# Contents

1 Introduction  
   1.1 The Benefits of Titan ......................................................... 1  
   1.2 Getting Started ................................................................. 3  
   1.3 Additional Background and Orientation Resources ............................ 7

2 Using Titan  
   2.1 TinkerPop ................................................................. 9  
   2.2 Transactions ................................................................. 9  
   2.3 Gotchas and Limitations ...................................................... 13

3 Managing Titan  
   3.1 Storage Backends ............................................................. 19  
   3.2 Deploying on Managed Machines ............................................. 19  
   3.3 Cassandra Specific Configuration .......................................... 22  
   3.4 Global Graph Operations ..................................................... 24  
   3.5 Deploying on Amazon EC2 ................................................... 24
1.1 The Benefits of Titan

Titan is designed to support the processing of graphs so large that they require storage and computational capacities beyond what a single machine can provide. This is Titan’s foundational benefit. This section will discuss the various specific benefits of Titan and its underlying, supported persistence solutions.

1.1.1 General Titan Benefits

- Support for very large graphs. Titan graphs scale with the number of machines in the cluster.
- Support for very many concurrent transactions. Titan’s transactional capacity scale with the number of machines in the cluster.
- Support for geo, numeric range, and full text search for vertices and edges on very large graphs.
- Native support for the popular property graph data model exposed by Blueprints.
- Native support for the graph traversal language Gremlin.
- Easy integration with the Rexster graph server for programming language agnostic connectivity.
- Numerous graph-level configurations provide knobs for tuning performance.
- Vertex-centric indices provide vertex-level querying to alleviate issues with the infamous super node problem.
- Provides an optimized disk representation to allow for efficient use of storage and speed of access.
- Open source under the liberal Apache 2 license.

1.1.2 Benefits of Titan with Cassandra

- Continuously available with no single point of failure.
- No read/write bottlenecks to the graph as there is no master/slave architecture.
- Elastic scalability allows for the introduction and removal of machines.
- Caching layer ensures that continuously accessed data is available in memory.
- Increase the size of the cache by adding more machines to the cluster.
- Integration with Hadoop.
• Open source under the liberal Apache 2 license.

1.1.3 Benefits of Titan with HBase

• Tight integration with the Hadoop ecosystem.
• Native support for strong consistency.
• Linear scalability with the addition of more machines.
• Strictly consistent reads and writes.
• Convenient base classes for backing Hadoop MapReduce jobs with HBase tables.
• Support for exporting metrics via JMX.
• Open source under the liberal Apache 2 license.

1.1.4 Titan and the CAP Theorem

“Despite your best efforts, your system will experience enough faults that it will have to make a choice between reducing yield (i.e., stop answering requests) and reducing harvest (i.e., giving answers based on incomplete data). This decision should be based on business requirements.”

—Coda Hale

When using a database, the CAP theorem should be thoroughly considered (C=Consistency, A=Availability, P=Partitionability). Titan is distributed with 3 supporting backends: Cassandra, HBase, and BerkeleyDB. Their trade-offs with respect to the CAP theorem are represented in the diagram below. Note that BerkeleyDB is a non-distributed database and as such, is typically only used with Titan for testing and exploration purposes.

HBase gives preference to consistency at the expense of yield, i.e. the probability of completing a request. Cassandra gives preference to availability at the expense of harvest, i.e. the completeness of the answer to the query (data available/complete data).
# 1.2 Getting Started

In the beginning, there existed two deities known as Uranus and Gaia. They gave birth to the Titans (a race of powerful beings). Saturn, Titan of time, set reality in motion. Ultimately, time yielded the existence of the sky, the sea, and the end of life–death. To rule these notions, Saturn had three sons: Jupiter (sky), Neptune (sea), and Pluto (underworld). The son’s of Saturn were not Titans, but a race of seemingly less powerful deities known the world over as the Gods. Fearful that his sons would overthrow him, Saturn devoured them and imprisoned them in his stomach. This caused a great war between the Titans and Gods. Ultimately, the Gods won and Jupiter took the throne as leader of the Gods.

## 1.2.1 The Graph of the Gods

The examples in this section make extensive use of a toy graph distributed with Titan called *The Graph of the Gods*. This graph is diagrammed below. The abstract data model is known as a property graph and this particular instance describes the relationships between the beings and places of the Roman pantheon. Moreover, special text and symbol modifiers in the diagram (e.g. bold, underline, etc.) denote different schematics/typings in the graph.

![Graph of the Gods](image)

<table>
<thead>
<tr>
<th>visual symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>bold key</td>
<td>a graph indexed key</td>
</tr>
<tr>
<td>bold key with star</td>
<td>a graph indexed key that must have a unique value</td>
</tr>
<tr>
<td>underlined key</td>
<td>a vertex-centric indexed key</td>
</tr>
<tr>
<td>hollow-head edge</td>
<td>a functional/unique edge (no duplicates)</td>
</tr>
<tr>
<td>tail-crossed edge</td>
<td>a unidirectional edge (can only traverse in one direction)</td>
</tr>
</tbody>
</table>

## 1.2.2 Downloading Titan and Running the Gremlin Shell

Unbeknownst to the Gods, there still lived one Titan. This Titan can not be seen, has no name, and is only apparent in the fact that reality exists. Upon the shoulders of this lost Titan, all of reality hangs together in an undulating web of relations.

Titan can be downloaded from the Downloads section of the project repository. Once retrieved and unpacked, a Gremlin terminal can be started. The Gremlin REPL (i.e. interactive shell) is distributed with Titan and differs slightly from the main TinkerPop Gremlin distribution in that is comes preloaded with Titan-specific imports and helper methods. In the example below, *titan.zip* is used, however, be sure to unzip the zip-file that was downloaded.
The Gremlin terminal is a Groovy shell. Groovy is a superset of Java that has various shorthand notations that make interactive programming easier. Likewise Gremlin is a superset of Groovy with various shorthand notations that make graph traversals easy. The basic examples below demonstrate handling numbers, strings, and maps. The remainder of the tutorial will discuss graph-specific constructs.

```
\,,,,/
(o o)
-------oOo-(_)-oOo-------
gremlin>
```

```
100-10
==>90
```
```
"Titan:" + " The Rise of Big Graph Data"
==>Titan: The Rise of Big Graph Data
```
```
{name:'aurelius',vocation:['philosopher','emperor']}
==>
name=aurelius
vocation=[philosopher, emperor]
```

**NOTE**: Please refer to GremlinDocs for a easy to use Gremlin reference.

### 1.2.3 Loading Data Into Titan

The example below will load *The Graph of the Gods* dataset diagrammed above into Titan. When working with a fresh graph (beyond this tutorial), TitanFactory provides methods to create various Titan instances (e.g. local, distributed, etc.). A local, single machine instance of Titan is created using the `TitanFactory.open(String directory)` method. Other pages in the documentation demonstrate distributing Titan across multiple machines, for instance [[using Cassandra]] or [[using HBase]]. Refer to the [[storage backend overview]] on how to choose the optimal persistence mode. For the purpose of this tutorial, a pre-constructed local graph is provided that is loaded with the above *Graph of the Gods* diagram.

```
g = GraphOfTheGodsFactory.create('/tmp/titan')
```
```
g = TitanFactory.open('bin/cassandra-es.local')
```
```
GraphOfTheGodsFactory.load(g)
null
```

Please see the GraphOfTheGodsFactory source code for details.

For those using Titan/Cassandra (or Titan/HBase), be sure to make use of `bin/cassandra-es.local` (or `bin/hbase-es.local`) and `GraphOfTheGodsFactory.load()`.
1.2.4 Global Graph Indices

The typical pattern for accessing data in a graph database is to first locate the entry point into the graph using a graph index. That entry point is an element (or set of elements) – i.e. a vertex or edge. From the entry elements, a Gremlin path description describes how to traverse to other elements in the graph via the explicit graph structure.

Given that there is a unique index on name property, the Saturn vertex can be retrieved. The property map (i.e. the key/value pairs of Saturn) can then be examined. As demonstrated, the Saturn vertex has a name of “saturn,” an age of 10000, and a type of “titan.” The grandchild of Saturn can be retrieved with a traversal that expresses: “Who is Saturn’s grandchild?” (the inverse of “father” is “child”). The result is Hercules.

```
gremlin> saturn = g.V('name','saturn').next()
===>v[4]
gremlin> saturn.map()
===>name=saturn
===>age=10000
===>type=titan
gremlin> saturn.in('father').in('father').name
==>hercules
```

The property place is also in a graph index. The property place is an edge property. Therefore, Titan can index edges in a graph index. It is possible to query The Graph of the Gods for all events that have happened within 50 kilometers of Athens (latitude:37.97 and long:23.72). Then, given that information, which vertices were involved in those events.

```
gremlin> g.query().has('place',WITHIN,Geoshape.circle(37.97,23.72,50)).edges()
===>e[2T-o-2F0LaTPQBM][24-battled->40]
===>e[2R-o-2F0LaTPQBM][24-battled->36]
gremlin> g.query().has('place',WITHIN,Geoshape.circle(37.97,23.72,50)).edges().collect {
   it.bothV.name.next(2)
}
==>hercules, hydra
==>hercules, nemean
```

Graph indices are one type of index structure in Titan. Graph indices are accessible via the Graph.query() method. The second aspect of indexing in Titan is known as vertex-centric indices. Vertex-centric indices are accessible via the Vertex.query() method. Vertex-centric indices are described later.

1.2.5 Graph Traversal Examples

Hercules, son of Jupiter and Alcmene, bore super human strength. Hercules was a Demigod because his father was a god and his mother was a human. Juno, wife of Jupiter, was furious with Jupiter’s infidelity. In revenge, she blinded Hercules with temporary insanity and caused him to kill his wife and children. To atone for the slaying, Hercules was ordered by the Oracle of Delphi to serve Eurystheus. Eurystheus appointed Hercules to 12 labors.

In the previous section, it was demonstrated that Saturn’s grandchild was Hercules. This can be expressed using a loop. In essence, Hercules is the vertex that is 2-steps away from Saturn along the in('father') path.

```
gremlin> hercules = saturn.as('x').in('father').loop('x')(it.loops < 3).next()
===>v[24]
```

Hercules is a demigod. To prove that Hercules is half human and half god, his parent’s origins must be examined. It is possible to traverse from the Hercules vertex to his mother and father. Finally, it is possible to determine the type of each of them – yielding “god” and “human.”
The examples thus far have been with respect to the genetic lines of the various actors in the Roman pantheon. The Property Graph Model is expressive enough to represent multiple types of things and relationships. In this way, *The Graph of the Gods* also identifies Hercules’ various heroic exploits — his famous 12 labors. In the previous section, it was discovered that Hercules was involved in two battles near Athens. It is possible to explore these events by traversing battled edges out of the Hercules vertex.

```
gremlin> hercules.outE('battled').has('time', T.gt, 1).inV.name

=> hydra
=> cerberus
```

The edge property `time` on battled edges is indexed by the vertex-centric indices of a vertex. Retrieving battled edges incident to Hercules according to a constraint/filter on `time` is faster than doing a linear scan of all edges and filtering (typically O(log n), where n is the number incident edges). Gremlin is intelligent enough to use vertex-centric indices when available. A `toString()` of a Gremlin expression shows the underlying query pipeline

```
gremlin> hercules.outE('battled').has('time', T.gt, 1).inV.name.toString() 

=> [StartPipe, VertexQueryPipe(out, [battled], has, edge), IdentityPipe, InVertexPipe, PropertyPipe(name)]
```

### 1.2.6 More Complex Graph Traversal Examples

In the depths of Tartarus lives Pluto. His relationship with Hercules was strained by the fact that Hercules battled his pet, Cerberus. However, Hercules is his nephew – how should he make Hercules pay for his insolence?

The Gremlin traversals below provide more examples over *The Graph of the Gods*. The explanation of each traversal is provided in the prior line as a `//` comment.

#### Cohabiters of Tartarus

```
gremlin> pluto = g.V('name', 'pluto').next()

=> v[32]
gremlin> // who are pluto's cohabitants?
gremlin> pluto.outE('lives').in('lives').name

=> pluto
=> cerberus
```
Pluto’s Brothers

Finally, Pluto lives in Tartarus because he shows no concern for death. His brothers, on the other hand, chose their locations based upon their love for certain qualities of those locations.

gremlin> pluto.outE('lives').reason
===>no fear of death
gremlin> g.query().has('reason',CONTAINS,'loves').edges()
==>[2B-g-2F0LaTPQBU][16-lives->8]
==>[2H-k-2F0LaTPQBU][20-lives->12]
gremlin> g.query().has('reason',CONTAINS,'loves').edges().collect{
  [it.outV.name.next(),it.reason,it.inV.name.next()]
}
==>[jupiter, loves fresh breezes, sky]
==>[neptune, loves waves, sea]

1.2.7 Next Steps

This section presented some basic examples of how to traverse *The Graph of the Gods* in Titan. In essence, a graph database is all about representing some world (structure) and traversing it to solve problems (process).

- Learn more about [[Titan’s core interface | Blueprints Interface]]
- Read about choosing a [[Titan storage backend | Storage Backend Overview]]

1.3 Additional Background and Orientation Resources

Here are some external resources with information about Titan, the TinkerPop stack, and the graph data model:

- Introduction to Titan Video
- Brief Primer on Graph Databases
• The Property Graph Model
• Big Graph Data Presentation
2.1 TinkerPop

Titan natively implements the Blueprints Interface which means that it supports all of the open-source technologies in the TinkerPop graph stack:

- **Blueprints** - The property graph model interface implemented by Titan which provides various utilities to aid developers.
- **Gremlin** - A graph traversal language for expressing complex walks through a graph. GremlinDocs is a cheat-sheet for the language.
- **Frames** - An object-to-graph mapper for rendering Java objects from graph data.
- **Rexster** - A graph server for exposing the graph via REST, a binary protocol, and HTML-based GUI tools.

Being a native implementation means that Titan directly implements the Blueprints interface without an adapter. This makes Titan one of the most efficient Blueprints implementations which benefits the performance of all TinkerPop projects when running on Titan.

2.2 Transactions

2.2.1 Transaction Handling

Every graph operation in Titan occurs within the context of a transaction. According to the Blueprints’ specification, each thread opens its own transaction against the graph database with the first operation (i.e. retrieval or mutation) on the graph:

```java
TitanGraph g = TitanFactory.open("/tmp/titan");
Vertex juno = g.addVertex(null); //Automatically opens a new transaction
juno.setProperty("name", "juno");
g.commit(); //Commits transaction
```

In this example, a local Titan graph database is opened. Adding the vertex “juno” is the first operation (in this thread) which automatically opens a new transaction. All subsequent operations occur in the context of that same transaction until the transaction is explicitly stopped or the graph database `shutdown()` which commits all currently running transactions. Note, that both read and write operations occur within the context of a transaction.
Transactional Scope

All graph elements (vertices, edges, and types) are associated with the transactional scope in which they were retrieved or created. Under Blueprint’s default transactional semantics, transactions are automatically created with the first operation on the graph and closed explicitly using `commit()` or `rollback()`. Once the transaction is closed, all graph elements associated with that transaction become stale and unavailable. However, Titan will automatically transition vertices and types into the new transactional scope as shown in this example:

```java
TitanGraph g = TitanFactory.open("/tmp/titan");
Vertex juno = g.addVertex(null); //Automatically opens a new transaction
g.commit(); //Ends transaction
juno.setProperty("name", "juno"); //Vertex is automatically transitioned
```

Edges, on the other hand, are not automatically transitioned and cannot be accessed outside their original transaction. They must be explicitly transitioned.

```java
Edge e = juno.addEdge("knows", g.addVertex(null));
g.commit(); //Ends transaction
e = g.getEdge(e); //Need to refresh edge
e.setProperty("time", 99);
```

Transaction Failures

When committing a transaction, Titan will attempt to persist all changes to the storage backend. This might not always be successful due to IO exceptions, network errors, machine crashes or resource unavailability. Hence, transactions can fail. In fact, transactions will eventually fail in sufficiently large systems. Therefore, we highly recommend that your code expects and accommodates such failures.

```java
try {
    if (g.getVertices("name", name).iterator().hasNext())
        throw new IllegalArgumentException("Username already taken: " + name);
    Vertex user = g.addVertex(null);
    user.setProperty("name", name);
    g.commit();
} catch (TitanException e) {
    //Recover, retry, or return error message
}
```

The example above demonstrates a simplified user signup implementation where `name` is the name of the user who wishes to register. First, it is checked whether a user with that name already exists. If not, a new user vertex is created and the name assigned. Finally, the transaction is committed.

If the transaction fails, a `TitanException` is thrown. There are a variety of reasons why a transaction may fail. Titan differentiates between potentially temporary and permanent failures.

Potentially temporary failures are those related to resource unavailability and IO hiccups (e.g. network timeouts). Titan automatically tries to recover from temporary failures by retrying to persist the transactional state after some delay. The number of retry attempts and the retry delay can be configured through the `Titan graph configuration(Graph Configuration)`.

Permanent failures can be caused by complete connection loss, hardware failure or lock contention. To understand the cause of lock contention, consider the signup example above and suppose a user tries to signup with username “juno”. That username may still be available at the beginning of the transaction but by the time the transaction is committed, another user might have concurrently registered with “juno” as well and that transaction holds the lock on the username therefore causing the other transaction to fail. Depending on the transaction semantics one can recover from a lock contention failure by re-running the entire transaction.

Permanent exceptions that can fail a transaction include:
• PermanentLockingException(*Local lock contention*): Another local thread has already been granted a conflicting lock.

• PermanentLockingException(*Expected value mismatch for X: expected=Y vs actual=Z*): The verification that the value read in this transaction is the same as the one in the datastore after applying for the lock failed. In other words, another transaction modified the value after it had been read and modified.

**Gotchas**

• Transactions are started automatically with the first operation executed against the graph. One does NOT have to start a transaction manually. The method `newTransaction` is used to start ([multi threaded transactions]) only.

• Transactions are automatically started under the Blueprints semantics but *not* automatically terminated. Transactions have to be terminated manually with `g.commit()` if successful or `g.rollback()` if not. Manual termination of transactions is necessary because only the user knows the transactional boundary.

A transaction will attempt to maintain its state from the beginning of the transaction. This might lead to unexpected behavior in multi-threaded applications as illustrated in the following artificial example:

```java
v = g.v(4) //Retrieve vertex, first action automatically starts transaction
v.bothE >> returns nothing, v has no edges
//thread is idle for a few seconds, another thread adds edges to v
v.bothE >> still returns nothing because the transactional state from the beginning is maintained
```

Such unexpected behavior is likely to occur in client-server applications where the server maintains multiple threads to answer client requests. It is therefore important to terminate the transaction after a unit of work (e.g. code snippet, query, etc). For instance, Rexster manages the transactional boundary for each gremlin query. So, the example above should be:

```java
v = g.v(4) //Retrieve vertex, first action automatically starts transaction
v.bothE
//thread is idle for a few seconds, another thread adds edges to v
v.bothE >> returns the newly added edge
```

**Next Steps**

• Read more about Blueprints Transactions

### 2.2.2 Multi-Threaded Transactions

Titan supports multi-threaded transactions through Blueprint’s `ThreadedTransactionalGraph` interface. Hence, to speed up transaction processing and utilize multi-core architectures multiple threads can run concurrently in a single transaction.

With Blueprints’ default transaction handling each thread automatically opens its own transaction against the graph database. To open a thread-independent transaction, use the `newTransaction()` method.

```java
TransactionalGraph tx = g.newTransaction();
Thread[] threads = new Thread[10];
for (int i=0;i<threads.length;i++) {
```
threads[i]=new Thread(new DoSomething(tx));
threads[i].start();
}
for (int i=0;i<threads.length;i++) threads[i].join();
tx.commit();

The `newTransaction()` method returns a new `TransactionalGraph` object that represents this newly opened transaction. The graph object `tx` supports all of the method that the original graph did, but does so without opening new transactions for each thread. This allows us to start multiple threads which all do-something in the same transaction and finally commit the transaction when all threads have completed their work.

Titan relies on optimized concurrent data structures to support hundreds of concurrent threads running efficiently in a single transaction.

**Concurrent Algorithms**

Thread independent transactions started through `newTransaction()` are particularly useful when implementing concurrent graph algorithms. Most traversal or message-passing (ego-centric) like graph algorithms are embarrassingly parallel which means they can be parallelized and executed through multiple threads with little effort. Each of these threads can operate on a single `TransactionalGraph` object returned by `newTransaction` without blocking each other.

**Nested Transactions**

Another use case for thread independent transactions is nested transactions that ought to be independent from the surrounding transaction.

For instance, assume a long running transactional job that has to create a new vertex with a unique name. Since enforcing unique names requires the acquisition of a lock (see [[Type Definition Overview]]) for more detail) and since the transaction is running for a long time, lock congestion and expensive transactional failures are likely.

```java
Vertex v1 = g.addVertex(null);
//Do many other things
Vertex v2 = g.addVertex(null);
v2.setProperty("uniqueName","foo");
g.addEdge(null,v1,v2,"related");
//Do many other things
g.commit(); // Likely to fail due to lock congestion
```

One way around this is to create the vertex in a short, nested thread-independent transaction as demonstrated by the following pseudo code:

```java
Vertex v1 = g.addVertex(null);
//Do many other things
TransactionalGraph tx = g.newTransaction();
Vertex v2 = tx.addVertex(null);
v2.setProperty("uniqueName","foo");
tx.commit();
g.addEdge(null,v1,g.getVertex(v2),"related"); //Need to load v2 into outer transaction
//Do many other things
g.commit(); // Likely to fail due to lock congestion
```
**Gotchas**

When using multi-threaded transactions via `newTransaction` all vertices and edges retrieved or created in the scope of that transaction are *not* available outside the scope of that transaction. Accessing such elements after the transaction has been closed will result in an exception. As demonstrated in the example above, such elements have to be explicitly refreshed in the new transaction using `g.getVertex(existingVertex)` or `g.getEdge(existingEdge)`.

**Next steps**

Read more about Blueprints’s ThreadedTransactionalGraph.

### 2.2.3 Transaction Configuration

Titan’s `TitanGraph.buildTransaction()` method gives the user the ability to configure and start a new *multi-threaded transaction* against a `TitanGraph`. Hence, it is identical to `TitanGraph.newTransaction()` with additional configuration options.

`buildTransaction()` returns a `TransactionBuilder` which allows the following aspects of a transaction to be configured:

- `readOnly()` - makes the transaction read-only and any attempt to modify the graph will result in an exception.
- `enableBatchLoading()` - enables batch-loading for an individual transaction. This setting results in similar efficiencies as the graph-wide setting `storage.batch-loading` due to the disabling of consistency checks and other optimizations. Unlike `storage.batch-loading` this option will not change the behavior of the storage backend.
- `setTimestamp(long)` - Sets the timestamp for this transaction as communicated to the storage backend for persistence. Depending on the storage backend, this setting may be ignored. For eventually consistent backends, this is the timestamp used to resolve write conflicts. If this setting is not explicitly specified, Titan uses the current time.
- `setCacheSize(long size)` - The number of vertices this transaction caches in memory. The larger this number, the more memory a transaction can potentially consume. If this number is too small, a transaction might have to re-fetch data which causes delays in particular for long running transactions.
- `checkInternalVertexExistence()` - Whether this transaction should double-check the existence of vertices during query execution. This can be useful to avoid *phantom vertices* on eventually consistent storage backends. Disabled by default. Enabling this setting can slow down query processing.

Once, the desired configuration options have been specified, the new transaction is started via `start()` which returns a `TitanTransaction`.

### 2.3 Gotchas and Limitations

There are various limitations and “gotchas” that one should be aware of when using Titan. Some of these limitations are necessary design choices and others are issues that will be rectified as Titan development continues. Finally, the last section provides solutions to common issues.
2.3.1 Design Limitations

Size Limitation

Titan can store up to a quintillion edges ($2^{60}$) and half as many vertices. That limitation is imposed by Titan’s id scheme.

DataType Definitions

When declaring the data type of a property key using `dataType(Class)` Titan will enforce that all properties for that key have the declared type, unless that type is `Object.class`. This is an equality type check, meaning that sub-classes will not be allowed. For instance, one cannot declare the data type to be `Number.class` and use `Integer` or `Long`. For efficiency reasons, the type needs to match exactly. Hence, use `Object.class` as the data type for type flexibility. In all other cases, declare the actual data type to benefit from increased performance and type safety.

Edge Retrievals are not O(1)

Retrieving an edge by id, e.g. `tx.getEdge(edge.getId())`, is not a constant time operation. Titan will retrieve an adjacent vertex of the edge to be retrieved and then execute a vertex query to identify the edge. The former is constant time but the latter is potentially linear in the number of edges incident on the vertex with the same edge label. This also applies to index retrievals for edges via a standard or external index.

2.3.2 Temporary Limitations

Key Index Must Be Created Prior to Key Being Used

To index vertices or edges by key, the respective key index must be created before the key is first used in a vertex or edge property. Read more about creating [[vertex indexes|Blueprints Interface]].

Unable to Drop Key Indices

Once an index has been created for a key, it can never be removed.

Types Can Not Be Changed Once Created

This pitfall constrains the graph schema. While the graph schema can be extended, previous declarations cannot be changed.

Batch Loading Speed

Titan provides a batch loading mode that can be enabled through the `configuration <Graph Configuration>`. (TODO link to graph configuration section when populated). However, this batch mode only facilitates faster loading into the storage backend, it does not use storage backend specific batch loading techniques that prepare the data in memory for disk storage. As such, batch loading in Titan is currently slower than batch loading modes provided by single machine databases. The [[Bulk Loading]] documentation lists ways to speed up batch loading in Titan.

Another limitation related to batch loading is the failure to load millions of edges into a single vertex at once or in a short time of period. Such supernode loading can fail for some storage backends. This limitation also applies to dense index entries. For more information, please refer to Issue #11.
2.3.3 Beware

Multiple Titan instances on one machine

Running multiple Titan instances on one machine backed by the same storage backend (distributed or local) requires that each of these instances has a unique configuration for the `storage.machine-id-appendix`. Otherwise, these instances might overwrite each other leading to data corruption. See [[Graph Configuration]] for more information.

Accidental type creation

By default, Titan will automatically create property keys and edge labels when a new type is encountered. It is strongly encouraged that users explicitly [[define types|Type-Definition-Overview]] and disable automatic type creation by setting the `autotype = none`.

Custom Class Datatype

Titan supports arbitrary objects as attribute values on properties. To use a custom class as data type in Titan, either register a custom serializer or ensure that the class has a no-argument constructor and implements the `equals` method because Titan will verify that it can successfully de-/serialize objects of that class. Please read [[Datatype and Attribute Serializer Configuration]] for more information.

Transactional Scope for Edges

Edges should not be accessed outside the scope in which they were originally created or retrieved.

Locking Exceptions

When defining unique [[Type Definition Overview]] with locking enabled (i.e. requesting that Titan ensures uniqueness) it is likely to encounter locking exceptions of the type `PermanentLockingException` under concurrent modifications to the graph.

Such exceptions are to be expected, since Titan cannot know how to recover from a transactional state where an earlier read value has been modified by another transaction since this may invalidate the state of the transaction. It most cases it is sufficient to simply re-run the transaction. If locking exceptions are very frequent, try to analyze and remove the source of congestion.

Double and Float Data Types

Titan internally represents double and float data types as fixed decimal numbers. Doubles are stored with up to 6 decimal digits and floats with up to 3. This representation enables range retrievals in vertex centric queries. However, it significantly limits the precision and range of doubles and floats. Use `FullDouble` and `FullFloat` as data type to get the full precision of floating point numbers. However, note that these data types cannot be used in range-constrained vertex centric queries.

Ghost Vertices

When the same vertex is concurrently removed in one transaction and modified in another, both transactions will successfully commit on eventually consistent storage backends and the vertex will still exist with only the modified properties or edges. This is referred to as a ghost vertex. It is possible to guard against ghost vertices on eventually consistent backends using key `out-uniqueness` but this is prohibitively expensive in most
cases. A more scalable approach is to allow ghost vertices temporarily and clearing them out in regular time intervals, for instance using ‘Titan tools’.

Another option is to detect them at read-time using the [[transaction configuration|Transaction-Configuration]] option checkInternalVertexExistence()

**Snappy 1.4 does not work with Java 1.7**

Cassandra 1.2.x makes use of Snappy 1.4. Titan will not be able to connect to Cassandra if the server is running Java 1.7 and Cassandra 1.2.x (with Snappy 1.4). Be sure to remove the Snappy 1.4 jar in the cassandra/lib directory and replace with a Snappy 1.5 jar version.

**Debug-level Logging**

When the log level is set to debug Titan produces a lot of logging output which is useful to understand how particular queries get compiled, optimized, and executed. However, the output is so large that it will impact the query performance noticeably. Hence, you info or above for production systems or benchmarking.

**Useful Tips**

**Titan OutOfMemoryException or excessive Garbage Collection**

If you experience memory issues or excessive garbage collection while running Titan it is likely that the caches are configured incorrectly. If the caches are too large, the heap may fill up with cache entries. Try reducing the size of the transaction level cache before tuning the database level cache, in particular if you have many concurrent transactions. Read more about [[Titan’s caching layers | Data Caching]].

**Removing JAMM Warning Messages**

When launching Titan with embedded Cassandra, the following warnings may be displayed:

958 [MutationStage:25] WARN org.apache.cassandra.db.Memtable - MemoryMeter uninitialized (jamm not specified as java agent); assuming liveRatio of 10.0. Usually this means cassandra-env.sh disabled jamm because you are using a buggy JRE; upgrade to the Sun JRE instead

Cassandra uses a Java agent called MemoryMeter which allows it to measure the actual memory use of an object, including JVM overhead. To use JAMM (Java Agent for Memory Measurements), the path to the JAMM jar must be specific in the Java javagent parameter when launching the JVM (e.g. -javaagent:path/to/jamm.jar). Rather than modifying titan.sh and adding the javaagent parameter, I prefer to set the JAVA_OPTIONS environment variable with the proper javagent setting:

```
export JAVA_OPTIONS=-javaagent:$TITAN_HOME/lib/jamm-0.2.5.jar
```

**Cassandra Connection Problem**

By default, Titan uses the Astyanax library to connect to Cassandra clusters. On EC2 and Rackspace, it has been reported that Astyanax was unable to establish a connection to the cluster. In those cases, changing the backend to storage.backend=cassandrathrift solved the problem.
ElasticSearch OutOfMemoryException

When numerous clients are connecting to ElasticSearch, it is likely that an OutOfMemoryException occurs. This is not due to a memory issue, but to the OS not allowing more threads to be spawned by the user (the user running ElasticSearch). To circumvent this issue, increase the number of allowed processes to the user running ElasticSearch. For example, increase the ulimit -u from the default 1024 to 10024.
3.1 Storage Backends

3.1.1 Using Cassandra

The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra’s support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages. The largest known Cassandra cluster has over 300 TB of data in over 400 machines.

—Apache Cassandra Homepage

3.2 Deploying on Managed Machines

The following sections outline the various ways in which Titan can be used in concert with Cassandra.

3.2.1 Local Server Mode

Cassandra can be run as a standalone database on the same local host as Titan and the end-user application. In this model, Titan and Cassandra communicate with one another via a localhost socket. Running Titan over Cassandra requires the following setup steps:

1. “Download Cassandra, unpack it, and set filesystem paths in conf/cassandra.yaml and conf/log4j-server.properties

2. Start Cassandra by invoking bin/cassandra -f on the command line in the directory where Cassandra was unpacked. Read output to check that Cassandra started successfully.

Now, you can create a Cassandra TitanGraph as follows:
Configuration conf = new BaseConfiguration();
conf.setProperty("storage.backend","cassandra");
conf.setProperty("storage.hostname","127.0.0.1");
TitanGraph g = TitanFactory.open(conf);

In the Gremlin shell, you can not define the type of the variables conf and g. Therefore, simply leave off the type declaration.

### 3.2.2 Remote Server Mode

When the graph needs to scale beyond the confines of a single machine, then Cassandra and Titan are logically separated into different machines. In this model, the Cassandra cluster maintains the graph representation and any number of Titan instances maintain socket-based read/write access to the Cassandra cluster. The end-user application can directly interact with Titan within the same JVM as Titan.

For example, suppose we have a running Cassandra cluster where one of the machines has the IP address 77.77.77.77, then connecting Titan with the cluster is accomplished as follows (comma separate IP addresses to reference more than one machine):

Configuration conf = new BaseConfiguration();
conf.setProperty("storage.backend","cassandra");
conf.setProperty("storage.hostname","77.77.77.77");
TitanGraph g = TitanFactory.open(conf);

In the Gremlin shell, you can not define the type of the variables conf and g. Therefore, simply leave off the type declaration.

### 3.2.3 Remote Server Mode with Rexster

Rexster can be wrapped around each Titan instance defined in the previous subsection. In this way, the end-user application need not be a Java-based application as it can communicate with Rexster over REST. This type of deployment is great for polyglot architectures where various components written in different languages need to reference and compute on the graph.

http://rexster.titan.machinewithIP/mygraph/vertices/1
http://rexster.titan.machinewithIP/mygraph/tp/gremlin?script=g.v(1).out('follows').out('created')
In this case, each Rexster server would be configured to connect to the Cassandra cluster. The following shows the graph specific fragment of the Rexster configuration. Refer to the “Rexster configuration page” for a complete example.

```xml
<graph>
  <graph-name>mygraph</graph-name>
  <graph-type>com.thinkaurelius.titan.tinkerpop.rexster.TitanGraphConfiguration</graph-type>
  <graph-location></graph-location>
  <graph-read-only>false</graph-read-only>
  <properties>
    <storage.backend>cassandra</storage.backend>
    <storage.hostname>77.77.77.77</storage.hostname>
  </properties>
  <extensions>
    <allows>
      <allow>tp:gremlin</allow>
    </allows>
  </extensions>
</graph>
```

### 3.2.4 Titan Embedded Mode

Finally, Cassandra can be embedded in Titan, which means, that Titan and Cassandra run in the same JVM and communicate via in process calls rather than over the network. This removes the (de)serialization and network protocol overhead and can therefore lead to considerable performance improvements. In this deployment mode, Titan internally starts a cassandra daemon and Titan no longer connects to an existing cluster but is its own cluster.

To use Titan in embedded mode, simply configure `embeddedcassandra` as the storage backend. The configuration options listed below also apply to embedded Cassandra. In creating a Titan cluster, ensure that the individual nodes can discover each other via the Gossip protocol, so setup a Titan-with-Cassandra-embedded cluster much like you would a stand alone Cassandra cluster. When running Titan in embedded mode, the Cassandra yaml file is configured using the additional configuration option `storage.cassandra-config-dir`, which specifies the yaml file as a full url, e.g. `storage.cassandra-config-dir = file:///home/cassandra.yaml`.

When running a cluster with Titan and Cassandra embedded, it is advisable to expose Titan through the Rexster server so that applications can remotely connect to the Titan graph database and execute queries.

Note, that running Titan with Cassandra embedded requires GC tuning. While embedded Cassandra can provide lower latency query answering, its GC behavior under load is less predictable.
3.3 Cassandra Specific Configuration

In addition to the general “Titan Graph Configuration”:Graph-Configuration, there are the following Cassandra specific Titan configuration options:
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Value</th>
<th>Default</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage.hostname</td>
<td>IP address or hostname of the Cassandra cluster node that this Titan instance connects to. Use a list of comma-separated hostnames or IP addresses to seed multiple Cassandra nodes.</td>
<td>IP address or hostname</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>storage.port</td>
<td>Port on which to connect to Cassandra cluster node</td>
<td>Integer</td>
<td>9160</td>
<td>Yes</td>
</tr>
<tr>
<td>storage.connection-timeout</td>
<td>Default time out in milliseconds after which to fail a connection attempt with a Cassandra node</td>
<td>Integer</td>
<td>10000</td>
<td>Yes</td>
</tr>
<tr>
<td>storage.connection-pool-size</td>
<td>Maximum size of the connection pool for connections to the Cassandra cluster</td>
<td>Integer</td>
<td>32</td>
<td>Yes</td>
</tr>
<tr>
<td>storage.read-consistency-level</td>
<td>Cassandra consistency level for read operations</td>
<td>String</td>
<td>QUORUM</td>
<td>Yes</td>
</tr>
<tr>
<td>storage.write-consistency-level</td>
<td>Cassandra consistency level for write operations</td>
<td>String</td>
<td>QUORUM</td>
<td>Yes</td>
</tr>
<tr>
<td>storage.replication-factor</td>
<td>The replication factor to use. The higher the replication factor, the more robust the graph database is to machine failure at the expense of data duplication. <em>The default value should be overwritten for production system to ensure robustness. A value of 3 is recommended.</em> This replication factor can only be set when the keyspace is initially created. <strong>On an existing keyspace, this value is ignored.</strong></td>
<td>Integer</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>storage.cassandra.thrift.frame_size_mb</td>
<td>The maximum frame size to be used by thrift for transport. Increase this value when retrieving very large result sets. <strong>Only applicable when storage.backend=cassandra/thrift</strong></td>
<td>Integer</td>
<td>16</td>
<td>No</td>
</tr>
</tbody>
</table>

3.3. Cassandra Specific Configuration
For more information on Cassandra consistency levels and acceptable values, please refer to the Cassandra Thrift API. In general, higher levels are more consistent and robust but have higher latency.

3.4 Global Graph Operations

Titan over Cassandra supports global vertex and edge iteration. However, note that all these vertices and/or edges will be loaded into memory which can cause OutOfMemoryException. Use Faunus to iterate over all vertices or edges in large graphs.

3.5 Deploying on Amazon EC2

bq. Amazon EC2 is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.

Follow these steps to setup a Cassandra cluster on EC2 and deploy Titan over Cassandra. To follow these instructions, you need an Amazon AWS account with established authentication credentials and some basic knowledge of AWS and EC2.

3.5.1 Setup Cassandra Cluster

These instructions for configuring and launching the DataStax Cassandra Community Edition AMI are based on the DataStax AMI Docs and focus on aspects relevant for a Titan deployment.

3.5.2 Setting up Security Group

- Navigate to the EC2 Console Dashboard, then click on “Security Groups” under “Network & Security”.
- Create a new security group. Click Inbound. Set the “Create a new rule” dropdown menu to “Custom TCP rule”. Add a rule for port 22 from source 0.0.0.0/0. Add a rule for ports 1024-65535 from the security group members. If you don’t want to open all unprivileged ports among security group members, then at least open 7000, 7199, and 9160 among security group members. Tip: the “Source” dropdown will autocomplete security group identifiers once “sg” is typed in the box, so you needn’t have the exact value ready beforehand.

3.5.3 Launch DataStax Cassandra AMI

- “Launch the DataStax AMI in your desired zone
- On the Instance Details page of the Request Instances Wizard, set “Number of Instances” to your desired number of Cassandra nodes. Set “Instance Type” to at least m1.large. We recommend m1.large.
- On the Advanced Instance Options page of the Request Instances Wizard, set the “as text” radio button under “User Data”, then fill this into the text box:

```
--clustername [cassandra-cluster-name]
--totalnodes [number-of-instances]
--version community
--opscenter no
```

[number-of-instances] in this configuration must match the number of EC2 instances configured on the previous wizard page. [cassandra-cluster-name] can be any string used for identification. For example:
--clustername titan
--totalnodes 4
--version community
--opscenter no

- On the Tags page of the Request Instances Wizard you can apply any desired configurations. These tags exist only at the EC2 administrative level and have no effect on the Cassandra daemons’ configuration or operation.
- On the Create Key Pair page of the Request Instances Wizard, either select an existing key pair or create a new one. The PEM file containing the private half of the selected key pair will be required to connect to these instances.
- On the Configure Firewall page of the Request Instances Wizard, select the security group created earlier.
- Review and launch instances on the final wizard page.

3.5.4 Verify Successful Instance Launch

- SSH into any Cassandra instance node:

  ```
  ssh -i [your-private-key].pem
  ubuntu@[public-dns-name-of-any-cassandra-instance]
  ```

- Run the Cassandra nodetool:

  ```
  nodetool -h 127.0.0.1 ring
  ```

  You should see as many nodes in this command’s output as instances launched in the previous steps.

Note, that the AMI takes a few minutes to configure each instance. A shell prompt will appear upon successful configuration when you SSH into the instance.

3.5.5 Launch Titan Instances

Launch additional EC2 instances to run Titan which are either configured in Remote Server Mode or Remote Server Mode with Rexster as described above. You only need to note the IP address of one of the Cassandra cluster instances and configure it as the host name. The particular EC2 instance to run and the particular configuration depends on your use case.

3.5.6 Example Titan Instance on Amazon Linux AMI

- Launch the Amazon Linux AMI in the same zone of the Cassandra cluster. Choose your desired EC2 instance type depending on the amount of resources you need. Use the default configuration options and select the same Key Pair and Security Group as for the Cassandra cluster configured in the previous step.
- SSH into the newly created instance via:

  ```
  ssh -i [your-private-key].pem
  ec2-user@[public-dns-name-of-the-instance].
  ```

  You may have to wait a little for the instance to launch.
- Download the current Titan distribution with ```wget``` and unpack the archive locally to the home directory. Start the gremlin shell to verify that Titan runs successfully. For more information on how to unpack Titan and start the gremlin shell, please refer to the “Getting Started guide”:Getting-Started.
- Create a configuration file with ```vi titan.properties``` and add the following lines:

  ```
  storage.backend = cassandra
  storage.hostname = [IP-address-of-one-Cassandra-EC2-instance]
  ```

You may add additional configuration options found on this page or under “Graph Configuration”:Graph-Configuration.
• Start the gremlin shell again and type the following:

```java
gremlin> g = TitanFactory.open('titan.properties')
==>
titangraph[cassandra:[IP-address-of-one-Cassandra-EC2-instance]]
```

You have successfully connected this Titan instance to the Cassandra cluster and can start to operate on the graph.

### 3.5.7 Connect to Cassandra cluster in EC2 from outside EC2

Opening the usual Cassandra ports (9160, 7000, 7199) in the security group is not enough, because the Cassandra nodes by default broadcast their ec2-internal IPs, and not their public-facing IPs.

The resulting behavior is that you can open a Titan graph on the cluster by connecting to port 9160 on any Cassandra node, but all requests to that graph time out. This is because Cassandra is telling the client to connect to an unreachable IP.

To fix this, set the “broadcast-address” property for each instance in `/etc/cassandra/cassandra.yaml` to its public-facing IP, and restart the instance. Do this for all nodes in the cluster. Once the cluster comes back, nodetool reports the correct public-facing IPs to which connections from the local machine are allowed.

Changing the “broadcast-address” property allows you to connect to the cluster from outside ec2, but it might also mean that traffic originating within ec2 will have to round-trip to the internet and back before it gets to the cluster. So, this approach is only useful for development and testing.

**TODO Sections from Github Wiki**

- [[Using HBase]]  
- [[Using Persistit]]  
- [[Using BerkeleyDB]]  
- **Indexing Backends**  
- [[Indexing Backend Overview]]  
- [[Using Elastic Search]]  
- [[Using Lucene]]  
- [[Full Text and String Search]]  
- [[Direct Index Query]]  
- **Type Management**  
- [[Type Definition Overview]] (*cheat sheet*)  
- [[Vertex-Centric Indices]]  
- **Configuration and Tuning**  
- [[Graph Configuration]] (*cheat sheet*)  
- [[Datatime and Attribute Serializer Configuration]]  
- [[Example Graph Configuration]] (*cheat sheet*)  
- **[Data Caching]]  
- [[Database Cache]]  
- **[Transaction Cache]]