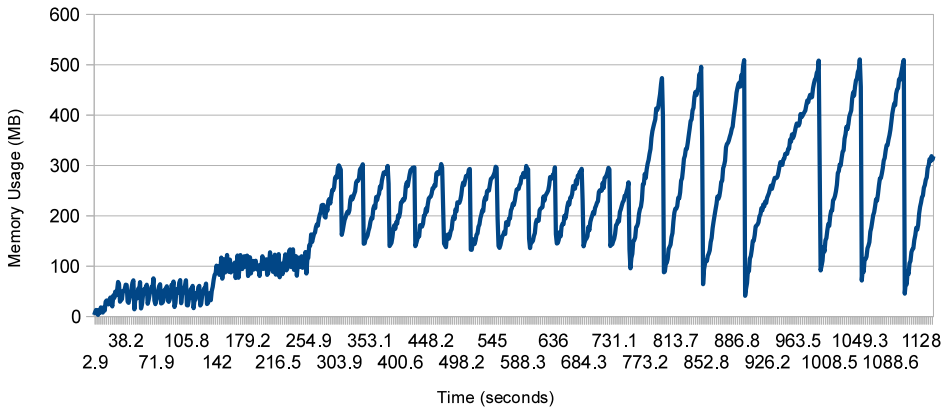


Figure 4.2: Memory consumption for a SSSP algorithm in Berkeley DB

4.3 Page Rank Loading Times

To evaluate how fast the storage back end can be filled with vertices, we tried to measure the pure loading time of each implementation. Normally when working with generated graphs the creation of new vertices and their storage would happen in parallel. Because we chose a lognormal distribution, where some of the vertices have a huge out degree the creation of the vertices was a limiting bottleneck for fast storage back ends such as the in-memory implementation but also for cached versions of on-disk storages like Berkeley DB. To measure only the loading time without the generation of the vertices, the vertices were created in advance and then held in memory for later inserting them into the different storage back ends. This storage process is repeated five times for a set of one million vertices and the average loading time is shown in table 4.4. For this evaluation we did not use throttling, since the 1 million adding requests easily fit into memory. The *toCollect* and *toSignal* collections are filled with the ids of the vertices while loading them into the storage, but because they are held in the same in-memory data structures for all storage back ends, they take the same amount of time for each implementation and can therefore be neglected in the comparison.

Comparing the serialized version of the in-memory storage with the baseline of the classical in-memory version, the overhead from serializing the vertices before storing them in the same data structure is apparent. These figures also help to split up the load times of the on disk versions in a serialization part, which is independent of the storage backend, and a part responsible for writing them persistently on disk. For Berkeley DB

Table 4.4: Average loading times of different storage back ends for 1 million vertices

Storage back end	Load time in s
InMemory	15.695
Serialized InMemory	19.148
Berkeley DB (cached)	22.264
Berkeley DB (no cache)	27.435
Mongo DB	34.087
Orient DB	25.173

the loading time is also dependant of the size of the cache. The *cached* version, that uses a cache size limitation well above the size that one million vertices would consume and has the deferred write mode set enabled, is considerably faster than the default *no cache* version that uses the normal write mode. This speedup arises from caching the write requests in-memory and only restoring persistency when this is explicitly requested by the user. This is still slower than the serialized in-memory approach but avoids the disk I/O involved with storing the file on disk at load time. Our MongoDB driver for Scala as well as the OrientDB implementation does not provide any function to implement a non-overwriting insertion, that checks first if an entry with the same identifier already exists in the graph. To provide this functionality the database has to be queried first to see if a entry already exist and the call a second function to insert the object. For not slowing down the insertion we omitted this assertion and directly inserted the vertices to the store regardless of whether an entry already existed or not. Using the same configuration with 10 million in table 4.5 shows that the load time for the two in-memory implementations scaled about linearly with the number of vertices, while Berkeley DB even used less time per vertex as in the smaller test run because the storage optimization mechanisms do not scale linearly with the with the size of the content of the database. Mongo DB and OrientDB were excluded from this bigger test run since together with the pre-initialized vertices they used too much virtual memory and therefore could not complete. MongoDB does not allow to specify the amount of memory it uses, like for instance Berkeley DB does, because it uses the replacement strategies of the operating system and therefore it couldn't be tuned to fit into memory. Likewise Orient DB did not allow setting the memory portion but used a operating system functionality based on virtual memory that failed for the bigger test run.

Table 4.5: Average loading times of different storage back ends for 10 million vertices

Storage back end	Load time in s
InMemory	157.713
Serialized InMemory	205.796
Berkeley DB (cached)	211.194
Berkeley DB (no cache)	286.333

4.4 Run Times SSSP

To show how the on-disk version of Signal/Collect performs compared to a distributed implementation, we repeated the SSSP experiment that is was performed on Pregel [Malewicz et al., 2010]. We performed the SSSP algorithm on the same highly interconnected synthetic graph with a lognormal distribution of out degrees and an average out degree of 127.6 that was also used in the Pregel experiments. As algorithm for the SSSP computation, the same simple state propagation as in the reference experiment was used too. The computation was performed on a computer with 8 cores, 72 GB RAM and an array of eight hard disk joined in a RAID 10, which means that it uses bit level stripping as well as mirroring and can write on four disks concurrently. Instead of using a distributed system as in the Pregel experiment we used a single machine that stored all the vertices on disk in a Berkeley DB, while keeping the lists of vertices that have to signal and to collect in memory. As in Pregel's implementation our system used a synchronous mode of execution to reduce the number of signal and collect operations and the signal buffer used a reduction function so that it only stored the lowest incoming signals to save memory.

While the Pregel system scaled about linearly for graphs of 100 million to 1 billion vertices, the runtime of our implementation skyrocked already at a vertex count of 10 millions. Compared to the 100 seconds the Pregel system took to compute the same algorithm on a graph with 100 million vertices the on-disk computation times are rather disillusioning and show the clear disadvantage of the on-disk approach. Figure 4.3 shows the runtimes of SSSP runs on graphs of increasing size and their almost exponential growth. As a reason for these huge performance drawbacks two explanations would be possible. First a large number of threads that try to access a small number of disks concurrently slow down the reading speed because the data blocks are randomly distributed over the disk. Also for a single reader thread scenario the different vertices are distributed over the whole disk but reading a single vertex is not interrupted because another thread re-

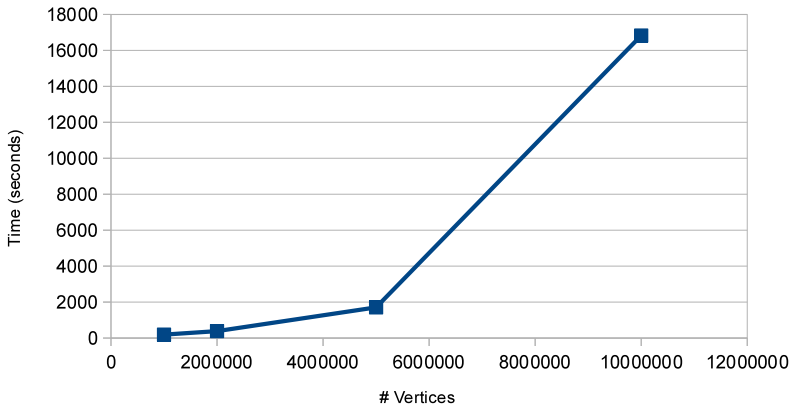


Figure 4.3: Runtimes of SSSP using Berkeley DB JE

quests different blocks from a totally different location on the disk. The other explanation can be observed in figure 4.4 that shows the runtime of each step during the synchronous execution of our SSSP algorithm. Because the vertices are stored on disk and the main portion of the time is spent on accessing vertices, we correlate the used time directly with the number of vertices retrieved form disk. That this assumption generally holds true can be observed in figure 4.5, where the SSSP algorithm was performed on a graph, where all the vertices are chained in one line with an edge that connects each vertex with its successor. By selecting the first vertex of that chain as the origin of the SSSP computation we guarantee that each vertex will only collect once and the number of accessed vertices scales linearly with the graph size. This graph shows that the runtime of this artificial SSSP computation will scale perfectly linear with the number of vertices accessed during the computation. The vertex-access pattern during the SSSP computation can therefore be observed by the time recorded in figure 4.4. The peak in the runtime of the fourth signal step shows that for the one and two million vertices graphs the majority of the vertices receive their distance in the third collect step. The following fourth signal step's runtime peaks, because all the vertices that received a signal that was lower than their old state propagate their new states to their neighbors. The next collect step takes less time than the signal step before, because the sent signals contain a lot of redundant information that was already filtered out by the reducers in the signal step. Because most of the vertices already received their final state in the third iteration, a lot less vertices have to propagate an updated step. For this reason, signaling in step five and the following collect are fairly quick. For the five million vertices run, the third collect step did

not yet spread wide enough. Because of that, the signal step of the fourth iteration has to spread the new signals to a larger number of vertices than it was the case for the two smaller graphs. This can be observed because the time of the following fourth collect step is higher since the signals contained less redundancy and therefore there were more vertices to retrieve than in the preceding signal step. Because a large proportion of the vertices collected a new state in step four and need to propagate their state, the signal operation in the fifth step takes longer than the one of the previous step. Since these signals contained a lot of redundancy the runtime of the following collect step is again much shorter and because all of the vertices have already received their final state the computation is terminated.
This graphic illustrates that a slower signal propagation compared to the graph size will result in a runtime that can not scale linearly on-disk because the number of collect and signal operations increase disproportionally high. For in-memory executions this effect seams less dramatic because before collecting, the worker can decide if the reduced signal is worth to be collected compared to the current state of the vertex. In the on-disk case, the vertex has to be retrieved and deserialized in order to access its state, which means that the main work is already performed before the vertex can decide if collecting the new signals is necessary at all. After the collecting phase the vertex can then decide if it has to signal its state or remain quiet. Essentially this means that in our on-disk case the vertices are read and deserialized for the collection operations even if the state will not change after the collecting of the signal. As a solution to that the buffer of the signals could also store the current state of a vertex. This would allow to determine if the collect operation will effect the vertex's state or if reading the vertex from the data base would be purposeless anyway. The disadvantage of this implementation is that it would require a lot more memory for the buffer and would harm the whole concept of storing entities with their states encapsulated from the signals through the collect functionality.

4.5 Scalability of SSSP using Berkeley DB JE

One of the crucial benefits of a parallel graph processing is that the run time is negatively correlated with the number of available worker threads, if enough processor cores are available to natively run them [Stutz et al., 2010] [Malewicz et al., 2010] [Haller and Miller, 2011]. For the Signal/Collect framework, Stutz et al. reported an almost linear scalability of a SSSP algorithm for 1 to 8 cores. To investigate the impact of on disk storage on the runtime of a SSSP algorithm we performed the same algorithm on a graph with the same

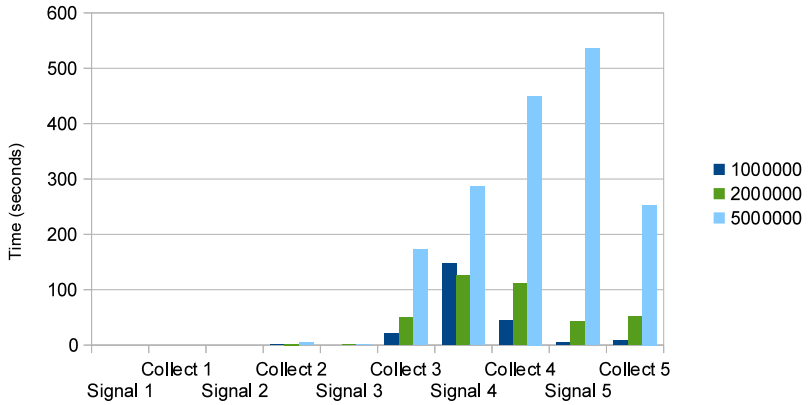


Figure 4.4: Runtimes of SSSP using Berkeley DB JE - Per synchronous iteration step

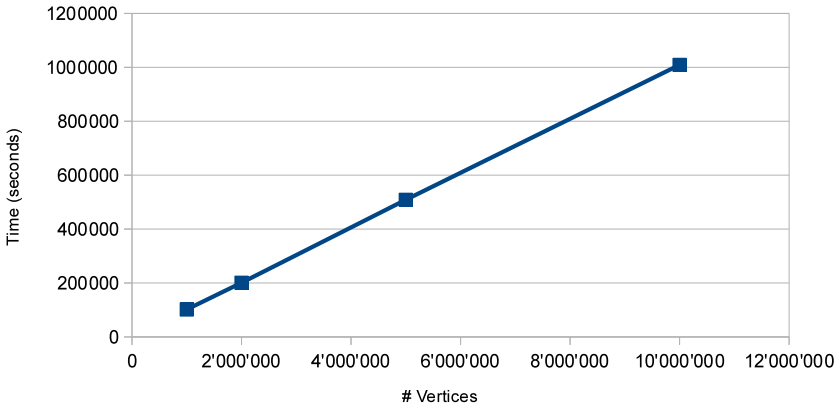


Figure 4.5: Scalability of SSSP using Berkeley DB JE on a chain

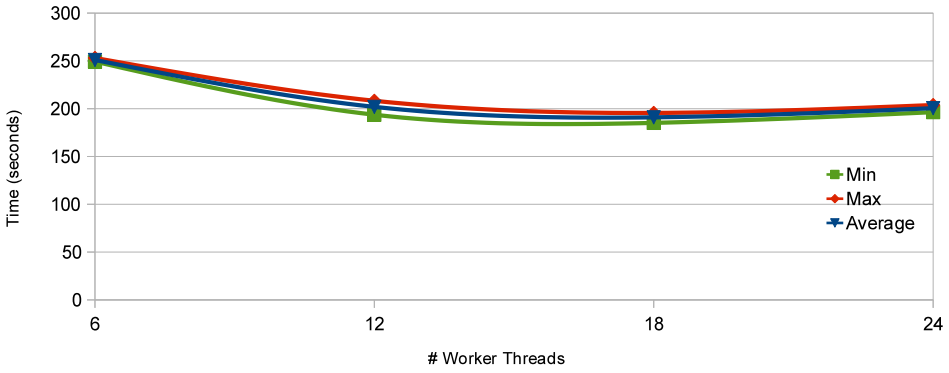


Figure 4.6: Scalability of SSSP using Berkeley DB JE

amount of vertices but a higher out degree. Regardless of the out degree, figure 4.6 shows clearly that the speedup manifested in the in-memory execution can not be reached, when storing the vertices on disk. It is obvious that reading the files from disk is the new bottleneck, while the computation of the states have less impact on the overall runtime of the algorithm. Some minimal performance gains can be observed, because parts of the computation, such as the computation of the states or the serialization process, can still be parallelized. However the limiting factor for this scenario is obviously disk access. Disk access is not parallelized and therefore it does not benefit from the increasing number of worker threads. The worker threads are slowed down during a SSSP algorithm that has to read and write all vertices from and to disk to about nine percent CPU activity each. How the parallelization would perform, when each worker thread had its own disk to write to, is beyond the scope of this evaluation but most likely some of the speedup could be regained because the workers would not have to share the bandwidth of a single disk or a small number of them as in our RAID 0 environment.

5

Related Work

The proposed storage back ends provide the Signal/Collect framework with the ability to reduce its in-memory footprint by storing parts of the graph on-disk and thereby allowing it to hold more vertices and edges than just the ones that would have fitted into memory. The storage back ends are interchangeable and, depending on the graph size and the available resources, an appropriate solution can be assembled. Possible configurations reach from a setting where all elements of the algorithm are held in memory, over mixed versions where the a portion or all of the vertices are persistently written to disk, while the *toSignal* and *toCollect* lists are kept in memory, to a very low memory consuming configuration, where all the graph elements as well as the tracking lists are stored on disk. Presented approach to store parts of the graph on disk stands in contrast to similar graph processing frameworks that follow different strategies to overcome the limitation of the maximal graph size by the availability of memory on a single machine.

Google's Pregel system [Malewicz et al., 2010] scales by distributing the workload among a set of commodity computers. All the vertices are distributed over many worker machines, whereby the algorithm that determines the responsible machine for a vertex can be explicitly specified to exploit locality effects. The system incorporates a persistent storage feature to enable fault recovery. Because of its synchronous execution mode Pregel can store checkpoints that hold safe fallback states for the system. As storage system it relies on Google's own GFS [Ghemawat et al., 2003] and BigTable [Chang et al., 2006] technology. Haller and Miller presented a framework [Haller and Miller, 2011] that is conceptually similar to the Signal/Collect approach and where the vertices exchange messages upon which the receiving vertex can update its state. Unlike Pregel it does not require the algorithm to run synchronously and can handle parts of the computation in parallel without central coordination. In their description of the programming model

they did not present a way to extend the framework to handle datasets that are too large to fit into memory of a single machine. However they mentioned their intentions to distribute the framework and implement fault handling in the near future, which would most likely require storing some information about safe fallback states. The GraphLab Project[Low et al., 2010] by Carnegie Mellon University is a third parallel framework that was originally designed for machine learning algorithms. It executes MapReduce tasks by mapping them in a graph structure for processing the calculations. In the implementation that they used in the paper to present the general computing model, all the graph elements are stored in shared memory and no fallback states are implemented. However, they also stated their goal to make GraphLab distributed to handle larger data sets. To the best of our knowledge there exists no similar framework that relies only on disk, or has intentions to do so, to enable processing graphs that would not fit into memory. Most likely the latency of frequent disk access is too high for such computational models and therefore disk storage is only used to store periodical snapshots of the graph for eventual fault recovery.

6

Conclusions and Future Work

This thesis presented several ways to extend the constraints of limited main memory availability for algorithms running based on the Signal/Collect framework. Since memory is one of the first limitation factors faced when increasing the size of a stored graph, reducing the in-memory footprint of an execution directly allows for more vertices to be stored. By serializing and storing parts or all of the vertices on disk, instead of holding them in an in-memory data structure, the objects that represent these vertices can be garbage collected to free memory. This allows to have enough memory left to hold the vertices that are needed for the current computation even if the graph size is very large. Apart from storing the vertices the system also has to hold other information about the current state of execution of the algorithm to control the execution of operations on the stored vertices. Storing the vertices on disk and only retrieving them when they are needed allows this utility information to use more memory than it could have used when it had to share the memory with all the vertices. By storing not only vertices and edges, but also all the elements that determine the control sequence of a Signal/-Collect algorithm, the maximum possible graph size would in theory only be limited by the available storage capacity of the secondary storage device. The used main memory would be constantly low because only the information that is currently needed would be retrieved from disk to perform some operations on it and would then be saved back to disk. The memory footprint would therefore only depend on the number of concurrent operations performed and be independent of the actual graph size. However, in a practical application, the supporting elements, such as the collections that store the information about the vertices that have to collect or to signal, need to be kept in memory. The slowdown of several magnitudes, when storing this information on disk as well is unacceptably high. This finding limits the effect of storing the vertices on disk, since it

only extends the limitations of the scarce main memory but does not provide a solution to overcome it. This means that the on-disk storage back ends presented will not allow to extend the maximum number of vertices that can be processed on a single computer to a billion vertices. It will however enable storing an additional amount of entries compared to a setting where all vertices are held in memory only. Additionally a number of adaptions of the Signal/Collect framework have been proposed that also benefit in-memory as well as a possible distributed application of Signal/Collect. They can both profit from the measures taken to reduce the memory consumption at loading time as well as during execution. Even with all these optimizations in place, the computation of SSSP was several magnitudes slower than a comparable distributed setting which shows that pure on disk storage is not qualified to provide scalability for graphs of very large size.
The evaluations have shown that in order to process data sets that exceed the limitation faced by the constraint of memory, a distributed version of Signal/Collect could provide the needed scalability. Indisputably a distributed case also increases the latency of a computation, because the messages need to be over the network, but overcomes the restriction of main memory limitations by adding more memory. Because in such a distributed scenario a the probability of a node failure increases with the number of nodes being uses, it would be valuable to have some persistent fallback state that is written on disk and probably replicated on other nodes. The vertex storages presented could easily be used to save the state of the vertices in such an application scenario. Another optimization that would improve the performance of a distributed as well as a single-machine implementation is the possibility to better balance the vertices among the different worker threads. Load balancing would also benefit from the introduction of the presented storage interface because it looses the tight coupling between workers and their vertices.
Even tough the on-disk storage did not provide the scalability I wished for, I am still impressed with the amount of vertices that now can be handled by one single machine. I believe that together with the distributed version and all the other optimizations performed around these two approaches, the on-disk storage will make Signal/Collect an even more attractive graph processing framework that suits not only different algorithms but can also be configured to exploit the resources of different hardware environments.

List of Figures

List of Tables

List of Listings

Bibliography

[Chang et al., 2006] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006). Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA. USENIX Association.

[Claude and Navarro, 2010] Claude, F. and Navarro, G. (2010). Fast and Compact Web Graph Representations. *ACM Trans. Web*, 4.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA. ACM.

[Gulli and Signorini, 2005] Gulli, A. and Signorini, A. (2005). The indexable web is more than 11.5 billion pages. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pages 902–903, New York, NY, USA. ACM.

[Haller and Miller, 2011] Haller, P. and Miller, H. (2011). Parallelizing Machine Learning-Functionally: A Framework and Abstractions for Parallel Graph Processing. In *2nd Annual Scala Workshop*.

[Kuznetsov, 2010] Kuznetsov, V. (2010). The CMS data aggregation system. *Procedia Computer Science*, 1(1).

[Low et al., 2010] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2010). Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA. ACM.

[Opyrchal et al., 1998] Opyrchal, L., , Opyrchal, L., and Prakash, A. (1998). Efficient Object Serialization in Java. In *Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshops (31 May-4*.

[Oracle, 2006] Oracle (2006). Berkeley DB Java Edition Architecture. Retrieved July 30, 2011, from `http://www.oracle.com/technetwork/database/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdf`.

[Page et al., 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab.

[Pearce et al., 2010] Pearce, R., Gokhale, M., and Amato, N. M. (2010). Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA. IEEE Computer Society.

[Stutz et al., 2010] Stutz, P., Bernstein, A., and Cohen, W. (2010). Signal/collect: graph algorithms for the (semantic) web. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 764–780, Berlin, Heidelberg. Springer-Verlag.