

Introduction

In fields as diverse as pharmacology, finance, fraud detection, and intelligence analysis, better analysis and decision making can be facilitated by taking into consideration large amounts of heterogeneous data from many sources in many formats, and degrees of structure, and update rates. Pouring this data together often yields new insights and interesting cross-connections not readily apparent when considering the various data sets in isolation. Such “mash-ups” can provide the basis for operational decision making in complex and dynamic domains, support new forms of online collaboration, and help manage risks in complex markets.

In order to address this problem, we require three things. First, we must be able to load and query very large data sets that exceed the reasonable processing capabilities of even high end server platforms. Second, those data sets are heterogeneous and interesting data often appears after the system has been deployed, so we must be able to dynamically align the schema for those data sets and to continuously integrate new data. Third, we require the ability to maintain data provenance and drill down into the source detail.

The relational model benefits tremendously from its structure, but lacks the flexibility to rapidly and declaratively integrate new schema into existing systems – relational data integration efforts are often measured in months, not minutes. Expressive Semantic Web technologies such as RDF and OWL have helped reshape this problem, but RDF database technology has not been able to keep up with scale demands. Until very recently, RDF databases and OWL reasoners have not tried to tackle the issues associated with large dynamic data sets, and were insufficiently scalable to attack real world problems where data size can be on the order of billions or even trillions of triples. Without the ability to reach scale, potential Semantic Web adopters turn to cloud computing technologies such as map/reduce, not fully understanding the tradeoffs between the two technologies and, in particular, the limitations of map/reduce processing for handling graph structured or linked data.

Bigdata®^{1 2} is a horizontally-scaled, general purpose storage and computing fabric for ordered data (B+Trees), designed to operate on a cluster of commodity hardware. While many clustered databases rely on a fixed, and often capacity limited, hash-partitioning architecture, bigdata® uses dynamically partitioned key-range shards. This architecture was chosen to remove any realistic scaling limits – in principle, bigdata® may be deployed on 10s, 100s, or even thousands of machines. Further, and unlike hash-partitioned approaches, new capacity may be added incrementally to data centers without requiring the full reload of all data. On top of that core is the bigdata® RDF Store, a massively scalable RDF database supporting RDFS and OWL Lite reasoning, high-level query (SPARQL), and datum level provenance.

¹ <http://www.bigdata.com/blog>

² <http://www.sourceforge.net/projects/bigdata>

| | |
|--|----|
| Introduction..... | 1 |
| Database Architecture..... | 3 |
| B+Trees..... | 3 |
| Dynamic Partitioning..... | 4 |
| Metadata Service..... | 5 |
| Data Services | 6 |
| Bloom filters | 9 |
| Concurrency Control..... | 9 |
| Managing database history | 10 |
| Services Architecture..... | 11 |
| Service Discovery | 11 |
| Deployment..... | 12 |
| High Availability | 13 |
| RDF Database Architecture | 14 |
| Resource Description Framework..... | 14 |
| RDF Database Architecture | 15 |
| Distributed jobs and bulk data load | 16 |
| Distributed Query..... | 18 |
| Inference, truth maintenance and datalog | 21 |
| Provenance..... | 23 |
| Performance evaluation | 25 |
| Evaluation platform | 25 |
| Data load performance..... | 26 |
| Disk utilization..... | 28 |
| Distributed Query performance | 29 |
| Related work | 29 |
| Conclusion | 30 |

Database Architecture

Bigdata® is a scale-out architecture for persistent, ordered data. The overall approach which we have chosen utilizes key-range partitioned B+Tree³ indices distributed across the resources of a cluster. This choice was influenced by previous work on distributed database architectures, including Google’s bigtable⁴ project.

B+Trees

At its heart, bigdata® is an architecture for distributed B+Trees. Each B+Tree instance can contain petabytes of data. In order to scale-out, the B+Tree is dynamically divided into key-range shards, known as index partitions. The B+Tree is a central data structure for database systems because it provides search, insert, update in logarithmic amortized time. The bigdata® B+Tree fully implements the tree balancing operations and remains balanced under inserts and deletes. Scale-out uses read-write B+Tree instances with multiple revisions and read-only index segment files. The index segment files support fast double-linked navigation between leaves. The B+Tree implementation is single threaded under mutation, but allows concurrent readers. In general, readers do not use the mutable view of a B+Tree, so readers do not block for writers.

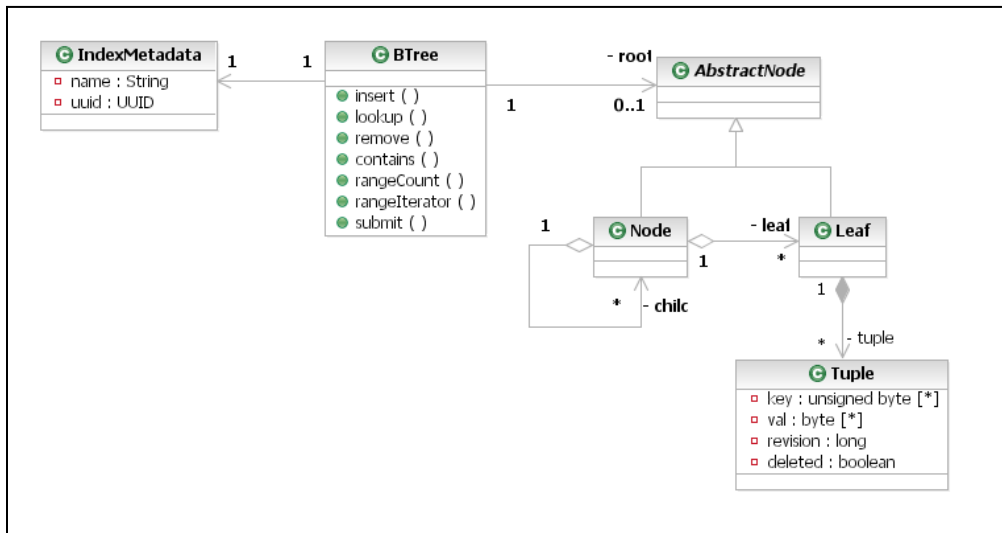


Figure 1 -- B+Tree architecture.

In bigdata®, an index maps *unsigned byte[]* keys to *byte[]* values. To facilitate applications, mechanisms are provided which support the encoding of single and multi-field numeric, ASCII, and Unicode data. Likewise, extensible mechanisms provide for (de)serialization of application data as *byte[]*s for values. An index entry is known as a “tuple”. In addition to the key and value, a tuple contains a “deleted” flag which is used to prevent reads through to historical data in index views, discussed below, and a revision

³ Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1 (1972) 173-189.; Bayer, R. and Unterauer, Prefix B-trees. *ACM Trans. On Database Systems*, 2,1 (Mar., 1977) 11-26

⁴ “Bigtable: A Distributed Storage System for Structured Data”, <http://labs.google.com/papers/bigtable.html>

timestamp, which supports optional transaction processing based on Multi-Version Concurrency Control (MVCC)⁵. The IndexMetadata object is used to configure both local and scale-out indices. Some of its most important attributes are the index name, index UUID, branching factor, objects which know how to serialize application keys and both serialize and deserialize application values store in the index, and the key and value coder objects.

The B+Tree never overwrites records (nodes or leaves) on the disk. Instead, it uses copy-on-write for clean records, expands them into Java objects for fast mutation and places them onto a hard reference ring buffer for that B+Tree instance. On eviction from the ring buffer, and during checkpoint operations, records are coded into their binary format and written on the backing store. All backing stores in the same JVM share an LRU cache. For a 64-bit JVM, that LRU cache may be gigabytes in size.

Records can be directly accessed in their coded form. The default key coding technique is front coding, which supports fast search with good compression. Canonical Huffman⁶ coding is supported for values. Custom coders may be defined, and can be significantly faster for specific applications. Record level compression may also be used. When specified, the entire B+Tree node or leaf is compressed before it is written onto the backing store.

The high-level API for the B+Tree includes methods which operate on a single key-value pair (insert, lookup, contains, remove), methods which operate on key ranges (rangeCount, rangeIterator), and a set of methods to submit Java procedures which are mapped against the index and execute locally on the appropriate data services (see below). Scale-out applications make extensive use of the key-range methods, mapped index procedures, and asynchronous write buffers (see below) to ensure high performance with distributed data.

Dynamic Partitioning

Bigdata® indices are dynamically broken down into key-range shards, called *index partitions*, in a manner which is completely transparent to clients. Each index partition is a collection of local resources which contain all tuples for some key-range of a scale-out index and is assigned a unique identifier (an integer). There are three basic operations on an index partition: *split*, which divides an index partition into two (or more) index partitions covering the same key-range; *move*, which moves an index partition from one data service to another, typically on a different machine in the cluster; and *join*, which joins two index partitions that are siblings, creating a single index partition covering the same key-range. These operations are invoked transparently and asynchronously.

⁵ Reed, D.P.. "Naming and Synchronization in a Decentralized Computer System". *MIT dissertation*.
<http://www.lcs.mit.edu/publications/specpub.php?id=773>

⁶ Huffman coding, http://en.wikipedia.org/wiki/Huffman_coding

⁷ Canonical Huffman coding, http://en.wikipedia.org/wiki/Canonical_Huffman_code

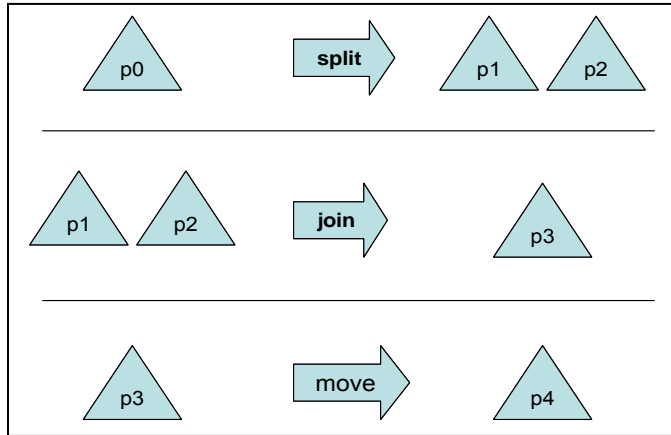


Figure 2 – Operations on index partitions : Split, Join, and Move.

The data in the indices is strictly conserved by these operations, only the index partition *identifier*, the index partition *boundaries* (split, join), or the index partition *location* (move) are changed. The index partition identifier is linked to a specific key-range and a specific location. Since these operations change either the key-range and/or the location, they always assign a new index partition identifier. Requests for old index partitions are easily recognized as having index partition identifiers which have been retired and result in *stale locator exceptions*. The client-side views of the scale-out indices automatically trap stale locator exceptions and redirect and reissue requests as necessary.

Metadata Service

Index partition *locators* are maintained in a *metadata index* that lives on a specialized *data service* known as the *metadata service*. An index partition locator maps a key-range for an index onto an index partition identifier and the (logical) data service identifier hosting that index partition. The key for the tuples in the metadata index is simply the first key that could enter the corresponding index partition⁸. The physical data services are resolved using zookeeper⁹, which handles the master election and failover chains for data services.¹⁰ Clients can monitor zookeeper nodes corresponding to logical data services and receive asynchronous updates if the master for a logical data service changes. A metadata service with a single 200MB shard can address nearly 3 terabytes of data. A metadata service with 1,000 shards can address nearly 200 petabytes. This is a reasonable upper bound for a single metadata server instance. Beyond that scale, the metadata service must itself be distributed.¹¹

⁸ Bigdata® has a theoretical upper bound of 400 petabytes per scale-out index (based on 32-bit partition identifiers and 200M index partitions). In order to realize the theoretical upper bound, bigdata® would need to manage a partitioned metadata index and reclaim old partition identifiers.

⁹ Zookeeper is part of the Hadoop project, <http://hadoop.apache.org/zookeeper/>.

¹⁰ Logical indirection for failover chains has not been implemented yet. Clients currently resolve logical data service identifiers directly as Jini service identifiers. Indirection will be implemented with the HA architecture.

¹¹ The metadata service can be replicated without being distributed. A replicated metadata service supports failover. A distributed metadata service breaks the single machine barrier and increases the total volume of data which can be managed.

Data Services

The data service maintains an append-only write buffer, known as a *journal*, and an arbitrary number of read-only, read-optimized *index segments*. Each index partition is, in fact, a *view* onto (a) the mutable B+Tree on the live journal; and (b) historical data on a combination of old journals and index segments. The nominal capacity of the journal is ~200M. Likewise, the target size for the index segments in a compact index partition view is ~200M. There may be 100s or 1000s of index partitions per data service¹². Thus index segment files form the vast majority of the persistent state managed by a data service.

Periodically the journal overflows, a new journal is created, and an asynchronous process begins which migrates buffered writes from the old journal onto new index segments. Asynchronous overflow processing defines two additional operations on index partitions: *build*, which copies *only* the buffered writes for the index partitions from the old journal onto a new index segment; and *merge*, which copies all tuples in the index partition view into a new index segment. Index partition *builds* make it possible to quickly retire the old journal, but they add a requirement to maintain delete markers on tuples in order to prevent historical tuples from re-entering the index partition view. Index partition *merges* are more expensive, but they produce a compact view of the tuples in which duplicates have been eradicated. The decision to *build* vs. *merge* is made locally based on the complexity of the index partition view and the relative requirements of different index views for a data service. The decision to *split* an index partition into two index partitions or to *join* two index partitions into a single index partition is made after a *merge* when there is a good estimate of the space requirements on disk for the index partition. The decision to move an index partition is based on load. A *merge* is always performed before a move to produce a compact view which is then blasted across a socket to the receiving service¹³.

When a scale-out index is registered, the following actions are taken: First, a metadata index is created for that scale-out index on the metadata service. This will be used to locate the index partitions for that scale-out index. Second, a single index partition is created an arbitrary data service. Third, a locator is inserted into the metadata index mapping the key-range ($[], \infty$) onto that index partition¹⁴. Clients resolve the metadata service, and probe it to obtain the locator(s) for the desired scale-out index. The locator contains the (logical) data service identifier as well as the key-range ($[], \infty$) for the index partition. Clients resolve the data service identifier to the data service and begin writing on the index partition on that data service.

¹² An estimate of between 1 and 2 billion triples per server class machine may be used for capacity planning. Each triple occupies approximately 57 bytes on the disk. One billion triples is approximately 53 GB on disk. Additional storage is required for temporary files. Either retaining historical revisions of the data or using replicated writes for high availability would increase the local disk requirements.

¹³ A similar design was described in “Bigtable: A Distributed Storage System for Structured Data”, <http://labs.google.com/papers/bigtable.html>.

¹⁴ Within bigdata, all keys are translated into unsigned byte[]s. An empty byte[] is the first possible key in any bigdata index. The symbol ∞ is used to indicate an arbitrarily long unsigned byte[] containing 0xFF in all positions and corresponds to the greatest possible key in any bigdata index and is indicated internally by a *null* reference.

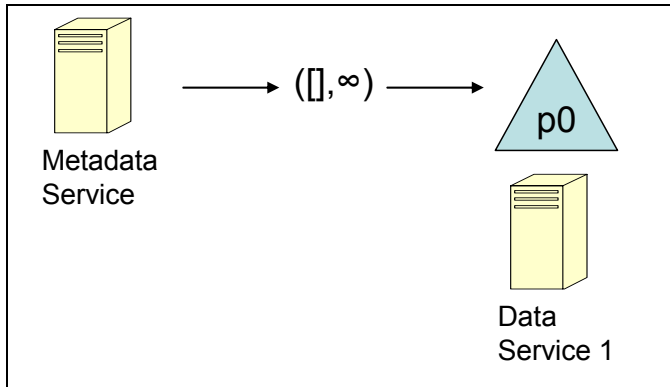


Figure 3 Initial conditions place a single index partition on an arbitrary host representing the entire B+Tree.

Eventually, writes on the initial index partition will cause its size on disk to exceed a configured threshold (~200M) and the index partition will be split.

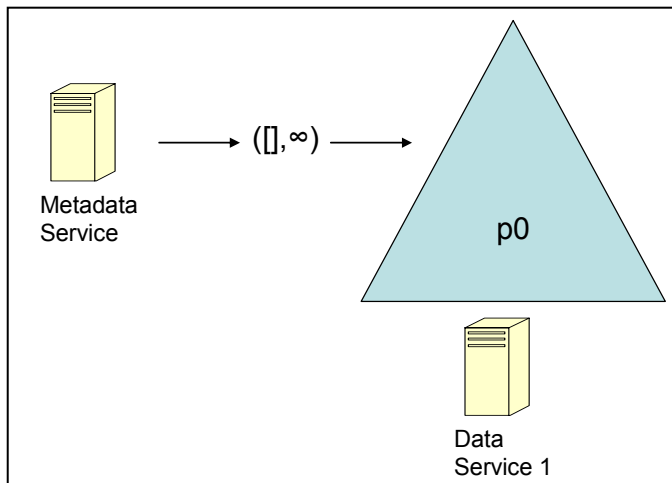


Figure 4 The index partition grows until it reaches a configured threshold in size on disk.

The split(s) are identified by examining the tuples in the index partition and choosing one or more *separator key(s)*. Each separator key specifies the first key which may enter a given index partition. The separator keys for the locators of a scale-out index always span the key range $([], \infty)$ without overlap. Thus each key always falls into precisely one index partition.

If necessary, applications may place additional constraints on the choice of the separator key. For example, this may be done to ensure that an index partition never splits a logical row. That guarantee may be used to achieve extremely high concurrent write rates using ACID, but non-transactional, operations since concurrency control may be conducted locally on the data service.

The potential throughput of an index increases as it is split and distributed across the machines in the cluster. In order to rapidly distribute an index across the cluster and

thereby increase the resources which can be brought to bear on that index, a *scatter split* is performed early in the life cycle of the first index partition for each scale-out index.

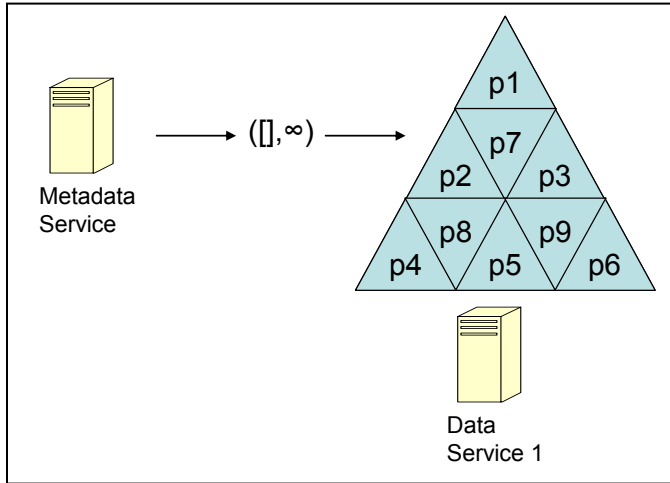


Figure 5 Preparing for the initial scatter split of an index.

Unlike a normal split, which replaces one index partition with two index partitions, the scatter split replaces the initial index partition with $N * M$ index partitions, where N is the number of data services and M is a small integer.

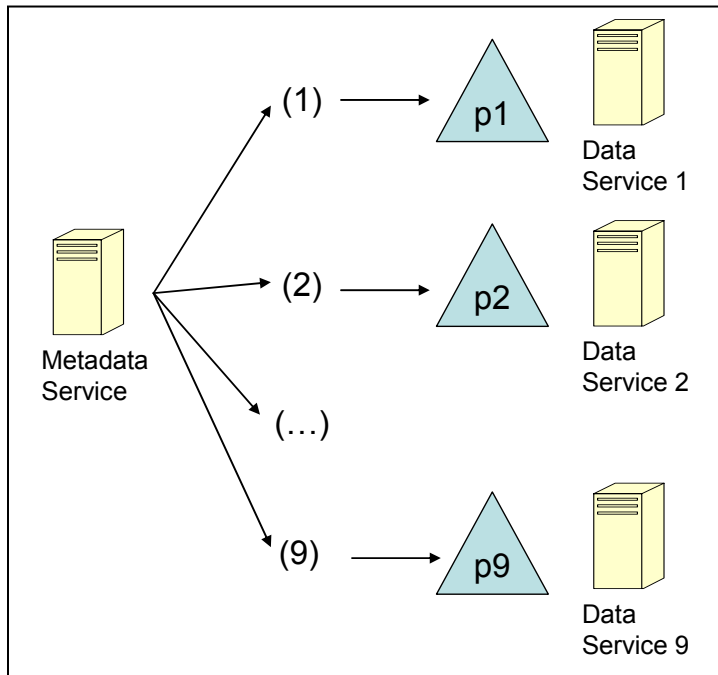


Figure 6 After the scatter split, the index is distributed across the resources of the cluster. In this example, there are nine data services in the cluster. There can be hundreds.

The new index partitions are redistributed across the cluster, leaving every N^{th} index partition on the original data service. After the scatter split, the throughput of the index

may be dramatically increased – assuming that the clients are capable of supplying a sufficient workload!

Bloom filters

A *bloom filter* is an in memory data structure that can very rapidly determine whether a key IS NOT in an index. When the bloom filter reports "no", you are done and you do not touch the index. When the bloom filter reports "yes", you have to read the index to verify that there really is a hit. Bloom filters are a stochastic data structure, require about 1 byte per index entry, and must be provisioned up front for an expected number of index entries. So if you expect 10M triples, that is a 10MB data structure. Since bloom filters do not scale-up, they are automatically disabled once the number of index entries in the mutable B+Tree exceeds about 2M tuples.

However, bloom filters are great for scale-out. Each time an index partition build or merge operation generates a new index segment file, the data on the mutable B+Tree is migrated into read-optimized index segments. Every time we overflow a journal, we wind up with a new (empty) B+Tree to absorb writes, so the bloom filter on the journal is automatically re-enabled. Further, during build and merge operations we have perfect knowledge of the number of tuples in an index segment and generate an exact fit bloom filter. This provides a dramatic boost for distributed query and is yet another way in which scale-out architectures can leverage more resources.

Concurrency Control

Bigdata® supports optional transactions based on MVCC. Many database architectures are based on two phase locking (2PL), which is a pessimistic concurrency control strategy. In 2PL, a transaction acquires locks as it executes and readers and writes will block in their access conflicts with the locks for running transactions. MVCC is an optimistic concurrency control strategy and relies on the use of timestamps to detect conflicts when a transaction is validated. MVCC allows very high concurrency since readers never block and writers can run concurrently even when they touch the same region of the disk (there is no sense of a row, page or table lock). If two writers modify the same tuple in an index, then that conflict is detected when the transaction validates and the second transaction will fail unless the conflict can be resolved (in fact, bigdata® can resolve most write-write conflicts for RDF). The MVCC design and the ability to choose whether or not operations will be isolatable by transactions is driven deep into the architecture, including the copy-on-write mechanisms of the B+Tree, the immortal store architecture of the journal, and history retention policy of data services.

Transaction processing is optional by design. Transactions can greatly simplify application architecture, but they can limit both performance and scale. Applications that need very large scale must carefully consider the tradeoffs involved with transaction processing. For example, Google® developed their “row store”¹⁵ to address a set of very specific application requirements. In particular, they had a requirement for extremely

¹⁵ “Bigtable: A Distributed Storage System for Structured Data”, <http://labs.google.com/papers/bigtable.html>

high concurrent read writes and very high concurrent write rates. Distributed transaction processing was ruled out because each commit must be coordinated with the transaction service, which limits the potential throughput of a distributed database. In their design, Google® opted to restrict concurrency control to ACID¹⁶ operations on “rows” within a “column family.” With this design, a purely local locking scheme may be used and substantially higher concurrency may be obtained. Bigdata® uses this approach for its “row store”, for the lexicon for an RDF database, and for high throughput distributed bulk data load.

The bigdata® architecture is designed to support full distributed transactions¹⁷, but not to require their use. An “isolatable” index (one which supports transactional isolation) maintains per-tuple revision timestamps, which are used to detect and, when possible, reconcile write-write conflicts. The transaction service is responsible for assigning transaction identifiers, which are timestamps, revision timestamps, and commit timestamps. The transaction service maintains a record of the open transactions and manages read-locks on the historical states of the database. The read-lock is just the timestamp of the earliest running transaction, but it plays an important role in managing resources as discussed below.

Managing database history

Bigdata® is an *immortal database* architecture with a configurable *history retention policy*. An immortal database is one in which you can request a consistent view of the database at any point in its history, essentially winding back the clock to the state of the database at some prior day, month or year. This feature can be used in many interesting ways, including regulatory compliance, examining changes in the state of accounts over time, etc.

For many applications, access to unlimited history is not required. Therefore you can configure the amount of history that will be retained by the database. This is done by specifying the minimum age before a commit point may be released, e.g., 5 minutes, 1 day, 2 weeks, or 12 months. The minimum release age can also be set to zero, in which case bigdata will release the resources associated with historical commit points as soon as the read locks for those resources have been released. Equally, the minimum age can be set to a very large number, in which case historical commit points will never be released.

The minimum release age determines which historical states you can access, not the age of the oldest record in the database. For example, if you have a 5 day history retention policy, and you insert a tuple into an index, then that tuple would remain in the index until 5 days after it was *overwritten or deleted*. If you never update that tuple, the original value will never be released. If you do delete the tuple, then you will still be able

¹⁶ ACID is an acronym for four common database properties: Atomicity, Consistency, Isolation, Durability. Reuter, Andreas; Haerder, Theo (December 1983). “Principles of Transaction-Oriented Database Recovery”. ACM Computing Surveys (ACSUR) 15 (4): 287-317.

¹⁷ Bigdata® supports both read-only and read-write transactions in its single server mode and distributed read-only transactions, which are used for query and when computing the closure over an RDF database. Support for distributed read-write transactions is not yet complete.

to read from historical database states containing that tuple for the next 5 days. Applications can apply additional logic if they want to delete records once they reach a certain age. This can be done efficiently in terms of the tuple revision timestamps.

Services Architecture

In addition to the metadata service, the data services, and the transaction service as described above, the bigdata® service architecture defines a *load balancer service*, a *services manager* (starts and stops services), and a *client service* (a container for executing distributed tasks). Jini (now the Apache river project) is used for service registry and discovery. Global synchronous locks and configuration management are realized using Apache zookeeper. Support for SPARQL processing is achieved by integration with the Sesame 2 platform¹⁸.

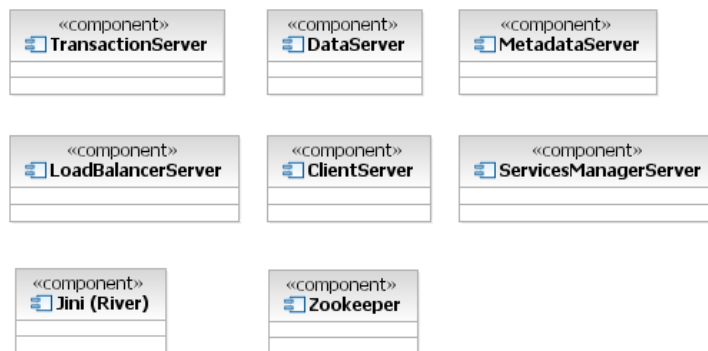


Figure 7 Bigdata® services

Service Discovery

Bigdata® services advertise themselves in a Jini service registrar and are discovered by lookups against that registrar. Clients await discovery of the transaction service and the metadata service, and then register or lookup indices using the metadata service. The metadata service maps key ranges for each scale-out index onto logical data services. When a client issues a request against a scale-out index, the bigdata® library transparently resolve the locator(s) for that query. Clients obtain proxies for the data services using jini, then talk directly to the data services. This process is illustrated in Figure 8 and is completely transparent to bigdata® applications. The client library automatically handles redirects when an index partition is moved, split or joins and data service failover.

¹⁸ Support for additional RDF platforms, including Jena, is being considered.

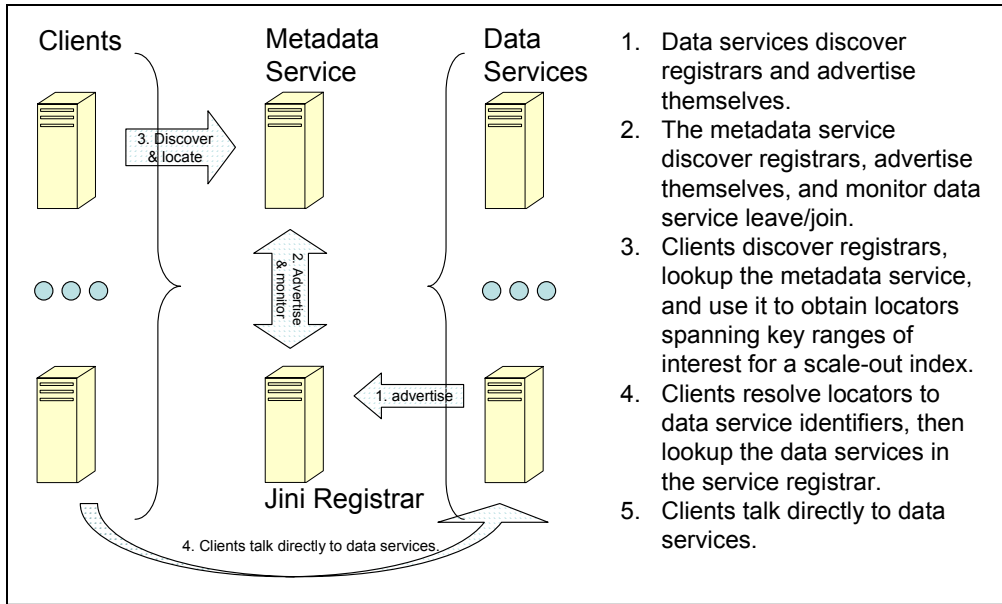


Figure 8 -- Service discovery in bigdata showing clients resolving locators to data services.

Deployment

A deployment of a bigdata® federation on a cluster maps services onto hosts. The entire configuration, including the constraints on service deployment, is described in a Jini configuration file. Each host runs a services manager instance. The services manager reads the configuration, optionally pushes it into zookeeper, and thereafter starts and stops services, as required and permitted, on the local host in order to satisfy the configuration requirements. An init.d style script is run from a cron job and reads the target run state {start, stop, status, destroy} from a file on a shared volume¹⁹. The script will start or stop bigdata service managers and associated components as necessary to achieve the desired run state.

In benchmarking trials we have used clusters of up to 15 nodes, but bigdata should scale-out transparently on much larger clusters. A sample configuration for four nodes is illustrated below. While the zookeeper documentation suggests that it should have a dedicated machine, in our experience zookeeper is a lightly utilized component and was placed for the purposes of evaluation with other relatively lightly utilized components.

¹⁹ Cluster deployment is only supported on Linux at this time, but alternative installers could be developed for Windows or other operating systems.



Figure 9 A sample deployment on a 4 node cluster. This is the minimum configuration for scale-out. Two machines are dedicated to data (blade1 and blade2). One machine is dedicated to client processing (blade3). The last machine is dedicated to various management services. In real world deployments, redundant instances would exist for critical services.

High Availability

The bigdata® architecture is designed to obtain maximum performance from *local* computing resources (CPU, DISK, RAM) and is intended to provide high availability (HA) and service failover through replicated state²⁰.

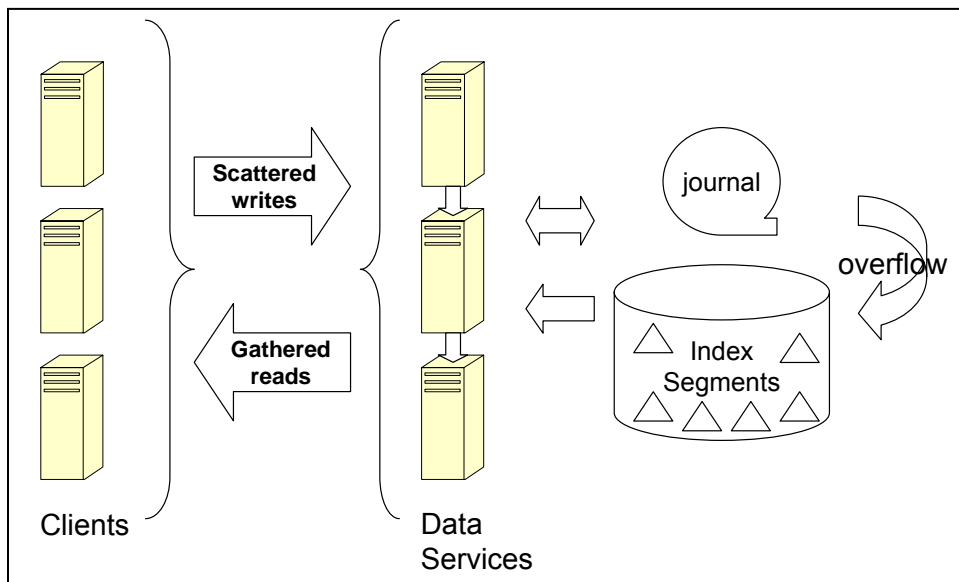


Figure 10 High Availability based on replicated writes.

Bigdata clients scatter reads and gather writes across the cluster as illustrated in Figure 10. In the high availability architecture, there are multiple physical instances for each logical data service. One of these physical instances wins an election in zookeeper and is

²⁰ Bigdata® has two high availability strategies. While neither of them has been realized yet, high availability based on shared volumes and failover services instances should be relatively easy to realize.

designated as the master. Clients write on the master for logical data service, and the master streams writes to the secondaries using their failover order. The failover order is determined by the same master election process, and is just the sequence of the services in the master election queue in zookeeper. The commit protocol is modified slightly for HA, and the master waits until the secondaries are synchronized before returning to the client. In HA, clients may read from any data service instance for the same logical data service.

In this scenario, configuration files, scripts, etc. are placed on a shared volume but the persistent state of the services, and especially the persistent state of the B+Trees, is placed on *local* disk. This optimizes the benefits of a commodity cluster, deriving redundancy and performance from comparatively inexpensive components.

An alternative HA deployment strategy is to use NFS, NAS, SAN, etc. for persistent state. The separation of the service layer from the file system can make it easier to administer and decouples service instance failure handling from storage management since the service may simply be restarted on another host (zookeeper provides the guarantee that the master will be safely elected). However, using shared volumes can substantially reduce performance and increase cost.

RDF Database Architecture

In this section we define the Resource Description Framework (RDF), and show how an RDF database is realized using the bigdata® architecture.

Resource Description Framework

The Resource Description Framework^{21 22} (RDF) may be understood as a general-purpose, schema-flexible model for describing metadata and graph-shaped information. RDF represents information in the form of statements (triples or quads). Each triple connotes an edge between two nodes in a graph. The quad position can be used to give statements identity (our provenance mechanism is based on this approach) or to place statements within a named graph. RDF provides some basic concepts used to model information - statements are composed of a subject (a URI or a Blank Node), a predicate (always a URI), an object (a URI, Blank Node, or Literal value), and a context (a URI or a Blank Node). URIs are used to identity a particular resource²³, whereas Literal values describe constants such as character strings and may carry either a language code or data type attribute in addition to their value. RDF also provides an XML-based syntax (called RDF/XML²⁴) for interchanging RDF graphs.

²¹ Resource Description Framework, <http://www.w3.org/RDF/>

²² RDF Semantics, <http://www.w3.org/TR/rdf-mt/>

²³ The benefit of URIs over traditional identifiers is two fold. First, by using URIs, RDF may be to describe addressable information resources on the Web. Second, URIs may be assigned within namespaces corresponding to Internet domain, which provides a decentralized mechanism for coining identifiers.

²⁴ <http://www.w3.org/TR/rdf-syntax-grammar/>

There is also a model theoretic layer above the RDF model and RDF/XML interchange syntax that is useful for describing ontologies and for inference. RDF Schema²⁵ and the OWL Ontology Web Language²⁶ (OWL) are two such standards-based layers on top of RDF. RDF Schema is useful for describing class and property hierarchies. OWL is a more expressive model. Specific OWL constructs may be applied to federation and semantic alignment, such as owl:equivalentClass and owl:equivalentProperty (for aligning schemas) and owl:sameAs (for dynamically snapping instance data together).

There is an inherent tension between expressivity and scale, since high expressivity is computationally expensive and only gets more so as data size increases. Bigdata® has focused on scale over expressivity, but we are also examining integrations with OWL reasoners.

RDF Database Architecture

An RDF database is a persistent collection of triples or quads, defining the arcs of a graph (for triples), or zero or more graphs (for quads). The nodes of the graph(s) are RDF Resources. The arcs describe named relationships between those nodes (RDF Predicates). In addition to these “link” properties, each node may have a collection of attribute properties.

Bigdata® supports three distinct RDF database modes: triples, triples with provenance, and quads. These modes reflect slight variations on a common schema. Abstractly, the schema can be conceptualized as a **Lexicon** and a **Statement** relation.

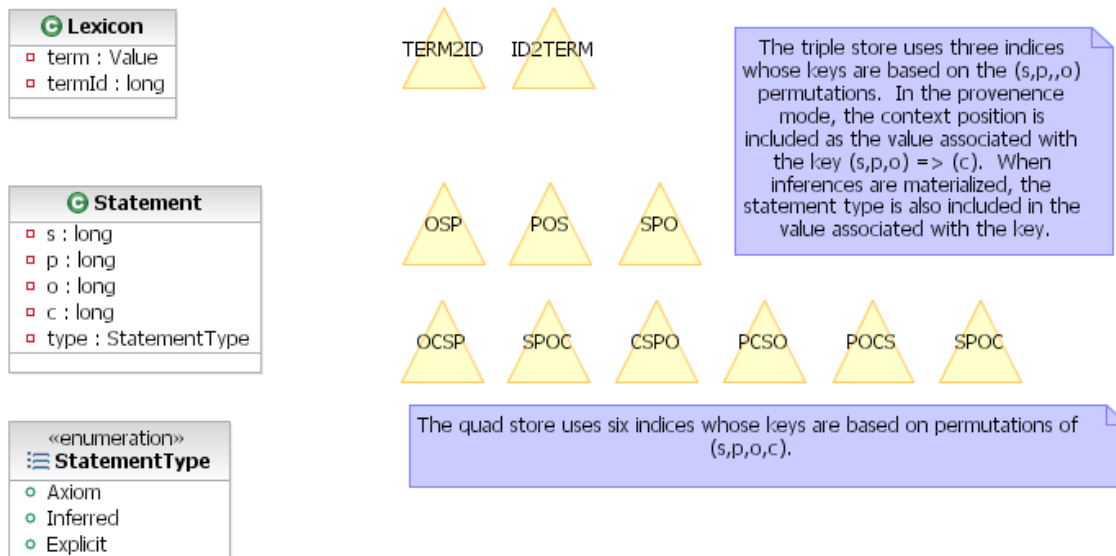


Figure 11 -- Abstract schema for the RDF database.

²⁵ <http://www.w3.org/TR/rdf-schema/>

²⁶ <http://www.w3.org/2004/OWL/>

The **Lexicon** relation fully normalizes RDF values using a pair of indices: TERM2ID, which maps RDF Values (URIs, literals and blank node identifiers) onto unique 64-bit internal identifiers; and ID2TERM, which maps the internal identifiers onto the RDF Value. These identifiers carry bit flags which make it possible to determine the kind of RDF Value (URI, literal, or blank node) by inspection. An optional full text index maps tokens extracted from RDF Values onto the internal identifiers for those RDF values and may be used to perform keyword search against the triple or quad store. The keys for the TERM2ID index are formed such that RDF data type literals and RDF language coded literals fall each within their own key-range and may optionally be normalized onto the value space of the associated data type.

The **Statement** relation models the Subject, Predicate, Object and, optionally, the Context, for each statement. For the triples mode, the Context is not used. For the provenance mode, the Context is always a statement identifier assigned by the Lexicon using a key formed from the internal identifiers for the statement in the (s,p,o) key order. Statement identifiers are externalized as blank nodes and may be bound in SPARQL queries in order to associate statements with their metadata. For the quads mode, the context position is told data as interchanged in a format such as TriG²⁷.

RDF query is based on *statement patterns*. A triple pattern has the general form (S,P,O), where S, P, and O are either variables or constants in the *Subject*, *Predicate*, and *Object* position respectively. For the provenance mode, query is based on triple patterns with a filter on the context position. For the quad store, this is generalized as patterns having the form (S,P,O,C), where C is the context (or graph) position and may be either a blank node or a URI.

The bigdata® RDF database architecture defines a perfect access path for each possible statement pattern²⁸. For the triple and provenance modes, these access paths are based on the SPO, POS, and OSP indices²⁹. In each case the name of the index indicates the manner in which the Subject, Predicate, and Object have been (re-)ordered.

The ensemble of these indices is collectively an RDF database *instance*. Any number of RDF database instances may be created within a single bigdata federation deployment. Each instance may support triples, triples with provenance, or quads. SPARQL query is available for all database modes.

Distributed jobs and bulk data load

The bigdata® architecture defines a distributed program execution model suitable for processing ordered data and therefore differing in several ways from traditional map/reduce processing. A distributed bigdata program (a “master”) is defined as a Java class and configured using the same Jini configuration mechanisms as the other services

²⁷ TriG, <http://www4.wiwiiss.fu-berlin.de/bizer/TriG/Spec/>

²⁸ Andreas Harth, Stefan Decker. "[Optimized Index Structures for Querying RDF from the Web](#)". 3rd Latin American Web Congress, Buenos Aires - Argentina, Oct. 31 - Nov. 2 2005.

²⁹ For the quad store, the perfect access paths require six indices. Those indices are: SPOC, POCS, OCSPO, PCSO, and SOPC.

in the bigdata federation. The master extends an abstract base class providing reusable functionality and defines factory methods for creating subtasks (the “clients”) that will be executed within client service containers distributed across the federation.

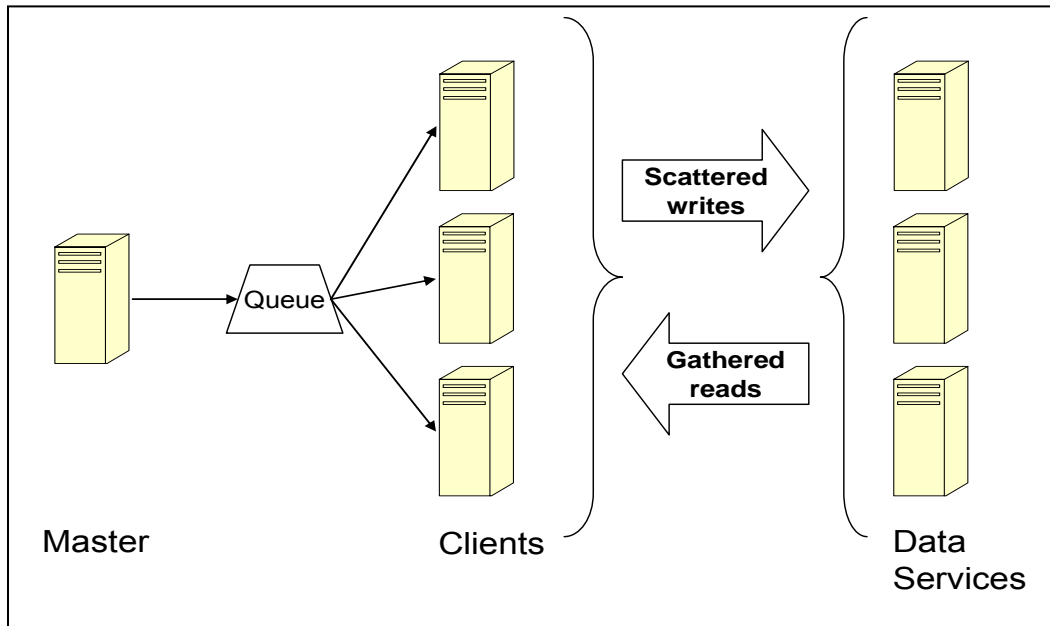


Figure 12 -- Master and clients for a distributed job.

The master runs a scanner, which can read from a file system or other data source. The master assigns work to the clients using a hash partitioning scheme similar to that found in map/reduce systems. Each client runs a parser/loader task. The parser/loader task consumes pathnames assigned to that client by the job master, reads from the corresponding file, parses the RDF/XML data, prepares writes for each index (TERM2ID, ID2TERM, SPO, POS, or OSP), and then buffers those writes on the appropriate *asynchronous write buffer*.

There is one asynchronous write buffer for each index. The buffer instances are shared by all threads for the same client task, thus writes are combined across tasks. Internally, see Figure 13, the asynchronous write buffer consists of an input queue, a redirect queue (not shown), and zero or more output queues – one for each index partition for which writes are currently buffered.

The asynchronous write buffer plays several roles. First, it decouples the client, to the maximum extent possible, from the latency of the write requests. Second, it transparently breaks down the tuples based on the current index partitions (and automatically handles exceptions if it learns that an index partitions has been split, joined, or moved, which is the role of the redirect queue). Third, duplicate tuples emerging from concurrent tasks can be easily filtered out. Fourth, it ensures that all remote index writes have a good “chunk” size (10,000 or more tuples). “Chunkier” writes translate into a performance benefit for several different reasons (B+Trees are ordered data structures and ordered writes make more efficient use of disk, plus there are efficiencies associated with the

B+Tree implementation, which uses a copy-on-write strategy, that result in substantially less DISK IO for chunkier writes).

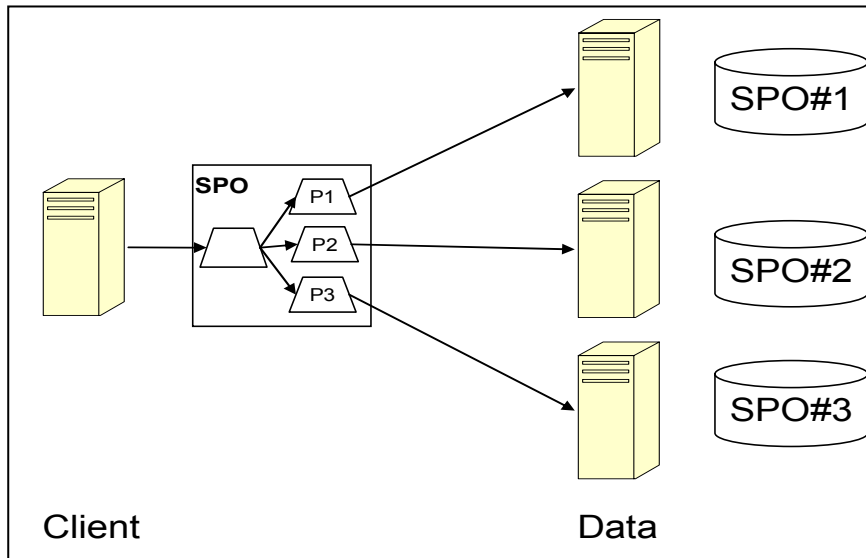


Figure 13 -- Asynchronous write buffer for SPO index on one client. This is the mechanism for efficient scattered writes for bulk data load.

Since any given parser/loader task must block until it holds the term identifiers assigned by the TERM2ID index before it can write on the other indices (ID2TERM, TEXT, SPO, POS, and OSP). This constraint was satisfied by introducing a synchronization mechanism based on a counter. The counter was incremented for each RDF Value written by a given parser/loader task and decremented when the result was available for that RDF Value.

The use of asynchronous writes for the distributed data loader easily doubles the throughput of the architecture when compared with synchronous RPC.

Distributed Query

Bigdata® uses Sesame 2³⁰ for SPARQL processing, but overrides query evaluation for efficiency. The integration point is the Storage And Inference Layer (SAIL) API, which is essentially a pluggable backend for the Sesame 2 platform. Sesame 2 parses SPARQL queries and generates an operator tree. The bigdata® SAIL implementation transforms the operator tree derived from the SPARQL query into custom operators which are mapped directly onto the bigdata® native rule model.

The bigdata® native rule execution model supports conjunctive query across arbitrary relations, filters on individual join dimensions and “optional” joins. The B+Tree implementation supports fast key-range counts and the join planner makes use of those range counts to re-order the joins for more efficient evaluation. These rules are used

³⁰ Sesame2, <http://www.openrdf.org/>

internally to express the RDF Schema model theory and for the evaluation of SPARQL queries. The rule model is extensible.

We have evaluated two join evaluation strategies to date. The first join strategy, *nested subquery*, was developed for a standalone triple store and has performance for that architecture that is competitive with commercial triple stores. However, RMI overhead and concurrency control required for access to remote indices makes this approach impractical for distributed query. Therefore we designed the *pipeline* join strategy which is designed to maximize the opportunity for local computing.

Bigdata® scale-out indices are broken down by key-ranges into index partitions. Therefore, the data for any given join dimension will be found in exactly those index partitions spanned by the key-range for the query pattern. Those index partitions are distributed across the cluster. One approach to this problem is to perform gathered reads. In a gathered read, the tuples which satisfy the query pattern streams back to a join task. Thus all results are materialized at the join task. This approach was taken by YARS³¹, a hash-partitioned scale-out RDF database. We decided against this approach because we thought that we would do better by distributing the computation to the data, rather than gathering the data to the computation (the join task).

Therefore we developed another join strategy, which we call a *pipeline join*. For each join dimension, the pipeline join submits a task to each data service having an index partition spanned by the query pattern. Rather than issuing queries against remote indices, the pipeline join passes binding sets between join processes.

When the client submits a query, a join master task is executed for the query. The join master merely coordinates the work to be performed, but does not “compute” anything. When a rule is evaluated as a query, the results are streamed back to a buffer in the join master task and the client drains results asynchronously from that buffer. When a rule is evaluated as a mutation operation, for example, when performing closure, the join task for the last join dimension writes on a buffer which writes the generated solutions onto the appropriate index partitions without causing them to be materialized on the join master.

A sample pipeline join query is illustrated in Figure 14 for LUBM Q1. The original query is:

```
SELECT ?x WHERE {
  ?x a ub:GraduateStudent ;
     ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>.
}
```

This query has two triple patterns: (?x, rdf:type, ub:GraduateStudent) and (?x, ub:takesCourse, <...GraduateCourse0>). When those triple patterns are resolved against the lexicon, the RDF Values are replaced with the corresponding internal 64-bit term

³¹ Andreas Harth, Stefan Decker. “[Optimized Index Structures for Querying RDF from the Web](#)”. 3rd Latin American Web Congress, Buenos Aires - Argentina, Oct. 31 - Nov. 2 2005.

identifiers, yielding (?x, 8, 256) and (?x, 400, 3048), where 8, 256, 400, and 3048 are the values assigned to the URIs in the original query. The join plan for the query considers the range counts for the two triple patterns, reorders them, and assigns them to specific access paths. The final form of the query for evaluation is query :- pos(x 400 3048) ^ spo(x 8 256). The first join dimension will be evaluation against the POS index. The second join dimension will be evaluated against the SPO index.

For the sake of this example, we assume that all tuples for the first join dimension are located within POS#2³² while the 2nd join dimension ranges across SPO#1, SPO#2, and SPO#3. For the 1st join dimension, the master creates a join task whose access pattern is pos(x,400,3048) on a data service having data for POS#2. For the 2nd join dimension, the master creates a task on data services having data for SPO#1, SPO#2, and SPO#3.

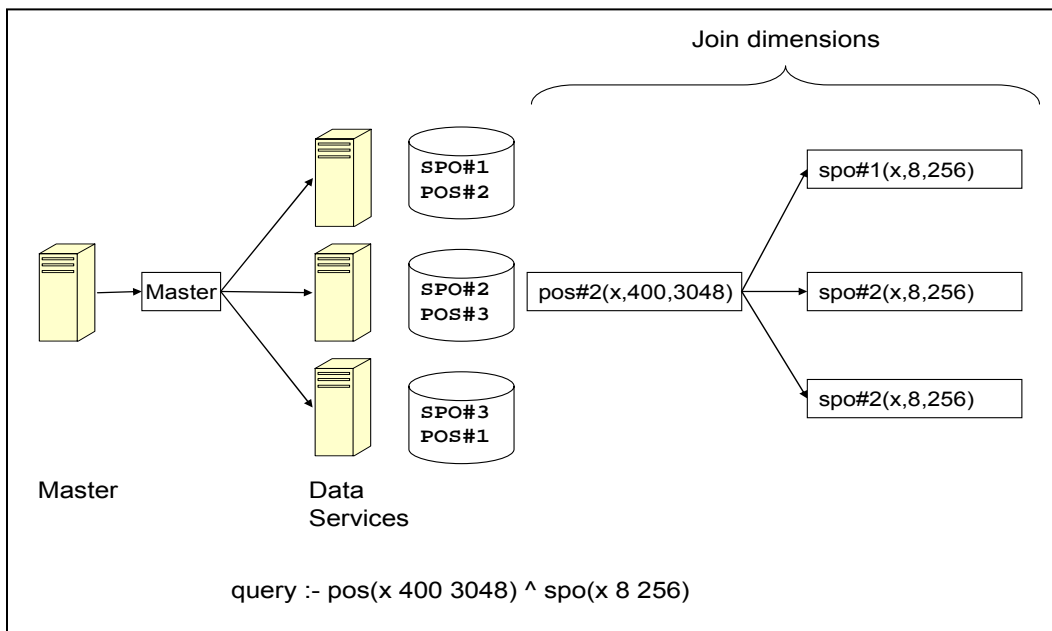


Figure 14 -- Pipeline join example for LUBM Q1.

Query execution begins when the master passes an initial binding set to tasks in the first join dimension. In this case, this binding set is empty. The first join dimension then reads on the local index partition POS#2, combines the data read from that access path with the input binding set, and pushes chunks of binding sets to the downstream join dimension. For each binding set, the upstream join dimensions considers the index partitions which will be spanned by that binding set and pushes the binding set to the appropriate join tasks in the down stream join dimension.³³ The last join dimension

³² Index partition identifiers begin at zero and are 32-bit integers. Each time an index is split, moved, or joined, its old index partition identifier is retired and a new one is assigned. POS#1, POS#2, and POS#3 might be created as the result of a scatter split of POS#0 on a cluster with three data services.

³³ For this example, ?x will be bound by the first join dimension, so the 2nd join dimension will consist of point tests and each binding set will cause a read on exactly one of the downstream join dimensions, but this is not true in general.

writes on a buffer. For query, that buffer is a proxy for a buffer object on the client issuing the query^{34 35}. In this manner, the client sees only the data which satisfy the query. Incomplete solutions are filtered before they reach the client.

Performance results for the pipeline join are encouraging. The algorithm is nearly two orders of magnitude more efficient than the naïve nested subquery join using RMI to read on the remote indices. Query performance against a distributed database using pipeline joins on three commodity servers outperforms, in several cases by a significant margin, query performance on a purely local triple store. Equally, the pipeline join algorithm is efficient on a single server without RMI overhead – the pipeline join nearly always dominates the performance of the nested subquery join.

Inference, truth maintenance and datalog

RDF model theory defines various entailments. The entailments are triples *not* explicitly given in the input, but the database must behave *as if* those triples were present. There are broadly speaking two ways of handling such entailments. First, they can be computed up-front when the data are loaded and stored in the database alongside the explicitly given triples. This approach is known as *eager closure* because you compute the closure of the model theory over the explicit triples and materialize those *inferred* triples in the database. The primary advantage of eager closure is that it materializes all data, both explicit and inferred, in the database which greatly simplifies query planning and provides equally fast access paths for entailed and explicit statements. Eager closure can be extremely efficient, but there can still be significant latency, especially for very large data sets, as the time to compute the closure is often on the order of the time to load the raw data. The other drawback is *space* as the inferred triples are stored in the indices, thereby inflating the on disk size of the data set.

The second approach is to materialize the inferences at query time. This has the advantage that the data set may be queried as soon as the raw data have been loaded and the storage requirements are those for just the raw data. There are a variety of techniques for doing this, including *backward reasoning*^{36 37}, which is often used in Prolog systems, and *magic sets*³⁸, which is often used in datalog systems. Magic sets or similar techniques are typically used in database systems because they lend themselves to *set-at-a-time* rather than *tuple-at-a-time* processing. In an in-memory system, the fast random access time means that backward reasoning can be highly effective. However, database systems can be orders of magnitude more efficient when they are performing set-at-a-time operations.

³⁴ A proxy is a remote object. Jini is used to export the proxy and invoke methods on the remote object.

³⁵ For mutation operations, e.g., when computing the eager closure of the knowledge base, the buffer writes onto the appropriate indices and the data does not flow through the master.

³⁶ Robinson, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23-41.

³⁷ Cohen, J. 1996. Logic programming and constraint logic programming. *ACM Comput. Surv.* 28, 1 (Mar. 1996), 257-259.

³⁸ J. D. Ullman, Bottom-up beats top-down for datalog, Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, p.140-149, March 1989, Philadelphia, Pennsylvania, United States.

Magic sets is a program rewrite technique. It accepts a program, in this case consisting of the RDF Schema entailment rules and the user's query, and returns a re-write of the program in which magic predicates have been introduced. The role of the magic predicates is to prevent rules from being triggered which have no bearing on the query. During the evaluation of the program, solutions for magic predicates may be found, which may cause rules guarded by those magic predicates to become enabled. Just like eager closure, magic sets computation proceeds until a *fixed point* – the point when no new data is produced by applying the rule sets to a view comprised of the database and the generated inferences.

RDF databases which utilize an *eager closure* strategy face another concern. They must maintain a coherent state for the database, including the inferred triples, as data are added to or removed from the database. This problem is known as *truth maintenance*. For RDF Schema, truth maintenance is trivial when adding new data. However, it can become quite complex when data is retracted (deleted) as a search must be conducted to determine whether or not inferences already in the database are still *entailed* without the retracted assertions.

Once again, there are several ways to handle this problem. One extreme is to throw away the inferences, deleting them from the database, and then re-compute the full forward closure of the remaining statements. This has all the drawbacks associated with eager closure and even a trivial retraction can cause the entire closure to be re-computed. Second, truth maintenance can be achieved by storing *proof chains* in an index³⁹. When a statement is retracted, the entailments of that statement are computed and, for each such entailment, the proof chains are consulted to determine whether or not the statement is still proven without the retracted assertion. However, storing the proof chains can be cumbersome. Third, magic sets once again offer an efficient alternative for a system using eager closure to pre-materialize inferences. Rather than storing the proof chains, we can simply compute the set of entailments for the statements to be retracted and then submit queries against the database in which we inquire whether or not those statements are still proven.

bigdata® supports a hybrid approach in which the eager closure is taken for some RDF Schema entailments while other entailments are only materialized at query time. This approach is not uncommon among RDF databases. In addition, bigdata® also supports truth maintenance based on storing proof chains. We are currently integrating the IRIS⁴⁰ open source datalog reasoner, which provides support for magic sets. This integration will provide two valuable new capabilities. First, we will be able to support query time materialization of inferences without eager closure. This means that bigdata® will be able to answer queries on very large data sets as soon as the raw data have been processed. Second, bigdata® will be able to combine eager closure with truth

³⁹ J. Broekstra, A. Kampman, Inferencing and Truth Maintenance in RDF Schema : Exploring a naive practical approach, <http://www.openrdf.org/doc/papers/inferencing.pdf>, in Workshop on Practical and Scalable Semantic Systems (PSSS), 2003.

⁴⁰ IRIS, <https://sourceforge.net/projects/iris-reasoner/>

maintenance based on magic sets and thereby avoid having to store proof chains in the database.

Provenance

Provenance requirements vary widely across applications. For some applications, it is sufficient to know the “source” of the document containing the assertions. However, provenance is a more general concept than just “source.” For example, provenance may include the author, confidence, algorithm, date, or other metadata. For many domains it is critical to know the *provenance* of each datum, that is, statement level provenance. Within RDF, there are two mechanisms which may be used to capture provenance: RDF reification and named graphs. Bigdata® offers a third mechanism based on the notion of statement identifiers⁴¹ designed to support applications with statement level provenance requirements.

RDF reification creates a model of a statement^{42 43}. You can then use that model to make assertions about the model. For example,

```
<_s1, subject, mike>  
<_s1, predicate, memberOf>  
<_s1, object, SYSTAP>  
<_s1, type, Statement>
```

is a model of the statement

```
<mike, memberOf, SYSTAP>
```

Using that model, you can then make statements about `_s1`. However, these are statements about the model, not about the original statement. Further, the statement model includes 4 new statements, which is a significant growth in both the size and the complexity of the data on the disk. Finally, in order to query the statements about the model, you have to write queries against the “reified” form of the statements, which introduces a considerable cognitive burden and significantly increases the computational cost of query answering.

A second approach is named graphs, as standardized by SPARQL. Named graphs provide a mechanism for placing statements within a context, the named graph. SPARQL then provides mechanisms for querying either the RDF Merge of the named

⁴¹ Statement identifiers reflect a concern best articulated in Topic Maps as the ability to make assertions about anything, even other assertions. Topic Maps are not less concerned with model theory and entailments and focus more on an architecture for making assertions about subjects, including that two resources may identify the same subject.

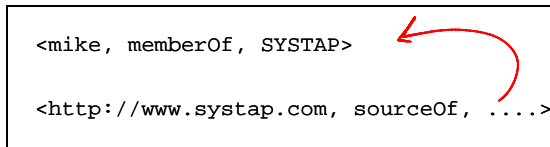
⁴² RDF Semantics, <http://www.w3.org/TR/rdf-nt/>

⁴³ RDF was shaped by the constraints of first order predicate logic. Allowing statements about statements into RDF model theory shifts that logic from first order predicate calculus, which does not permit statements about statements, into second order predicate calculus. The Semantic Web has steered clear of second order predicate calculus in order to avoid some pitfalls associated with previous knowledge representation frameworks.

graphs in a quad store, in which case the context is discarded, or for querying one or more specific named graphs, in which case the context may be discovered by the query.

Named graphs are suitable for situations where it is possible to make assertions about a collection of statements, which might be termed coarse grained provenance. Named graphs are more space efficient than RDF reification, requiring only twice as many indices as a pure triple store, and many queries may be executed fully as efficiently against named graphs as against a pure triple store.

However, what we actually want to say is:



Using statement identifiers (also known as SIDs), this is exactly what we do. In this example, `_s1` is the statement identifier.

```
<mike, memberOf, SYSTAP, _s1>
<http://www.systap.com, sourceOf, _s1>
```

Statement identifiers look just like blank nodes. They are unified with other occurrences of the same blank node when used in an RDF/XML document⁴⁴. Internally, the statement identifier is assigned by the Lexicon using a key formed from the term identifier for the subject, predicate, and object of the statement. Statement identifiers are therefore uniquely associated with a *triple*.

Statement identifiers may become bound in SPARQL queries, thus serving to connect a statement with statements about that statement in a transparent manner.

```
construct { ?s <memberOf> ?o . ?s1 ?p1 ?sid . }
where {
  ?s1 ?p1 ?sid .
  GRAPH ?sid { ?s <memberOf> ?o }
}
```

Statement identifiers are an ideal solution for an application with a *fine-grained* provenance architecture requirement. Unlike reification or named graphs, statement identifiers do not significantly increase the size of the data on disk⁴⁵. Neither do they increase the cognitive complexity of formulating provenance queries nor the computational cost of answering those queries.

⁴⁴ Bigdata® supports an extension to RDF/XML for this purpose.

⁴⁵ Explicit statements are redundantly encoded in the TERM2ID index. This is equivalent to carrying two copies of the SPO index. In contrast, a quad store uses twice as many statement indices while reification requires four times as many statements.

Each of the models described here has its use. A pure triple store has the minimum on disk footprint and is suitable when there are no provenance requirements. Named graphs are appropriate for a wide range of applications and also work well for applications with coarse grained provenance requirements. Statement identifiers support applications with fine-grained provenance requirements, and do so with the minimum disk footprint, cognitive burden, and computational cost.

Performance evaluation

Evaluation platform

The distributed data load and data query performance evaluation was performed on a cluster of 15 Dell blades. Each blade was a Quad Core (4x512K Cache, 2.3GHz) server with 32G of RAM and 146G of local SAS3 disk with 10K spindles. The operating system was Red Hat Fedora Core 10. Java 1.6.0_14 was used for the evaluation runs reported here.

Of the 15 blades, 10 were used to host data services. 8 of the blades were used to run distributed data load clients, and were unused during query. The remaining blade was used to run all other services and also hosted a NFS volume mounted by the other blades in the cluster.

Bigdata® instruments and collects metrics on the operating system (CPU, RAM, DISK), on each bigdata® process (per process measures of CPU, RAM, and DISK), and on the process internals for bigdata® processes. These measures are aggregated at the load balancer, and logged for analysis. The logged events are readily accessible using HTTP from either a web browser or a web-enabled worksheet. Custom renderings of various data may be automatically generated and rendered as DHTML. This end-to-end instrumentation has made it possible to analyze the total cluster performance and identify and rectify bottlenecks. Similar end-to-end instrumentation is used in other grid computing networks⁴⁶.

For the purposes of evaluation, we decided to work with the LUBM⁴⁷ synthetic data set. The characteristics of the LUBM data set and its queries are well known, are well suited to evaluating common JOIN algorithms, and performance benchmarks are posted for most RDF databases in terms of this data set.

The pre-generated RDF/XML data size (on disk) exceeds the space demands for the indices themselves. For larger trials, this on disk requirements exceeded the shared volume storage available for the cluster. Therefore, in consideration of the desired data scale, we decided to modify the LUBM generator so as to performance incremental generation of the data set. Each client ran a generator instance. There were N clients, and each client was responsible for generating 1/Nth of the total data set. This was achieved by taking the university module the #of clients and generating the data for a

⁴⁶ I. Legrand, R. Voicu, C. Cirstoiu, C. Grigoras, L. Betev, A. Costan, Monitoring and Control of Large Systems With MonALISA, Communications of the ACM Vol. 52 No. 9, Pages 49-55.

⁴⁷ The Lehigh University Benchmark, <http://swat.cse.lehigh.edu/projects/lubm/>.

given university iff the modulo function evaluated to the unique client number [0:N-1]. The pathnames of the generated RDF/XML files for each university were fed into a queue within that client, and the queue in turn was fed into a thread pool running RDF/XML parser/loader tasks.

Data load performance

Figure 15 shows data load performance for the LUBM U8000 synthetic data set. The load of the raw RDF/XML files, containing over one billion assertions, was completed in less than 60 minutes. This is an aggregate rate of 270,000 triples per second across the data load operation and of approximately 29,000 triples per second per data service for each of the 10 servers running data service processes. The client throughput was closer to 70,000 triples per second for each of the 8 servers running the client side of the distributed data load operation (reading and parsing RDF/XML and performing scattered writes on the distributed indices).

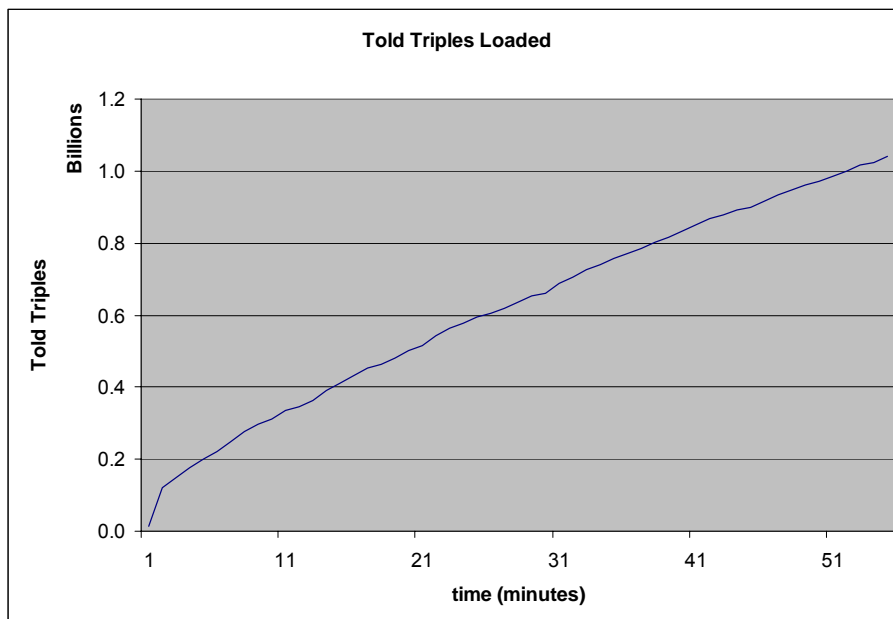


Figure 15 -- Data load performance for LUBM U8000.

The elbow in Figure 15 at approximately 3 minutes is caused by the onset of the scatter splits, see Figure 16. At first, all data is being absorbed by journals on blades 5, 10, 11, 13, 14. These data services then undergo scatter splits, redistributing the data from the initial index partitions to new index partitions on data services on each machine in the cluster. This can be seen in Figure 16, which plots split and move events. Within each event type (split or move), the data are grouped by the host for which the event was reported. Each event is indicated by a line on the chart with a small circle indicating the time at which the split or move operation commenced and another circle indicating the time at which the operations was completed. During a scatter split, the split and move operations proceed with a high degree of concurrency, leading to clumping of the events in the chart.

In the top half of the Figure 16, we see the index partition splits, which are performed on the 5 data services on which the index partitions were first allocated (blades 5, 10, 11, 13, 14). While this is not evident from the embedded graph, flyovers in the DHTML reveal the indices were split in the following sequence: ID2TERM, TERM2ID, OSP, POS, and finally SPO. Beneath each split event, there are a series of “move” events in which the resulting index partitions are redistributed to data services on other hosts in the cluster. The moves are recorded under the host receiving the index partition.

The split events for each host occur in two distinct bunches. The first bunch for any given host consists of the “merge” operations. The second bunch is the atomic update, during which the locator for the old index partition is retired and a locator for the new index partition is assigned. After the atomic update, clients will be notified that their index partition locator is “stale” and will query the metadata service, obtaining the new index partition locator(s) for each key-range of the index which they need to address. After approximately 3 minutes, all 10 machines hosting data services are hosting partitions for each of the 5 RDF database indices. This ends the “scatter split.”

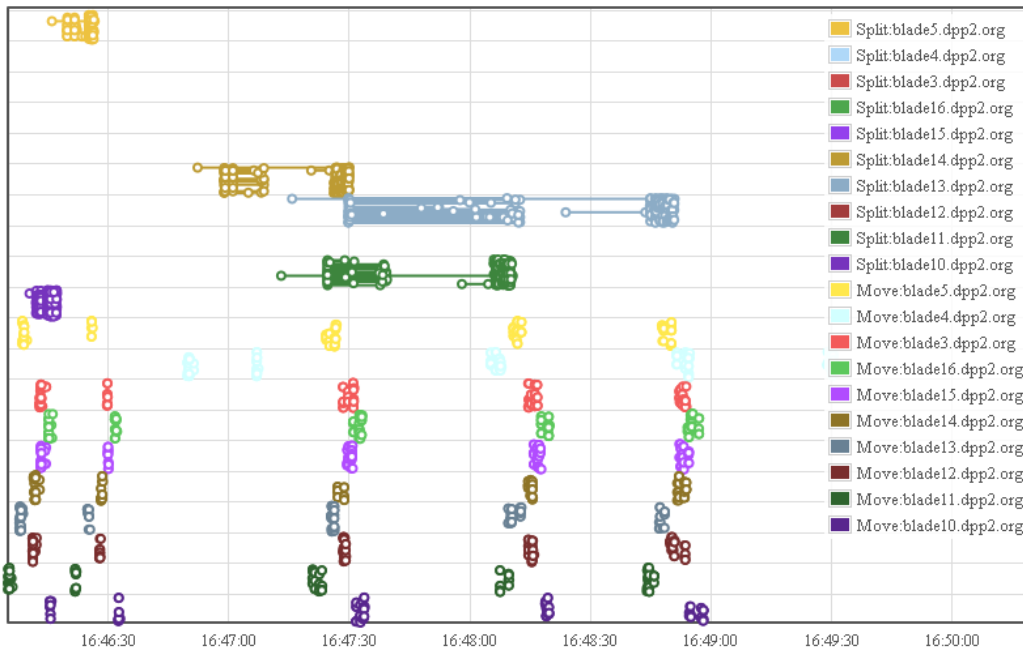


Figure 16 -- Scatter split for LUBM U8000.

After the initial scatter split, the only index partition builds and merges are performed until approximately 40 minutes into the run. At that time ID2TERM index partitions were split again and a few index partitions were moved to other hosts. This occurred over a period of approximately 6 minutes.

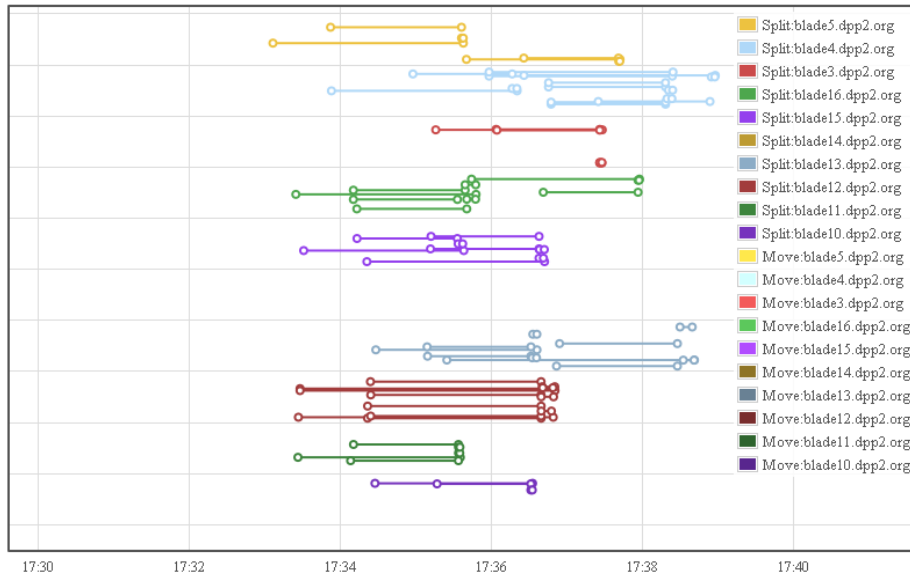


Figure 17 -- Split of the OSP index partitions at 40 minutes into the run.

When loading billions of triples, each of the indices will be split repeatedly in order to keep the size of each index partition to no more than approximately 200 MB. Because the index partition writes are well distributed, both split and merge operations tend to occur at nearly the same time for all partitions of a given index. Split and merge operations for full index partitions are relatively heavy operations. Because these operations come due at very nearly the same time for all partitions of a given index, the overall write throughput of the cluster is diminished while the index partitions are processed. In the future, we will schedule split and merge operations proactively in order to distribute the effort of those operations more uniformly when there are 100s of index partitions on each server.

Disk utilization

Disk utilization for RDF is illustrated in Figure 18, which plots the total bytes under management for the federation against the bytes per triple during an LUBM U8000 data load. For this evaluation, the minimum release age was set to zero, effectively disabling the immortal database. Disk space requirements therefore increase as new data are buffered on the journal, increase as new index segment files are generated, and decrease as the journals and index segment files associated with older commit points are released. This can be seen in the “Bytes Under Management” line in the graph. The total disk requirements of course trend upwards as new data are loaded.

In contrast to the increasing on disk requirements, the “Bytes per Told Triple” decreases over time, reaching approximately 100 bytes per told triple by the end of this data set. In fact, observed value of bytes per triple converges to a steady state of approximately 57 bytes per triple by the time 2-3 billion triples are loaded. This steady state value has been observed out to 13 billion triples. The increasing storage efficiency is due to several factors, including the increasing percentage of the data in index segment files, as opposed to the data buffered on journals, the ongoing elimination of historical revisions of B+Tree

nodes and leaves by index partition merge⁴⁸ operations, and the elimination of journals and index segments stored solely for access to historical commit points.⁴⁹

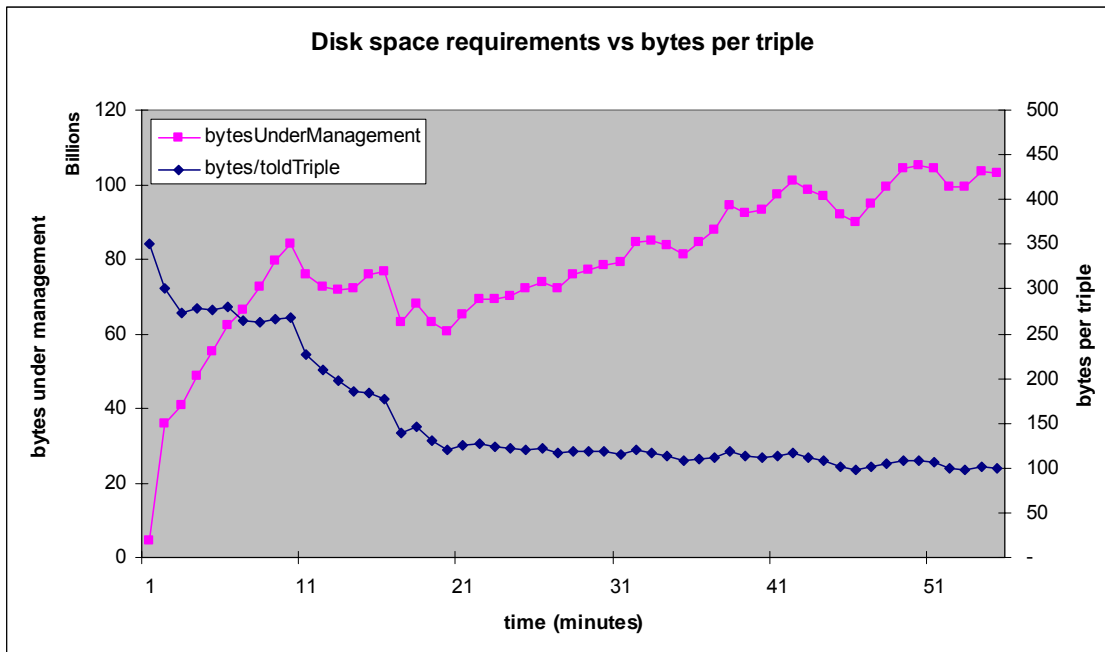


Figure 18 -- Disk space utilization for RDF.

Distributed Query performance

Query performance is the current focus of our investigations for the clustered bigdata® database. Single machine query performance is comparable to commercial triple stores. On small data sets, distributed query on 15 machines was nearly twice as fast as query on a single machine without RMI. We plan to publish details on distributed query performance shortly.

Related work

Many distributed database architectures use hash-partition indices. In this approach, the hash function is computed for each key and taken module the number of machines in the cluster. The result is the index of the machine on which that key would be stored. This approach has been applied successfully to large amounts of RDF data extracted from web pages⁵⁰. Hash-partitioned indices have the advantage of being simpler to construct but

⁴⁸ An index partition build operation copies only the last committed state of the index partition from the journal onto an index segment. In contrast, an index partition merge produces a compact view of the index in which all duplicate tuples have been eliminated. Build operations are approximately ten times as common as merges and are less expensive. However, merge operations can significantly reduce the on disk requirements for the index partition.

⁴⁹ The index segment files are optimized B+Trees storing only the state of the index partition as of a specific commit point while journals store multiple revisions of B+Tree nodes and leaves in order to preserve historical commit points. This makes the index segment files much more space efficient.

⁵⁰ A. Harth, J Umbrich, A. Hogan, S. Decker, YARS2: A Federated Repository for Querying Graph Structured Data from the Web, 6th International and 2nd Asian Semantic Web Conference (ISWC2007+ASWC2007), 2007.

have several disadvantages. First, the computing resources must be specified in advance and a fixed mapping (the hash function) governs the allocation of the data onto the machines. Second, key-range queries must be flooded to all machines in the cluster – this means that nearly all joins touch all machines. In contrast, a system which utilizes dynamic key-range index partitioning, such as bigdata®, can fit the data to the available hardware, dynamically accommodate additional resources, and issue key-range queries to precisely those machines which span the data of interest.

Bigdata® also differs from some hash-partitioned systems and from attempts to use map/reduce processing for database operations in that its high throughput is obtained for sustained index writes. Map/reduce processing requires that the reduce host wait until all data is available on the reduce host, at which point the reduce operation generates the hash-partitioned index locally on that host. Bigdata is capable of sustained high data load rates with incremental write onto the backing indices. Because of its MVCC architecture, bigdata can also provide consistent views of historical database states. This makes it possible to query data from a historical commit point while high-speed data loads proceed concurrently.

Conclusion

Bigdata® was designed from the ground up as a scale-out architecture for commodity hardware and is capable in theory of transparently and incrementally scaling to 1000s of servers hosting trillions of triples. A trillion triples could be managed by a data center with 512 servers, each hosting 2 billion triples, and would require 60 terabytes of replicated data. In trials to date we have put 100s of billions of triples safely on disk and achieved throughput rates of over 300,000 triples per second on a cluster of 15 machines for the first 1B triples, with 3.3B triples loaded at a rate of 100,000 triples per second, the fastest known throughput for an RDF database. With 13 billion triples loaded, bigdata® is currently the second largest known knowledge base instance in the world.