Bigdata: Approaching web scale for the semantic web.

Abstract

We introduce bigdata®, a horizontally scaled distributed B+Tree database, discuss its architecture and its application to develop a capability for web-scale semantic graphs. Unlike many recent distributed index schemes, bigdata® uses dynamic key-range partitioning and can maintain pace with the data scale by incrementally adding more computational resources. Unlike recent work on column flexible row stores, bigdata is designed to support efficient distributed query processing.

Keywords: RDF, Semantic Web, Scale-out, Distributed, Database, Mashup

Motivation

TODO: … web-scale mashups … dynamic schema-flexible database … standards-based query language

Approach

The problem that we have set has several interesting dimensions. First, we require the ability 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 require the ability to dynamically align the schema for those data sets. Third, we require the ability to track the provenance of each datum. Our response to this problem is (a) the bigdata® scale-out architecture (http://www.bigdata.com/blog, http://www.sourceforge.net/projects/bigdata), which is designed for commodity hardware clusters; (b) the Resource Description Framework (RDF); and (c) exploiting the range of available semantics for the "graph" or "context" position in a SPARQL endpoint to provide datum-level provenance with high-level query.

The rest of this paper is organized as follows:

1. The bigdata® architecture
    a. B+Trees
    b. Index partitions
    c. Metadata Service
    d. Data Service
    e. Concurrency Control
    f. Service Architecture
    g. Deployment
2. RDF
    a. RDF database; full text index, and provenance architecture
    b. Bulk data load; master / clients and data services.
    c. Query; SPARQL, rules and native join evaluation.

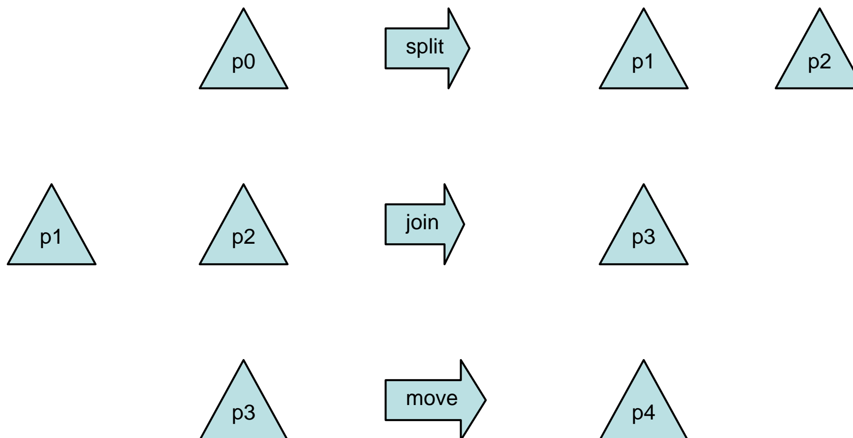
© SYSTAP, LLC 2006-2009, All rights reserved

   d. Truth Maintenance & Datalog
3. Results
4. Conclusion

Architecture

Bigdata® is a scale-out architecture for persistent, ordered data. The overall approach which we have chosen utilizes key-range partitioned B+Tree [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] indices distributed across the resources of a cluster. A bigdata® 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 the (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 "delete flag" which is used to prevent reads through to historical data in index views, discussed below, and a version timestamp, which supports optional transaction processing based on Multi-Version Concurrency Control (MVCC, Reed, D.P.. "Naming and Synchronization in a Decentralized Computer System". *MIT dissertation*. http://www.lcs.mit.edu/publications/specpub.php?id=773).

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.



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*

© SYSTAP, LLC 2006-2009, All rights reserved

(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.

The index partition *locators* are persisted as tuples in a *metadata index* that lives on a specialized data service known as the *metadata service*. An index partition locator maps a key-range onto an index partition identifier and a (logical) data service identifier. The key for the tuples in the metadata index is simply the first key that could enter the corresponding index partition. Bigdata® has a theoretical upper bound of 400 exabytes per scale-out index (based on 32-bit partition identifiers and 200M index partitions). Bigdata can address petabytes of index data today. In order to realize the theoretical upper bound, bigdata® will need to manage a partitioned metadata index and reclaim old partition identifiers.

Each index partition is assigned to a (logical) *data service*. 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 service. 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 retiring the old journal, but they add a requirement to maintain delete markers on tuples in 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. 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 service. A similar design was described in (“Bigtable: A Distributed Storage System for Structured Data”, http://labs.google.com/papers/bigtable.html).

© SYSTAP, LLC 2006-2009, All rights reserved

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 ([],∞) onto that index partition. 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 ([],∞) 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.

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. 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 always span the key range ([],∞) without overlap. Thus each key always falls into precisely one index partition.

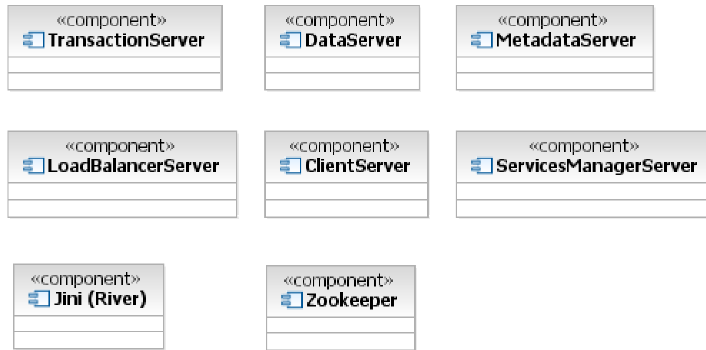
If necessary, applications may place additional constraints on the choice of the separator key. For example, this may be done to ensure that an index partition never splits a logical row. That guarantee may be used to achieve extremely high concurrent write rates using ACID but non-transactional operations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

TODO: Discuss additional aspects of concurrency control, including the interpretation of commit timestamps and transaction identifiers, read-only and read-write tx; and the role of the transaction services in managing read locks and their interaction with the history retention policy.

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. 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. The new index partitions are redistributed across the cluster, leaving every Nth 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!

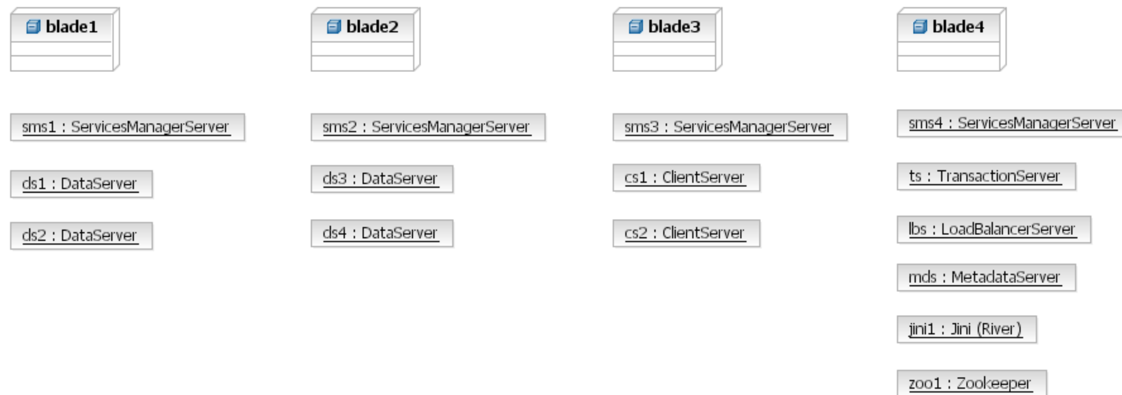
In addition to the metadata service and the data services described above, the bigdata® services architecture defines a *transaction service*, a *load balancer service*, a *services manager* (starts and stops services), and a *client service* (a container for executing distributed tasks). Jini (now the 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 via an integration with the Sesame 2 platform.

© SYSTAP, LLC 2006-2009, All rights reserved



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 volume. The script will start or stop bigdata service managers and associated components as necessary to achieve the desired run state.

A sample configuration for four nodes is illustrated below. While the zookeeper documentation suggests that it should have a dedicated machine, in our experience it was a lightly utilized component and was placed for the purposes of evaluation with other relatively lightly utilized components.



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

© SYSTAP, LLC 2006-2009, All rights reserved

An alternative 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/or increase cost.

Resource Description Framework

TODO: A brief summary of RDF.

RDF Database Architecture

TODO: This section needs some examples to make it clear

RDF is a collection of triples, defining the arcs of a graph whose nodes are RDF Resources (people, places, things, events and ideas) and whose arcs describe named relationships between those nodes (RDF Predicates). In addition, each node may have a collection of attribute data describing that node. Likewise, for our purposes, RDF query is based on *triple 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.

In the bigdata® RDF database, the RDF Values are fully normalized by 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 Literals onto the internal identifiers and may be used to perform search against the triple store. The keys for the TERM2ID index are formed such that RDF datatype literals and RDF language coded literals fall each within their own key-range. Thus, a key-range query may be formed for all RDF Literals whose type is xsd:int and the order of the tuples in the database will directly correspond to natural order of the xsd:int domain.

The RDF database architecture defines a perfect access path for each possible triple pattern. These are the SPO, POS, and OSP indices. The name of the index indicates the manner in which the Subject, Predicate, and Object have been (re-)ordered. The keys are formed by concatenating the internal identifiers assigned by the TERM2ID index in the order corresponding to that index. The value for the statement index indicates whether the triple is *explicit* (given in some source data); *inferred*, or an *axiom*. These flags are used for truth maintenance, described below. When the provenance mode is enabled, the TERM2ID index assigns unique statement identifiers in precisely the same manner as it assigns identifiers for RDF Values and the assigned statement identifier is stored as part of the value of the tuple. For this purpose it uses the internal identifiers in the SPO key ordering.

© SYSTAP, LLC 2006-2009, All rights reserved

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.

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

For the purposes of evaluation, we decided to work with the LUBM [x] synthetic data set. While the data only superficially resemble our target domain, 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. 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 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.

The parser/loader tasks feed *shared, asynchronous write buffers*, which write on the various indices in the RDF database instance. These buffers are shared in that all parser/loader tasks in the same client write onto the same buffers, thus writes are combined across tasks. 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, the asynchronous write buffer 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). Third, duplicate tuples emerging from concurrent tasks can be easily filtered out. Fourth, the asynchronous write buffer ensures that all remote index writes have a good “chunk” size (10,000 or more tuples), which translates 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, which result in substantially less DISK IO for chunkier writes).

© SYSTAP, LLC 2006-2009, All rights reserved

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

TODO: Discuss how we form the join plan and native rule execution, including the pipeline join algorithm.

Bigdata® uses Sesame 2 (Sesame2, http://www.openrdf.org/) for SPARQL processing but overrides query evaluation for efficiency. The integration point the Sesame 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 model supports conjunctive query across arbitrary relations. 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 internally to express the RDF Schema model theory and for the evaluation of SPARQL queries.

TODO: Sample rule based on SPARQL query, perhaps for LUBM, and the re-ordered JOINs for query evaluation. This can be threaded through the description including the original SPARQL query above and its parse tree.

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, the RMI overhead involved makes this approach impractical for distributed query. Therefore we designed *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 YARs, a

© SYSTAP, LLC 2006-2009, All rights reserved

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.

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.

Preliminary performance results for the pipeline join are encouraging. Query performance against a distributed database running on three commodity servers outperforms, in several cases by a significant margin, query performance on a purely local triple store. Overall, distributed query on 15 machines was nearly twice as fast as query on a single machine without RMI.

TODO: results on various data sets.

A *bloom filter* is an in memory data structure that can very rapidly determine whether a fully bound point test is NOT an index hit. 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 #of index entries in the mutable B+Tree exceeds about 2M tuples. BUT, they are great for scale-out since the data on the mutable B+Tree is migrated into perfect index segments, and we generate perfect fit bloom filters for those index segments. Every time we overflow a journal, we wind up with a new (empty) B+Tree to absorb writes, so the bloom filter is automatically re-enabled. Since the #of tuples in an index segment is known, we generate a perfect fit bloom filter each time we generate a new index segment. This provides a dramatic boost for distributed query and is yet another way in which scale-out can leverage more resources than a scale-up architecture.

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

© SYSTAP, LLC 2006-2009, All rights reserved

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. 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* [x], which is often used in Prolog systems, and *magic sets* [x], 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* ("Top down beats bottom up for datalog", http://portal.acm.org/citation.cfm?id=73736) 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.

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 added 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 index ("Inferencing and Truth Maintenance in RDF Schema", http://www.openrdf.org/doc/papers/inferencing.pdf). When a statement is retracted, the

© SYSTAP, LLC 2006-2009, All rights reserved

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.

As of this writing, 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 an open source datalog reasoner (IRIS, https://sourceforge.net/projects/iris-reasoner/) with support for magic sets. This integration will provide two valuable new capabilities. First, we will be able to support both 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 maintenance based on magic sets and thereby avoid having to store proof chains in the database.

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 pages, (YARS2, http://sw.deri.org/2007/02/swsepaper/iswc2007.pdf). Hash-partitioned indices have the advantage of being simpler to construct but have several disadvantages. First, the computing resources available 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. 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. However, it can also produce a series of consistent views identified by timestamps corresponding to commit points of interest. This makes it possible to query data from a historical commit point while high-speed data loads proceed concurrently.

© SYSTAP, LLC 2006-2009, All rights reserved

Results

TODO: Summarize results on various benchmarks.

Conclusion

TODO: Conclusion.

© SYSTAP, LLC 2006-2009, All rights reserved