Clustered TDB: A Clustered Triple Store for Jena

Alisdair Owens
IAM Group
Electronics and Computer Science
University of Southampton, UK
+44 (0)23 8059 8367
ao@ecs.soton.ac.uk

Andy Seaborne
HP Labs Bristol
Stoke Gifford
Bristol, UK
+44 (0)117 312 8181
andy_seaborne@hp.com

Nick Gibbins
IAM Group
Electronics and Computer Science
University of Southampton, UK
+44 (0)23 8059 8879
nmg@ecs.soton.ac.uk

mc schraefel
IAM Group
Electronics and Computer Science
University of Southampton, UK
+44 (0)23 8059 8372
mc@ecs.soton.ac.uk

ABSTRACT

This paper describes the design of Clustered TDB, a clustered triple store designed to store and query very large quantities of Resource Description Framework (RDF) data. It presents an evaluation of an initial prototype, showing that Clustered TDB offers excellent scaling characteristics with respect to load times and query throughput. Design decisions are justified in the context of a literature review on Database Management System (DBMS) and RDF store clustering, and it is shown that many techniques created during the course of DBMS research are applicable to the problem of storing RDF data.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed databases.*

H.2.2 [**Database Management**]: Physical Design – *access methods.*

H.2.4 [**Database Management**]: Systems – *distributed databases, concurrency.*

General Terms

Algorithms, Performance, Design, Reliability.

Keywords

RDF, Semantic Web, DBMS, Cluster, Triple Store, Distributed

1. Introduction

RDF stores suffer from well documented issues with both read and write performance [1, 17]. The semi-structured nature of RDF makes it ideal for applications where the structure of data added to a store is not well known in advance, is liable to change rapidly, or where there are many different structures being linked together. Unfortunately, this flexibility makes it challenging to design a high performance catch-all schema to describe the data, and results in database schemas featuring long, thin tables with very large index depths, and a requirement for a comprehensive indexing strategy. This approach results in limited performance and poor characteristics when scaling to larger datasets.

Since RDF is a key language for the Semantic Web, and is used as flexible language for data exchange in both business and large scale science (for example, the UNIPROT project[1]), it can be expected that it will be necessary to store and query very large volumes of RDF data, and stores featuring improved performance are thus highly desirable. The most powerful single machine triple stores are currently capable of storing up to around two billion triples[2], and to realise very large improvements upon this using current technologies it is necessary to allow RDF stores to make use of the power of multiple machines.

Traditional Database Management Systems (DBMSs) underwent a similar evolution, as ever-increasing dataset sizes required the development of DBMSs with better scaling characteristics. Modern databases are often clustered over more than one machine, in an effort to make use of their combined power. RDF stores are a type of DBMS, and research into prior systems can be applied to the creation of a highly scalable RDF store. This document presents the Jena Clustered Tuple Database (Clustered TDB), a clustered RDF store created using techniques used in highly scalable relational DBMSs. The paper includes evaluations of a prototype supporting future work, extended from the single machine Jena Tuple Database (TDB) described in section 4.

Our focus in this paper is the creation of a system that is clustered, that is, the latency between machines is expected to be low due to their being sited in one geographical location, and the system as a whole can be administered from a single point: there is no need to explicitly aggregate content from heterogeneous database systems, nor expectation that those individual machines will be able to provide meaningful answers to queries run on them rather than the system as a whole. This paper describes the creation of a high performance, scalable storage layer: while distributed query optimisation is a topic of great importance to this work, it is largely beyond the scope of this paper.

This paper contains a literature review on distributed DBMSs in section 2, relating it to existing clustered triple stores in section 3. It is expected that this review will aid in the design of future clustered stores. Section 4 describes the single machine Jena

Copyright is held by the author/owner(s).
WWW 2009, April 20-24, 2009, Madrid, Spain.

[1] http://dev.isb-sib.ch/projects/uniprot-rdf/

[2] http://esw.w3.org/topic/LargeTripleStores

Tuple Database (TDB) from which an evaluation prototype was extended. Section 5 details the design for Clustered TDB, and the prototype of this design is evaluated for scaling characteristics in section 6.

2. Distributed DBMS

When the workload on a given database becomes too large (whether this be a result of data size or query load), a traditional approach in the DBMS world is to split the database across more than one system. It is hoped that the power of multiple machines can thus be leveraged to work on the same problem. This section describes the background information on distributed DBMSs that informed the development of Clustered TDB, as described in section 5.

The desired performance improvements in distributed DBMSs can be categorised as follows [3, 8]:

- **Scaleup**: An increase in the number of machines leads to the ability to store more data.
- **Speedup**: An increase in the number of machines leads to a reduction in the amount of time taken to serve an individual query, all other factors being equal.
- **Throughput Scaleup**: An increase in the number of machines leads to the ability to perform more transactions in a given time frame.

While ideally both speedup and scaleup will be linear with the amount of processing power available, this is a practical impossibility in any database system: some algorithms (such as sort) do not scale in linear time. There are other significant barriers to such a perfect level of system scalability[8]:

- **Startup**: the time needed to start a parallel operation - if a small operation results in lots of processes being started across a lot of nodes, the cost of startup can overwhelm any advantages gained through increased parallelism.
- **Interference**: The slowdown each new process creates when accessing shared resources.
- **Skew**: The effect where one part of a parallelised operation takes much longer to complete than the others: since the job is limited by the slowest process, this can seriously affect performance.

Each of these factors can be mitigated by various mechanisms: the system should be arranged such that small operations are parallelised to a degree commensurate with their size, reducing the influence of startup time. Interference can be reduced by minimising the amount of resource sharing required in the system, and skew by taking steps to divide workloads at 'hotspots' in the system.

2.1 Hardware Architectures

A variety of hardware architectures have been utilised to create parallel database systems. These can be broadly grouped into three categories: shared memory (SM), shared disk (SD) and shared nothing (SN) [18]. In SM systems all processors share a common central memory, in SD they have a private memory but a common collection of disks, and in SN they share only the ability to communicate with each other via messages over a network.

Generally speaking, shared nothing systems are favoured today for their excellent characteristics with regards to resource contention: the only shared resource is network access, and there is no need for the complex resource locking methods seen in SM and SD systems. This means that scaling up SN clusters has historically been easier than the alternatives [8, 18]. Further, SN clusters can be built out of commodity parts, as seen in companies like Google, offering an excellent price/performance profile.

The disadvantage of the SN approach is that there is greater complexity in deciding where data is placed: it is important to place data such that each machine undergoes a similar load profile to enable efficient scaling, and does not require excessive use of network resources. Ongoing maintenance (whether manual or automatic) to the distribution of data is necessary to prevent 'hot spots', or points at which data or query skew has caused a machine to have too high a workload. When these hot spots occur, they can usually be eliminated by redistribution of data on the machine.

2.2 Enabling Parallelism

Parallel execution can be enabled through a variety of strategies. Most obviously, it is possible to partition (or decluster) information across more than one machine, such that the time required to retrieve a large block of data is reduced, and the number of users who can retrieve data at any one time (assuming they are not both trying to access the same data) is also increased. It should be noted that typically, when reading or writing very small amounts of data, it is desirable to perform the work on one machine. This is because the setup costs will dwarf any advantages gained from partitioning. The 'Data Partitioning' section considers the problem of how to decluster data in more detail.

Another way of parallelising database systems is to cluster the execution of relational operations, so that for a given operation each machine processes a defined range of data values out of an overall dataset. This prevents one machine from doing all the processing work and becoming a bottleneck.

Pipelining of operations is another way to parallelise: many relational operators do not need to complete before they start emitting results. In this sense they can be viewed as a stream. The output of this stream can be directed to other operations, which can start processing them in parallel with the first operation. The benefits of this approach are somewhat limited, however: firstly, pipelines are usually relatively short, limiting the number of machines that can work on one, secondly, some relational operations (such as sort) do not emit results until they complete, and thirdly, some operations take much longer than others (an example of skew), thus causing some machines to have to undertake much more work than others.

Finally, parallelism is supported by simply allowing multiple users to access a system, and allowing the subqueries that form an individual query to run in parallel. This is enabled by the likelihood that different users and subqueries will likely be accessing different pieces of information, so hardware resources can be shared between them and the queries run in parallel.

These mechanisms for enabling parallelism can be characterised as occuring at three levels [14]:

- **inter-query**: The ability to run more than one query simultaneously.
- **intra-query**: The ability to run different subqueries in parallel and pipeline operations.
- **intra-operation**: Distributing single operations over more than one node for concurrent execution.

2.3 Data Partitioning

A standard approach to partitioning data in an RDBMS is horizontally partitioning (or declustering) each relation in the system. In these systems, tuples of each relation in the database are partitioned across the disk storage units attached to each processing node on the network, allowing multiple machines to scan a relation in parallel. It also addresses hotspot issues, as the contents of regularly accessed relations are spread across multiple machines, and more can be added as necessary.

[8] describes methods for horizontal partitioning of data, dividing them into three common techniques:

- **Round Robin**: simply distributing the tuples in a round robin fashion to each server. This approach works well for sequential scans, but is inefficient if there is a desire to access tuples based on attribute values, since the location of a given tuple is unknown.
- **Hash Partitioning**: distribution of tuples by applying a hash function to an attribute value. The function emits a number which specifies a machine (and possibly disk location) on which to store the information. This approach is effective if tuples are accessed based on a fully specified attribute, but is much less effective for range queries: hashing does not do a good job of clustering related data. Further, hash partitioning suffers from difficulties with the addition of new machines to a cluster, and addressing hot spots: in a naive implementation it is not possible to repartition data.
- **Range Partitioning**: distribution of tuples by selecting a range over one attribute. For example, all tuples with a value of 'surname' between A-C go on one partition, D-E on another, and so on. This approach clusters data effectively. The major issue with this is that it risks both data and data and execution skew: one part of the range may have a disproportionately large quantity of the actual data, and one part of the cluster may get accessed much more frequently than others (this being particularly likely if it has to store more of the data).

Partitioning improves the response time of sequential scans, because more processors and disks are used to perform the scan. It aids associative scans (scanning based on an attribute value) because the number of tuples stored at each node is reduced, and hence index sizes are reduced.

It is important to decluster data in a manner appropriate to both the dataset itself, and the manner in which it will be accessed. In particular, the following factors have a significant influence:

- **Degree of declustering:** it is important to decluster to an appropriate extent. If a very small relation is partitioned over a very large number of machines, startup costs and overheads (such as disk seeks) will overwhelm any advantages gained from parallelism. In practise, parallel systems such as Bubba[2, 3] have found that full declustering is often inappropriate.
- **Skew:** It is important to ensure that each machine undergoes a comparable workload. A simple implementation will balance the quantity of information stored on each server, but it is also important to take into account the possibility that certain data ranges will be accessed much more regularly than others, creating an excessive load on some servers. This type of skew (*execution* skew) can be countered by balancing data distribution not by the amount of volume stored on each machine in the cluster, but by the frequency with which each machine has to access data, particularly that which is uncached.
- **Declustering attribute:** it is necessary to partition on an appropriate attribute: the location of tuples is only known, if it is known at all, based on a function of that attribute. Queries that reference a relation based on a different attribute value have to be flooded to all machines that store a portion of the relevant relation[13]. This presents no barriers in a store with comprehensive indexing such as TDB, since each index can be distributed based on its primary attribute, but is of interest when considering other strategies.

2.4 Parallel Operations

Parallelising relational operations can be quite a simple process, requiring the addition of two simple operations:

- **merge**: If one considers a scan of a relation that has been distributed into N partitions, a scan of this relation can be implemented as N scan operations that then send their output to a common merge operator. This produces a single output stream that can be used by the next relational operator.
- **split**: Split is used to partition an output stream produced by a relational operator, such that each sub-stream can be processed by a different machine.

With the aid of these two operations, all that is required is to decide how many machines will process a given piece of data, and, using a common split function, partition data processing across these nodes. This can be performed using partitioning methods such as those described in the 'Data Partitioning' section.

Performing operations in parallel requires the transmission of data over the network. This is usually not a significant issue in very small clusters, but as with growth can become a significant bottleneck. Typical network structures offer low/no contention when communicating between machines connected to the same switch, but (assuming a star or hybrid mesh/star topography) suffer from significantly higher contention when communicating across multiple switches (machines that are 'further away'). As a result of this, modern massively distributed file systems such as the Google File System (GFS) and the Hadoop Distributed File System (HDFS) make an effort to site related data in similar areas of the network [4].

A further result of both limited network bandwidth and the desire to avoid unnecessary latency is the observation that it is usually cheaper to move computation to the node where data is situated than to move the data to a specified computation node. Systems like Hadoop[4] and MapReduce[7] follow the example of distributed DBMSs[13] in making an effort to schedule processing at or near the node that stores the relevant chunk of data.

2.5 Redundancy

On single machine or small cluster systems, the likelhiood of machine failure is very low, and there is relatively little requirement for redundancy except in critical systems. As clusters expand to tens, hundreds or thousands of machines, likelihood of machine failure becomes nontrivial[7]. It therefore becomes important to have a strategy for dealing with these failures.

A simple strategy for redundancy is mirroring servers. This not only provides increased data security, but also improves performance by allowing two machines to answer a given request. Unfortunately, using this approach machine failure still has a significant effect upon performance: failure of one machine results in a huge increase in load upon its mirror(s), and the creation of a hot spot. A better strategy for handling redundancy is to distribute data using more than one function. This results in the data on any individual machine being mirrored across many other servers in the network. In this scenario, machine failure results in the load that machine was undertaking being spread across the rest of the cluster, rather than one or a few machines.

3. Clustered RDF Stores

This section describes existing clustered RDF stores, in particular YARS2 and Virtuoso. Federated stores are not considered in this review, since their objectives are different to our own.

3.1 Virtuoso

Virtuoso's RDF component is a quad store based on an Object-Relational DBMS heavily optimised for RDF storage. The recently released clustered variant uses a traditional hash partitioning scheme to split its data, except that indexes are partitioned across machines, as well as data. This makes sense for an RDF store, where the size of indexes can easily overwhelm the size of the data itself.

Virtuoso's creators[9] emphasise the point that a web scale system needs to have a means for repartioning data without causing downtime. As noted in 2.3, hash distribution does not provide an inherent mechanism for rebalancing: a hash by its very nature forces a piece of data to a single given point. Virtuoso uses a common system whereby one pretends that there are (for example) 50n machines in an n machine cluster. In this example, each machine is initially responsible for 50 of the virtual machines. Rebalancing can then be accomplished by moving responsibility (and relevant data) for certain virtual machines from one physical machine to another. Rebalancing is a time consuming process, but one that can be performed on-the-fly.

Virtuoso performs query optimisation without the aid of statistics: the authors note that traditional SQL statistics are of little use for triple or quad stores, and that in order to most effectively optimise such a store it is necessary to have access to a large quantity of statistical information. Virtuoso performs optimisation not by precalculating statistics, but by sampling the data directly and performing estimates on the fly.

3.2 YARS2

YARS2[11] is a heavily read optimised federated repository, using six different indexes into six data orderings (plus an inverted index of text), supporting full retrieval of RDF quads. The index type used is called a 'sparse' index, which is an in memory index into a sorted and blocked data file. To retrieve data, a binary search is performed upon the index, and the closest block of data is retrieved. To enable it to stay in memory, the index gets less specific as the dataset gets larger. This results in near-constant retrieval time with respect to index size, as disk seeks are minimised, and the major cost is the disk seek rather than the amount of data retrieved.

YARS2 uses a hash partitioning over the first attribute of the quad to distribute its indexes. This mechanism can keep closely related data clustered on a single machine (which reduces the amount of time-consuming communication between machines), but can be disadvantageous when considering data orderings that are predicate-first. The solution used by YARS2 is to randomly distribute predicate-first orderings, and flood queries that require this ordering to all machines. It is not clear how the hash function will continue to work with addition or removal of machines, although it can be assumed that a mechanisms such as virtual servers or consistent hashing are used to ensure that there is not an excessive amount of data reorganisation required when machines are added to or removed from the network.

3.3 Summary

Existing clustered triple stores already implement some of the techniques described in section 2. Hash partitioning is used in both Virtuoso Cluster and YARS2, and makes sense for RDF storage systems, where the lexical values of URIs have no bearing on their meaning. Rebalancing is offered in Virtuoso, but is a time consuming process, and neither store offers an obvious solution for the problem of high-cardinality properties. Both stores, however, have been successful in scaling to larger datasets than standalone systems, showing the value of shared-nothing clusters.

4. The Jena Tuple Database

The standalone TDB system is a single-machine graph persistence mechanism for the Jena Semantic Web framework [5]. Jena provides an extension point (the Graph interface) that allows different storage implementations to be used with the common Jena APIs for RDF, ontologies and SPARQL query.

The first Jena Graph implementation was an SQL-backed system [19] optimized for API access. A denormalized design stored RDF terms directly in the triple table, so that matching a single triple pattern required only a single partial scan of the triple table. SPARQL [16] introduces a standard way to ask queries involving more than a single triple pattern, based on the matching of Basic Graph Patterns [16]. A normalized design, where triples are stored with fixed length, short identifiers mapped to RDF terms by a separate table is more efficient for SPARQL query access, since the identifiers are smaller and much quicker to join on.

TDB's design goals are to provide the storage layer for both a single machine usage and also distributed clusters of industry standard servers, as found in enterprise datacentres. TDB exploits modern operating system features, primarily memory mapped I/O on 64 bit hardware rather than relying on its own caching algorithms. TDB does not provide database-style ACID transactions. Other variations of the basic TDB design exist (for example transactions and indexing variations), but are not covered in this paper. An independent review of standalone TDB's performance can be found in [6].

4.1 TDB Design Overview

To represent the RDF graph internally, TDB holds three composite indexes in the form of B+ trees: Subject-Predicate-Object (SPO), Predicate-Object-Subject (POS), Object-Subject-Predicate (OSP). There is no "triple table" because each composite index contains all three fields. The choice of a comprehensive indexing strategy is made to avoid any full table scans when performing a triple match, at the cost of increased load time.

RDF terms (henceforth called "nodes") are represented internally in TDB by 64 bit node identifiers. RDF triples are stored in the three triple indexes as a series of three of these identifiers, or

NodeIDs. Each NodeID is a unique reference into the node table, created during the load process. The NodeID itself is, under normal circumstances, a disk address for retrieving a node serialization. The placement of the disk address directly in the NodeID has desirable consequences: it allows the node table to be written to using simple, fast appending writes, and removes the requirement for an index over the node table. This elimates an index lookup on the critical path of NodeID to node conversion.

In order to allow conversion of queries into canonical form, it is necessary to allow nodes to be converted into NodeIDs. Hence, a small index from node to NodeID (henceforth called the Node/NodeID index) is maintained that maps a 64 bit hash of each node to its NodeID [10].

Since it is desirable to eliminate expensive NodeID to Node conversions where possible, NodeIDs can directly encode (or *inline*) literals of certain datatypes. NodeIDs are comprised of 8 bits of type information, and 56 bits of disk address, which allows literals that can be encoded in 56 bits or less to be inlined. XML Schema Datatypes integer (and derived types), decimal, datetime, date and boolean are encoded directly into the 56 bit section if possible. For example, an XSD dateTime with millisecond resolution can be encoded over a range of 8000 years, including the timezone. RDF Literals whose values are outside the encoded range are stored in the node table, as are RDF literals with illegal lexical forms for the datatype. A consequence of storing values rather than lexical forms is that TDB does not preserve the difference between integers "1" and "01", nor between xsd:byte and xsd:int. This is permitted as D-entailment [12].

4.2 Query processing of Basic Graph Patterns

The SPARQL algebra is built on matching basic graph patterns. TDB evaluates filtered basic graph patterns, that is, filter expressions applied to a basic graph pattern matched against the stored RDF. The rest of the SPARQL algebra is handled by ARQ (Jena's query system). The procedure substitutes any known values for variables, and optimizes the evaluation order of the pattern. This is performed by choosing the triple pattern that is expected to return the fewest number of solutions, based on statistics provided by the graph (most importantly, the distribution of predicates in the data). All variables that this first triple pattern will return are marked as known, and the process is applied to the remainder of the basic graph pattern. Once an execution order for a basic graph pattern has been decided, any filters are placed at the first point at which all variables in the expression would have become bound. This execution plan is then evaluated to yield a stream of results.

Matching a triple pattern is performed by choosing the index that most closely aligns to the constants of the triple pattern: a triple pattern with a known S and P, for example, will use the SPO index. Next, a range scan of the index is performed to find the NodeIDs for the unknown parts of the pattern. The NodeID is only converted into an RDF term when it is needed in a filter expression or the application accesses that solution binding.

5. Design

TDB has a simple, extensible design that yields performance improvements over traditional triple stores, particularly in the area of read/write performance to the node table. Given these desirable characteristics, it was decided to use the techniques described in section 2 to extend it into a cluster store.

This section describes the design of the store, and in particular how the store preserves the benefits of the existing TDB system. The key points of interest in this design are:

- Application of existing DBMS clustering techniques to the problem of RDF storage.
- The mechanism by which the data is distributed - particularly, avoidance of skew.
- Extension of TDB's NodeID system: since standalone TDB's NodeIDs reference a location in a file, these will not be unique in a clustered system. It was necessary to adjust this system such that NodeIDs referenced a unique location on the network, while still retaining TDB's fast appending writes to the node table.

5.1 Application of Clusters to RDF Storage

The process of distributing RDF stores is not fundamentally different to distributing relational DBMSs: the techniques described in section 2 are all applicable to RDF storage. This is unsurprising given that RDF can be effectively represented on relational DBMSs. There are, however, a variety of differences between distributing RDF data and usual distributed relational schemas.

The relatively unstructured nature of RDF does not lend itself to storage in anything but the most broad of data structures, and RDF stores such as TDB are effectively indexing a very long, thin single table of data, which is then repeatedly joined to itself to answer queries. This makes query planning more difficult: traditional SQL optimisers are expected to work on normalised multi-table layouts, where a relatively small amount of table-level statistics are often sufficient to inform optimisation. While there is little fundamental difference between optimising queries for SQL and SPARQL [9], the statistics held by traditional SQL optimisers may not provide the requisite level of detail for dealing with RDF. In particular, since SPARQL queries often perform joins over large quantities of data, it is important to be able to calculate a rough value for how many matches will be found when performing a given subquery: the consequences of a massively increased working set can be disastrous. This requires either the ability to sample data to generate statistics at runtime [9], or the maintenance of a large body of statistical data.

Since triple stores employ heavy indexing to provide adequate read performance, the index size of an RDF database often dominates the size of the data itself. It is thus necessary to distribute indexes across the cluster along with the data itself. In a store with covering indexes, the problem of performing attribute scans on a non-indexed attribute disappears, since an index will be distributed on each attribute. Further, the indexes are not required to answer range queries. Since URIs reference discrete concepts, it makes little sense to perform a restricted range query over them: it is usually only necessary to retrieve either one or all of a particular attribute. This makes hash-based partitioning more attractive.

SPARQL queries are usually of an analytical nature: that is, it is rare for an operation to request or update a single record in an RDF database, in contrast to On-Line Transaction Processing (OLTP) systems. The workload is somewhat closer to On-Line Analytical Processing (OLAP), except that there are few enough columns in the schema to maintain a covering index, and there may be an expectation that there are ongoing rather than simply bulk updates to the store.

5.2 Overall System Structure

Figure 1 shows an example network topology for clustered TDB. Within this diagram, there are two types of machine:

- **Query Coordinator (QC):** Query coordinators are responsible for receiving queries, transforming them into a canonical form (including transformation of URIs and literals to NodeIDs), producing a query plan, and controlling execution on the data nodes. Query coordinators store the distributed Node/NodeID mapping table, and any relevant statistical information.
- **Data Node (DN)**: Data nodes are responsible for storing the node table and triple indexes, extracting data as required from them, and performing operations such as sorts, joins, and so on.

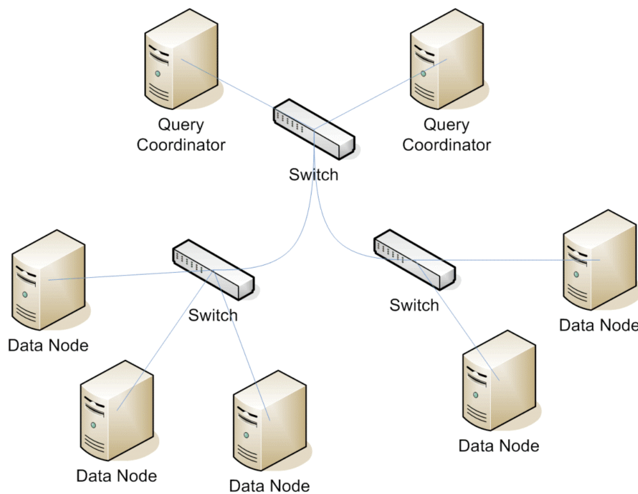

Figure 1. Clustered TDB Network Topology

The rationale behind choosing a layout like this is that it is expected that it will be necessary to maintain a large amount of statistical data and Node/NodeID mappings. Sharing this between as few machines as possible reduces latencies since QCs can be closely colocated, and it is more likely that a given machine will already have a piece of data.

5.3 Balancing and Fault Tolerance

Allowing rebalancing is a fundamental requirement for this application. Clustered TDB uses a traditional technique for allowing easy rebalancing, regardless of distribution method: Clustered TDB deals with virtual processing nodes (or vnodes). A given machine can be responsible for a quantity of vnodes, and a registry of where each is located is stored on every machine. When it is necessary to rebalance, a vnode's files can simply be moved from one server to another, and the registry updated.

Each of the distributed node table, Node/NodeID mapping index, and triple indexes has a different vnode space. This allows a greater degree of flexibility in distributing data and eliminating skew. The use of virtual processing nodes enables a simple solution for redundancy and fault tolerance: mirroring of vnodes. This offers a finer grain of replication than simple server cloning: to prevent the inefficiencies of machine-level cloning, mirrored vnodes can be distributed in a different manner to the primary copy.

5.4 NodeIDs

NodeIDs in TDB are comprised of a byte of type information, used for storing certain literal values inline, and 7 bytes of index into the node table file. These IDs are not suitable for a clustered store without modification, since the cluster will have multiple node files, and the NodeID will thus not encode a known unique location. The solution to this is an additional field: the vnode id in which the node is stored. This is illustrated in Figure 2:

Figure 2. NodeID composition

This scheme allows nodes to be unique, yet still act as a direct index into a node file. Future ID schemes may use a single bit to indicate whether the ID is an inline literal or not, combined with a smaller index into the file. This would allow a reduction in NodeID size, and hence smaller, more cache friendly indexes. This change would break code compatibility with standalone TDB, however.

5.5 Distributing the Node/NodeID Index

The proposed mechanism for partitioning the Node/NodeID mapping index is quite traditional, using a simple hash distribution scheme. Each entry in the index is comprised of the hash of a node and a NodeID. A vnode is decided for the entry based on the node hash modulated by the number of vnodes.

Hash distribution is the obvious choice for this case: the attribute that is being partitioned on is already a hash value, so any sense of data clustering has already been removed, eliminating the major justification for range partitioning. Hash partitioning should give an even distribution of information, and any hotspots can be eliminated using vnode rebalancing.

5.6 Distributing the Node Table

The node table has a less conventional distribution method. When data is being asserted, the Query Coordinator decides a vnode to send nodes to based on any given distribution mechanism, and sends the nodes to that vnode. The DN hosting the vnode then generates IDs for the nodes, returns them to the QC, and stores them in a node table.

A point of particular interest here is that the DN is not obliged to use the vnode that the QC decided on: it can add the node to any one of the vnodes it hosts. If it were limited to the vnode chosen by the QC, then a DN servicing 20 vnodes would be writing to 20 node tables simultaneously, reducing the benefit of TDB's appending node table. Instead, the DN can decide which vnode to write node information to, write blocks, and occasionally switch to another vnode on a round-robin basis. This produces balanced data, and allows a DN to write to one node table at a time.

Redistribution of the node table is a simple operation: the node table file related to the relevant vnode is simply copied to another server, the vnode tables on all the servers updated, and the file deleted on the original server.

5.7 Distributing Triple Indexes

Triple indexes are distributed in a conventional manner. The Query Coordinator distributes each triple three times, based on

hashes of S, P, and O. These three distributions correspond to the three indexes, SPO, POS, and OSP respectively. When a Data Node receives a triple, it stores it only in the index upon which it was distributed: that is, a triple distributed on S is inserted only into the SPO index, and so on. The effect of this is that the data in indexes on each machine is inconsistent: each machine will have different triple data each of its indexes. This means that Data Nodes cannot be queried meaningfully as an entity independent of the cluster.

5.8 Exceptions

While the described distribution methods are simple and robust, they do break down in certain cases. Most particularly, certain properties, such as rdfs:label, have extremely high cardinalities. A hash distribution scheme, even with vnode balancing, provides no mechanism for balancing this out. This leads to the creation of a large hot spot on whatever machine has to store the vnode containing all the data on rdfs:label.

YARS2[11] worked around this problem by simply flooding property-ordered triples to all servers, and flooding property-oriented subqueries to all servers. While this works, it creates a large unnecessarily large load when answering property-centric queries - particularly for properties of low cardinality, which would ideally be stored on a single server.

Clustered TDB's solution to this is an exception list for each index. These lists, replicated across all servers, allow data related to a given NodeID to be partitioned by its first two attributes (optionally across a list of specified vnodes), rather than just the primary attribute. Queries specifying, for example, PO can still be answered on a single server, while queries with just a P are distributed across all (or several) of the servers.

The exception list is expected to be short, and used only when absolutely required, since its contents are mirrored on every server. However, it will prove invaluable for these niche cases.

5.9 Operations

Clustered TDB is expected to use the standard techniques for parallelising operations: pipelining and partitioning, as described in sections 2.2 and 2.3. The mechanism for distributing operations fairly must produce a split function that provides an equitable division of labour. This can be accomplished by adding load information to the heartbeat messages that let each machine in the cluster know that other machines are alive, allowing Query Coordinators to determine servers that are experiencing the least load and use them.

6. Evaluation

To test the design, a prototype was produced that implemented a distributed node table and triple indexes, queried by a non-distributed Query Coordinator. No attempt has been made at this stage to compare the design to other clustered stores, since the code has not been optimised, and there is no query optimiser in the system: the focus of this evaluation is validation of the scaling characteristics of Clustered TDB's storage layer.

The following tests were performed on standalone TDB, and Clustered TDB running on 1, 2, and 3 machines:

- Load rates
- Individual triple pattern, without node retrieval
- Individual triple pattern, with node retrieval
- Join tests, without node retrieval

The tests were performed over a synthetically generated dataset summing approximately 375 million triples, and containing approximately 4,000 distinct properties. Since the conversion of NodeIDs to Nodes (node retrieval) can easily dominate other costs in queries that return even moderate numbers of results, this step is not always performed: this should provide greater insight into Clustered TDB's performance. Theoretically, both large joins and node retrieval should offer useful opportunities for parallelisation, as they require a large number of subqueries.

6.1 Test Configuration

The hardware configuration for these machines was a cluster of three identical systems. Each of these was specced as follows:

- Quad AMD 880 processors, total 8 cores running at 2.4GHz (64 bit)
- 32 GB RAM
- 2x 140GB, 10,000RPM disk drives, in RAID 0 configuration
- Gigabit Ethernet
- Ubuntu Linux, running kernel 2.6.24

Machine one ran both the QC and a DN. Machines two and three (when required) each ran a DN. Prior to load and the start of the read tests, each machine had its disk cache flushed to ensure results were not prejudiced by earlier activities. Commands used were as follows:

```
sync
echo 3 > /proc/sys/vm/drop_caches
```

Prior to the start of the read tests, the system was subsequently warmed up with a series of single triple pattern matches over the SPO, POS, and OSP indexes, each of which performed Node to NodeID conversion.

6.2 Load Rates

This section analyses the load rates achieved when running Clustered TDB on a cluster of varying size. It should be noted that the absolute load rates achieved in this section are significantly slower than would normally be expected, due to a flaw in the handling of the data file by the TDB code underlying the prototype. This section does, however, still serve to illustrate the scaling effects when increasing the size of the cluster. Table 1 shows the overall load rate for the data file for standalone TDB and for 1, 2, and 3 machines, referred to as CTDB1, CTDB2, and CTDB3 respectively.

Table 1. Load rates

System Type	Average Load Rate (triples/s)
Standalone	6,946
CTDB1	4,276
CTDB2	8,973
CTDB3	12,536

Figure 3 illustrates loading rate over time. Figure 3 does not include standalone TDB, as it uses an optimised loading process that is not directly comparable to Clustered TDB. This process has not yet been implemented in Clustered TDB, and accounts for

much of the difference in load rate between CTDB1 and standalone TDB.

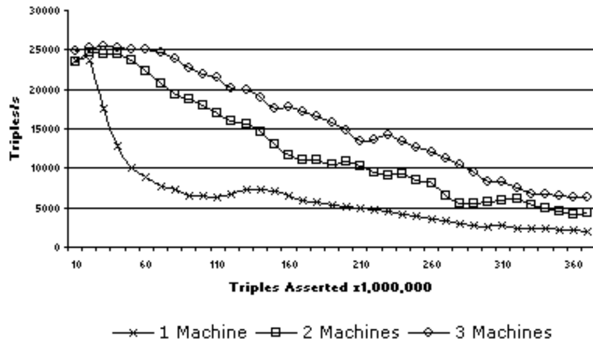

Figure 3. Rates of assertion during Clustered TDB load

These tests indicate that loading on Clustered TDB scales extremely well over small numbers of machines.

6.3 Read Performance

This section describes the tests of read performance that were run on Clustered TDB, including an explanation of the results. The read tests for TDB were performed using a test that simulated a load of 1 and 5 users, giving an indication of throughput scalability.

6.3.1 Individual Triple Pattern

An individual triple pattern, retrieved without NodeID to Node translation is the basic unit of a query, and can usually be retrieved quickly the node will be located on a single file on one server. It is not necessarily expected that there will be significant improvements in performance in a non-loaded situation, as only one machine will be performing work. Tests were performed that tested each of the subject, predicate, and object indexes, with Table 2 showing the results for a load of one user, and Table 3 showing the results for five:

Table 2. Reading individual triple patterns (1 user)

System Type	SPO Index (ms)	POS Index (ms)	OSP Index (ms)
Standalone	439	21530	512
CTDB1	361	1733	483
CTDB2	187	1958	264
CTDB3	178	1821	363

Table 3. Reading individual triple patterns (5 users)

System Type	SPO Index (ms)	POS Index (ms)	OSP Index (ms)
Standalone	1904	65602	2205
CTDB1	1784	6648	2053
CTDB2	664	6879	672
CTDB3	647	5617	628

Note that, since predicates generally have a much higher cardinality than subjects or objects, the predicate-centric tests are slower than subject-or object centric ones. These results show a general improvement in performance as the number of machines increases, particularly when the system is under higher throughput load. The performance of the predicate-centric tests is particularly notable, since clustered TDB even in a one machine configuration is much faster than standalone TDB. This is probably due to the horizontal partitioning effect that the use of vnode partitioning creates: index depths are lower in Clustered TDB.

6.3.2 Individual Triple Pattern With NodeID to Node conversion

This test measures NodeID to Node conversion. Since each NodeID to Node conversion is a separate operation, this test is highly parallelisable, and it should be expected that performance will improve significantly as the number of machines increases. Tables 4 and 5 detail the results.

Table 4. NodeID to Node tests (1 user)

System Type	SPO Index (ms)	POS Index (ms)	OSP Index (ms)
Standalone	398	122609	349
CTDB1	700	229453	550
CTDB2	433	172543	312
CTDB3	355	120815	333

Table 5. NodeID to Node tests (5 users)

System Type	SPO Index (ms)	POS Index (ms)	OSP Index (ms)
Standalone	1715	246026	1006
CTDB1	3208	813783	2183
CTDB2	1594	372920	762
CTDB3	1312	281082	878

These tests show generally useful scaling within Clustered TDB, particularly as the throughput load increases. It is noticeable that standalone TDB often outperforms Clustered TDB in this test. It is likely that this can be attributed to inefficient netcode within the Clustered TDB prototype: while some effort is made to batch process subqueries to make efficient use of network resources, each vnode is communicated with as a separate entity, even if it is located on the same server as many vnodes. This results in an unnecessarily large number of threads of communication, and the costs associated with these small queries can cause slowdown.

An example of how seriously skew can affect overall system performance was discovered during this phase of the evaluation. Initially, Clustered TDB's NodeIDs were formed of 8 bits type, followed by 12 bits of vnode id, and 44 bits file index. Tests showed that queries of the form ?s <p> ?o were exhibiting virtually no improvement in performance as the cluster had more machines added.

The reason for this poor performance was the fact that triple matches were emitted from the POS index in sorted order. Since the vnode ID bits were higher order than the disk address, all of the NodeIDs which had mappings stored on vnode ID n would return before the mappings in vnode ID n+1. This resulted in a

complete lack of parallelism when mapping NodeIDs to Nodes. Since this was the dominant cost in the query, the system did not improve in performance as the cluster scaled up. This effect was not as noticeable on subject or object oriented queries, because these tend to produce fewer results, and thus all the NodeID to Node conversions necessary could fit inside a single batch operation. The solution to this problem was to place vnode IDs after the disk address in the NodeID.

6.3.3 Join tests

This section details the performance of index joins on Clustered TDB, simulating the joins that are performed during SPARQL queries.

Join 1 is comprised of queries similar in form to "Select all the people in the system who like cheddar and live in Southampton", or:

```
SELECT ?person WHERE {?person <likes-cheese>
<cheddar> . ?person <lives-in> <Southampton>
.}
```

This test is extremely simple, with the first triple pattern of each query in the set typically matching 4-10 results, and the join as a whole processing up to 15 records. The fact that multiple results are returned from the first triple pattern means that multiple threads can be launched to answer all the following subqueries that make up the index join, meaning that this test is typically somewhat parallelisable. It is, however, small enough that speedup as the cluster increases in size is limited.

Table 6. Join test 1

System Type	1 User (ms)	5 Users (ms)
Standalone	379	201
CTDB1	297	319
CTDB2	114	135
CTDB3	122	124

Join 2 is comprised of queries similar in form to "For each person in the system, find the books that they like, and the interests they have", or:

```
SELECT ?person ?some-book ?some-interest
WHERE {?person <likes-book> ?some-book .
?person <has-interests> ?some-interest .}
```

For the given dataset, this query typically matches approximately 50,000 results on the first triple pattern, and returns around 70,000 records overall. The large number of matches in the first triple pattern make this query highly parallelisable, and this is borne out by the results detailed in Table 7.

Table 7. Join test 2

System Type	1 User (ms)	5 users (ms)
Standalone	131273	155147
CTDB1	133023	133963
CTDB2	62756	87676
CTDB3	50883	64518

Join 3 is comprised of queries similar in form to "Tell me everything about every book in the system", or:

```
SELECT ?book ?p ?o WHERE {?book <type>
<book> . ?book ?p ?o}
```

This query typically matches around 6,000 results on the first triple pattern, and around 90,000 overall. This query ought to be highly parallelisable, but interestingly shows poor results on mulit-machine systems. This result is a topic for further investigation.

Table 8. Join test 3

System Type	1 User (ms)	5 users (ms)
Standalone	1579	18414
CTDB1	3066	21082
CTDB2	7055	49108
CTDB3	6287	36051

Join 4 is comprised of queries similar in form to "Who else likes books that Alisdair likes?", or:

```
SELECT ?person ?book WHERE {<Alisdair>
<likes-book> ?book . ?person <p> ?book .}
```

This query typically matches 2-10 results on the first triple pattern, and up to 25 results overall. This query is somewhat parallelisable, but due to the low cardinality of the first triple match is not expected to scale to large numbers of machines.

Table 9. Join test 4

System Type	1 User (ms)	5 users (ms)
Standalone	379	1044
CTDB1	297	1115
CTDB2	114	402
CTDB3	122	485

Join 5 is comprised of queries similar to "who else feels about anything the same way Alisdair feels about food?", or:

```
SELECT ?s ?p ?o WHERE {<Alisdair> ?p <food>
. ?s ?p ?o .}
```

This query typically matches only a single result in the OSP index, and 15,000-40,000 in the POS index. It is not expected to parallelise well because each of the triple patterns can be matched from a single file on a single machine, and this expectation is borne out in the results. In a fully-featured prototype, the exceptions mechanism described in section 5.8 would offer a mechanism to spread such high cardinality properties across the cluster, enhancing parallelism.

Table 10. Join test 5

System Type	1 User (ms)	5 users (ms)
Standalone	3491	4530
CTDB1	696	2446
CTDB2	966	2320
CTDB3	1020	1401

7. Future Work

Beyond the full implementation of the design specified in this paper, we have two major goals for future work: enabling efficient distributed query optimisation, and exploration of different indexing techniques. TDB's current comprehensive indexing strategy is workable for triple storage, but becomes less practical when considering the additional dimensions required to store graph names and temporal information.

To mitigate this problem, we propose to investigate single-index solutions such as space filling curves[15]. Since related data in these indexes cannot be perfectly contiguous on disk, high latency disk based systems currently impair the practicability of such methods. However, as low latency solid state disks become increasingly practical, indexing mechanisms such as these may prove extremely worthwhile.

8. Conclusion

RDF stores suffer from issues with poor read and write performance. This paper has presented a review of techniques to enable the storage and querying of RDF over multiple machines, as a means of working with the large volumes of RDF that are inevitable as more and more information is encoded into semantic languages.

This paper contributes the design of Clustered TDB, a system that implements many of the techniques described in this review. The evaluation of the prototype so far has shown that Clustered TDB offers near linear scaling characteristics with respect to load times, and shows expected characteristics with respect to speedup and scaleup. Further, the design offers many useful characteristics: fault tolerance through vnode mirroring, a variety of mechanisms for addressing issues such as skew, and a mechanism to distribute properties fairly throughout the cluster. Further, Clustered TDB offers the ability to rebalance more cheaply than other clustered stores. As the Semantic Web scales up, clustered RDF stores such as Clustered TDB will make it possible to query very large volumes of data.

9. REFERENCES

[1] Abadi, D., Marcus, A., Madden, S. and Hollenbach, K. Scalable Semantic Web Data Management Using Vertical Partitioning. Proc. VLDB.

[2] Boral, H. Parallelism in Bubba Databases in Parallel and Distributed Systems, 1988. Proceedings. International Symposium on, 1988, 68-71.

[3] Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M. and Valduriez, P. Prototyping Bubba, a highly parallel database system. IEEE Transactions on Knowledge and Data Engineering, 2 (1). 4-24.

[4] Borthakur, D. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/current/hdfs_design.html, 2008.

[5] Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A. and Wilkinson, K. Jena: implementing the semantic web recommendations. International World Wide Web Conference. 74-83.

[6] Chris Bizer, A.S. Berlin SPARQL Benchmark Results. http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/index.html, 2008.

[7] Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters.

[8] DeWitt, D.J. and Gray, J. Parallel Database Systems: The Future of High Performance Database Processing. University of Wisconsin-Madison, Computer Sciences Dept., 1992.

[9] Erling, O. and Mikhailov, I. Towards Web Scale RDF, ISWC2008 submission. www.openlinksw.com/weblog/oerling/2008iswc_webscale_rdf.pdf, 2008.

[10] Harris, S. SPARQL query processing with conventional relational database systems.

[11] Harth, A., Umbrich, J., Hogan, A. and Decker, S. YARS2: A Federated Repository for Querying Graph Structured Data from the Web.

[12] Hayes, P. RDF Semantics, http://www.w3.org/TR/rdf-mt/, 2004.

[13] Hua, K.A. and Lee, C. An adaptive data placement scheme for parallel database computer systems Proceedings of the sixteenth international conference on Very Large Databases, 1990, 493-506.

[14] Khan, M., Paul, R., Ahmed, I. and Ghafoor, A. Intensive Data Management in Parallel Systems: A Survey. Distributed and Parallel Databases, 7 (4). 383-414.

[15] Lawder, J.K. and King, P.J.H. Using Space-Filling Curves for Multi-dimensional Indexing. proceedings of the 17th British National Conference on Databases (BNCOD 17), 1832. 20-35.

[16] Prud'hommeaux, E. and Seaborne, A. SPARQL Query Language for RDF. W3C Candidate Recommendation. World Wide Web Consortium, April 2008.

[17] Smith, D.A., Owens, A., schraefel, m., Sinclair, P., Andre, P., Wilson, M., Russell, A., Martinez, K. and Lewis, P. Challenges in Supporting Faceted Semantic Browsing of Multimedia Collections.

[18] Stonebraker, M. The Case for Shared Nothing. Database Engineering Bulletin, 9 (1). 4-9.

[19] Wilkinson, K., Sayers, C., Kuno, H. and Reynolds, D. Efficient RDF Storage and Retrieval in Jena2. Proceedings of SWDB, 3. 7-8.